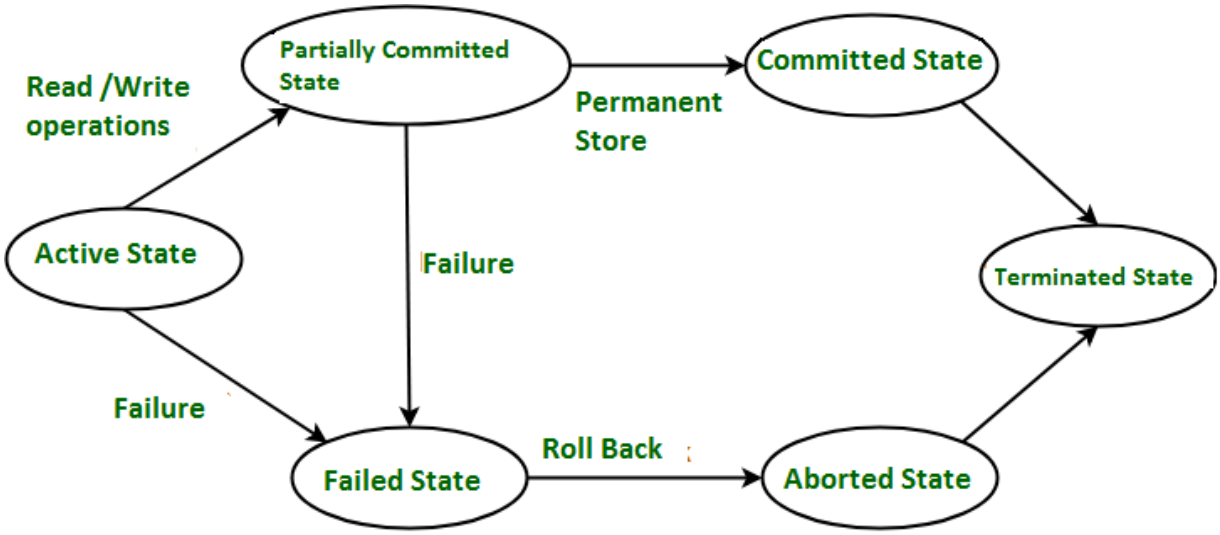
1. **Transaction State:**

States through which a transaction goes during its lifetime. These are the states which tell about the current state of the Transaction and also tell how we will further do the processing in the transactions. These states govern the rules which decide the fate of the transaction whether it will commit or abort.

They also use **Transaction log.** Transaction log is a file maintain by recovery management component to record all the activities of the transaction. After commit is done transaction log file is removed.



Transaction States in DBMS

These are different types of Transaction States :

1. **Active State –**
   When the instructions of the transaction are running then the transaction is in active state. If all the 'read and write' operations are performed without any error then it goes to the "partially committed state"; if any instruction fails, it goes to the "failed state".

2. **Partially Committed –**
   After completion of all the read and write operation the changes are made in main memory or local buffer. If the changes are made permanent on the DataBase then the state will change to "committed state" and in case of failure it will go to the "failed state".

3. **Failed State –**
   When any instruction of the transaction fails, it goes to the "failed state"

or if failure occurs in making a permanent change of data on Data Base.

4. **Aborted State –**
   After having any type of failure the transaction goes from “failed state” to “aborted state” and since in previous states, the changes are only made to local buffer or main memory and hence these changes are deleted or rolled-back.

5. **Committed State –**
   It is the state when the changes are made permanent on the Data Base and the transaction is complete and therefore terminated in the “terminated state”.

6. **Terminated State –**
   If there isn’t any roll-back or the transaction comes from the “committed state”, then the system is consistent and ready for new transaction and the old transaction is terminated.

2. **Implementation of Atomicity and Durability**

Implementation of Atomicity and Durability in DBMS

Atomicity and durability are two important concepts in database management systems (DBMS) that ensure the consistency and reliability of data.

Atomicity:

One of the key characteristics of transactions in database management systems (DBMS) is atomicity, which guarantees that every operation within a transaction is handled as a single, indivisible unit of work.

Importance:

A key characteristic of transactions in database management systems is atomicity (DBMS). It makes sure that every action taken as part of a transaction is handled as a single, indivisible item of labor that can either be completed in full or not at all.

Even in the case of mistakes, failures, or crashes, atomicity ensures that the database maintains consistency. The following are some of the reasons why atomicity is essential in DBMS:

- **Consistency:** Atomicity ensures that the database remains in a consistent state at all times. All changes made by a transaction are rolled back if it is interrupted or fails for any other reason, returning the database to its initial state. By doing this, the database's consistency and data integrity are maintained.

- **Recovery:** Atomicity guarantees that, in the event of a system failure or crash, the database can be restored to a consistent state. All changes made by a transaction are undone if it is interrupted or fails, and the database is then reset to its initial state using the undo log. This guarantees that, even in the event of failure, the database may be restored to a consistent condition.
- **Concurrency:** Atomicity makes assurance that transactions can run simultaneously without affecting one another. Each transaction is carried out independently of the others, and its modifications are kept separate. This guarantees that numerous users can access the database concurrently without resulting in conflicts or inconsistent data.
- **Reliability:** Even in the face of mistakes or failures, atomicity makes the guarantee that the database is trustworthy. By ensuring that transactions are atomic, the database remains consistent and reliable, even in the event of system failures, crashes, or errors.

3. Concurrency control :
Concurrency Control in DBMS

Read

Discuss
Courses

Concurrently control is a very important concept of DBMS which ensures the simultaneous execution or manipulation of data by several processes or user without resulting in data inconsistency. Concurrency Control deals with **interleaved execution** of more than one transaction.

What is Transaction?
A set of logically related operations is known as a transaction. The main operations of a transaction are:

- **Read(A):** Read operations Read(A) or R(A) reads the value of A from the database and stores it in a buffer in the main memory.

- **Write (A):** Write operation Write(A) or W(A) writes the value back to the database from the buffer.

(Note: It doesn't always need to write it to a database back it just writes the changes to buffer this is the reason where dirty read comes into the picture)

Let us take a debit transaction from an account that consists of the following operations:

1. R(A);
2. A=A-1000;
3. W(A);

Assume A's value before starting the transaction is 5000.

- The first operation reads the value of A from the database and stores it in a buffer.
- the Second operation will decrease its value by 1000. So buffer will contain 4000.
- the Third operation will write the value from the buffer to the database. So A's final value will be 4000.

But it may also be possible that the transaction may fail after executing some of its operations. The failure can be because of **hardware, software or power,** etc. For example, if the debit transaction discussed above fails after executing operation 2, the value of A will remain 5000 in the database which is not acceptable by the bank. To avoid this, Database has two important operations:

- **Commit:** After all instructions of a transaction are successfully executed, the changes made by a transaction are made permanent in the database.
- **Rollback:** If a transaction is not able to execute all operations successfully, all the changes made by a transaction are undone.

For more details please refer Transaction Control in DBMS article.

Properties of a Transaction
Atomicity: As a transaction is a set of logically related operations, **either all of them should be executed or none**. A debit transaction discussed above should either execute all three operations or none. If the debit transaction fails after executing operations 1 and 2 then its new value of 4000 will not be updated in the database which leads to inconsistency.

Consistency: If operations of debit and credit transactions on the same account are executed concurrently, it may leave the database in an inconsistent state.

- For Example, with T1 (debit of Rs. 1000 from A) and T2 (credit of 500 to A) executing concurrently, the database reaches an inconsistent state.

- Let us assume the Account balance of A is Rs. 5000. T1 reads A(5000) and stores the value in its local buffer space. Then T2 reads A(5000) and also stores the value in its local buffer space.
- T1 performs A=A-1000 (5000-1000=4000) and 4000 is stored in T1 buffer space. Then T2 performs A=A+500 (5000+500=5500) and 5500 is stored in the T2 buffer space. T1 writes the value from its buffer back to the database.
- A's value is updated to 4000 in the database and then T2 writes the value from its buffer back to the database. A's value is updated to 5500 which shows that the effect of the debit transaction is lost and the database has become inconsistent.
- To maintain consistency of the database, we need **concurrency control protocols** which will be discussed in the next article. The operations of T1 and T2 with their buffers and database have been shown in Table 1.

T1	**T1's buffer space**	**T2**	**T2's Buffer Space**	**Database**
				A=5000
R(A);	A=5000			A=5000
	A=5000	R(A);	A=5000	A=5000
A=A-1000;	A=4000		A=5000	A=5000

	A=4000	A=A+500 ;	A=5500	
W(A);			A=5500	A=4000
		W(A);		A=5500

Isolation: The result of a transaction should not be visible to others before the transaction is committed. For example, let us assume that A's balance is Rs. 5000 and T1 debits Rs. 1000 from A. A's new balance will be 4000. If T2 credits Rs. 500 to A's new balance, A will become 4500, and after this T1 fails. Then we have to roll back T2 as well because it is using the value produced by T1. So transaction results are not made visible to other transactions before it commits.

Durable: Once the database has committed a transaction, the changes made by the transaction should be permanent. e.g.; If a person has credited $500000 to his account, the bank can't say that the update has been lost. To avoid this problem, multiple copies of the database are stored at different locations.

4. Serializability-
A schedule is serialized if it is equivalent to a serial schedule. A concurrent schedule must ensure it is the same as if executed serially means one after another. It refers to the sequence of actions such as read, write, abort, commit are performed in a serial manner.

Example
Let's take two transactions T1 and T2,

If both transactions are performed without interfering each other then it is called as serial schedule, it can be represented as follows −

T1	**T2**
READ1(A)	
WRITE1(A)	
READ1(B)	
C1	

	READ2(B)
	WRITE2(B)
	READ2(B)
	C2

Non serial schedule − When a transaction is overlapped between the transaction T1 and T2.

Example

Consider the following example −

T1	**T2**
READ1(A)	
WRITE1(A)	
	READ2(B)
	WRITE2(B)
READ1(B)	
WRITE1(B)	
READ1(B)	

Types of serializability :

There are two types of serializability −

View serializability

A schedule is view-serializability if it is viewed equivalent to a serial schedule.

The rules it follows are as follows −

- T1 is reading the initial value of A, then T2 also reads the initial value of A.
- T1 is the reading value written by T2, then T2 also reads the value written by T1.
- T1 is writing the final value, and then T2 also has the write operation as the final value.

Conflict serializability

It orders any conflicting operations in the same way as some serial execution. A pair of operations is said to conflict if they operate on the same data item and one of them is a write operation.

That means

Readi(x) readj(x) - non conflict read-read operation

Readi(x) writej(x) - conflict read-write operation.
Writei(x) readj(x) - conflict write-read operation.
Writei(x) writej(x) - conflict write-write operation.

Recoverability in DBMS

Recoverability is a property of database systems that ensures that, in the event of a failure or error, the system can recover the database to a consistent state. Recoverability guarantees that all committed transactions are durable and that their effects are permanently stored in the database, while the effects of uncommitted transactions are undone to maintain data consistency.

The recoverability property is enforced through the use of transaction logs, which record all changes made to the database during transaction processing. When a failure occurs, the system uses the log to recover the database to a consistent state, which involves either undoing the effects of uncommitted transactions or redoing the effects of committed transactions.

There are several levels of recoverability that can be supported by a database system:

No-undo logging: This level of recoverability only guarantees that committed transactions are durable, but does not provide the ability to undo the effects of uncommitted transactions.

Undo logging: This level of recoverability provides the ability to undo the effects of uncommitted transactions but may result in the loss of updates made by committed transactions that occur after the failed transaction.

edo logging: This level of recoverability provides the ability to redo the effects of committed transactions, ensuring that all committed updates are durable and can be recovered in the event of failure.

Undo-redo logging: This level of recoverability provides both undo and redo capabilities, ensuring that the system can recover to a consistent state regardless of whether a transaction has been committed or not.

In addition to these levels of recoverability, database systems may also use techniques such as checkpointing and shadow paging to improve recovery performance and reduce the overhead associated with logging.

5. Isolation in DBMS

Introduction:

Isolation is a database-level characteristic that governs how and when modifications are made, as well as whether they are visible to other users, systems, and other databases. One of the purposes of isolation is to allow many transactions to run concurrently without interfering with their execution.

Isolation is a need for database transactional properties. It is the third ACID (Atomicity, Consistency, Isolation, and Durability) standard property that ensures data consistency and accuracy.

ACID:

To maintain database consistency, "ACID properties" are followed both before and after a transaction.

1. **Atomicity:**
   The term atomicity relates to the ACID Property in DBMS, which refers to the notion that data is kept atomic.
   That means that any operation done on the data must be finished entirely or not at all.It also suggests that the operation should not be discontinued or completed only halfway. When working on a transaction, operations should be done completely rather than partially.
   The transaction is canceled if any of the operations is unfinished. When another operation enters with a higher priority, the current operation might be carried out.This terminates the current operation and causes it to be aborted.

2. **Consistency:**
   This ACID Property will make sure that the sum of the remaining seats in the train plus the number of seats that users have reserved will equal the total number of seats in the train. Each transaction ends with a consistency test to make sure nothing goes wrong.
3. **Durability:**
   The term "durability" in relation to DBMS refers to the idea that if an operation is successfully finished, the database remains in the disc forever. The database's resilience should allow it to continue operating even if the system malfunctions or crashes.
   The recovery manager is in charge of guaranteeing the database's long-term viability in the event that it is lost. Every time we make a change, we must use the COMMIT command to commit the values.
4. **Isolation:**
   Isolation is referred to as a state of separation. A DBMS's isolation feature ensures that several transactions can take place simultaneously and that

no data from one database should have an impact on another. In other words, the process on the second state of the database will start after the operation on the first state is finished.

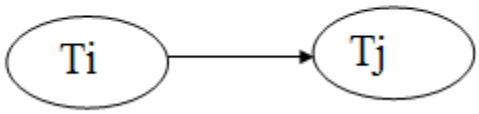
Testing for serializability:
Testing of Serializability

Serialization Graph is used to test the Serializability of a schedule.

Assume a schedule S. For S, we construct a graph known as precedence graph. This graph has a pair G = (V, E), where V consists a set of vertices, and E consists a set of edges. The set of vertices is used to contain all the transactions participating in the schedule. The set of edges is used to contain all edges Ti ->Tj for which one of the three conditions holds:

1. **Create a node Ti → Tj if Ti executes write (Q) before Tj executes read (Q).**
2. **Create a node Ti → Tj if Ti executes read (Q) before Tj executes write (Q).**
3. **Create a node Ti → Tj if Ti executes write (Q) before Tj executes write (Q).**

Precedence graph for Schedule S



- **If a precedence graph contains a single edge Ti → Tj, then all the instructions of Ti are executed before the first instruction of Tj is executed.**
- **If a precedence graph for schedule S contains a cycle, then S is non-serializable. If the precedence graph has no cycle, then S is known as serializable.**

For example:

Time

T1	T2	T3
Read(A)		
	Read(B)	
$A := f_1(A)$		
		Read(C)
	$B := f_2(B)$	
	Write(B)	
		$C := f_3(C)$
		Write(C)
Write(A)		
		Read(B)
	Read(A)	
	$A := f_4(A)$	
Read(C)		
	Write(A)	
$C := f_5(C)$		
Write(C)		
		$B := f_6(B)$
		Write(B)

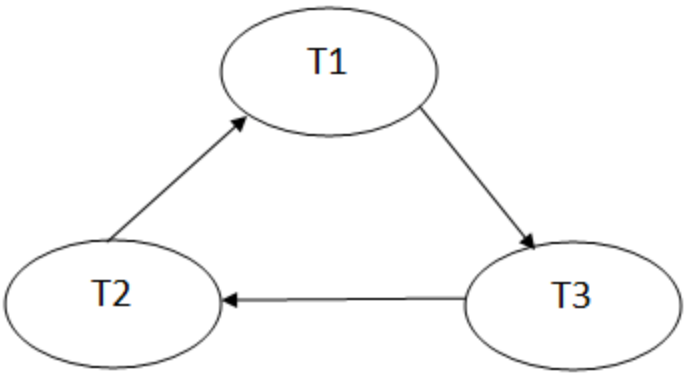
Schedule S1

Explanation:

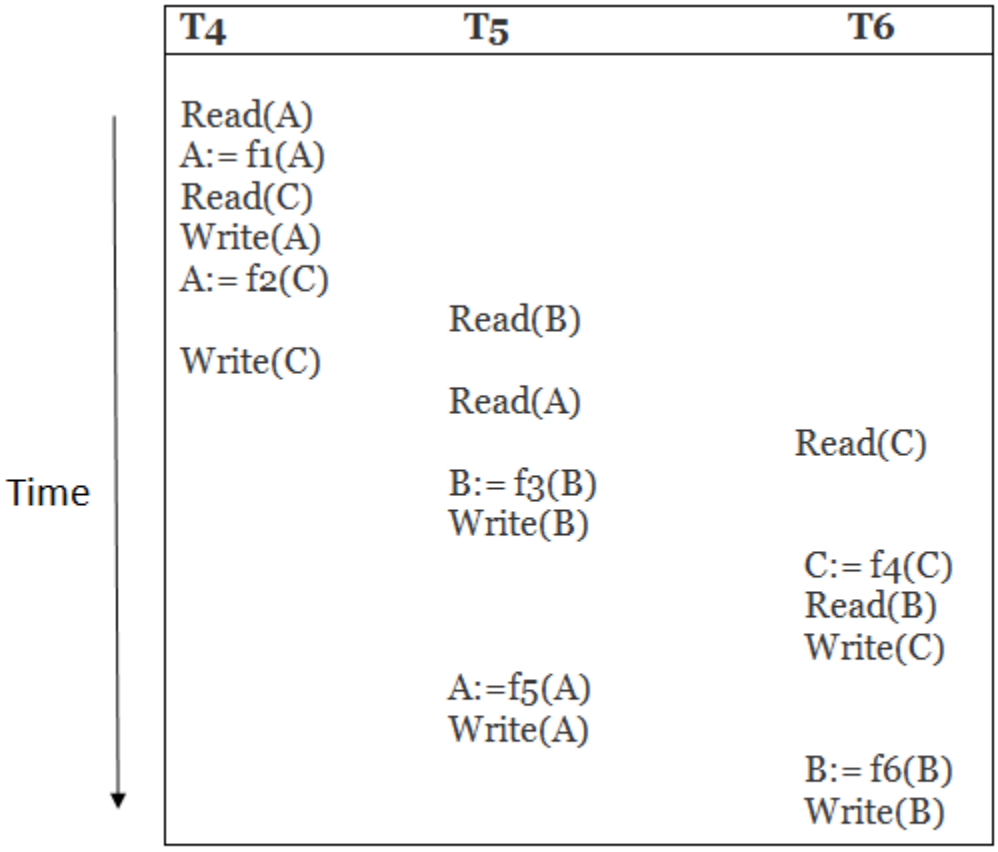
[redacted]

Read(A): In T1, no subsequent writes to A, so no new edges
Read(B): In T2, no subsequent writes to B, so no new edges
Read(C): In T3, no subsequent writes to C, so no new edges
Write(B): B is subsequently read by T3, so add edge T2 → T3
Write(C): C is subsequently read by T1, so add edge T3 → T1
Write(A): A is subsequently read by T2, so add edge T1 → T2
Write(A): In T2, no subsequent reads to A, so no new edges
Write(C): In T1, no subsequent reads to C, so no new edges
Write(B): In T3, no subsequent reads to B, so no new edges

Precedence graph for schedule S1:



The precedence graph for schedule S1 contains a cycle that's why Schedule S1 is non-serializable.



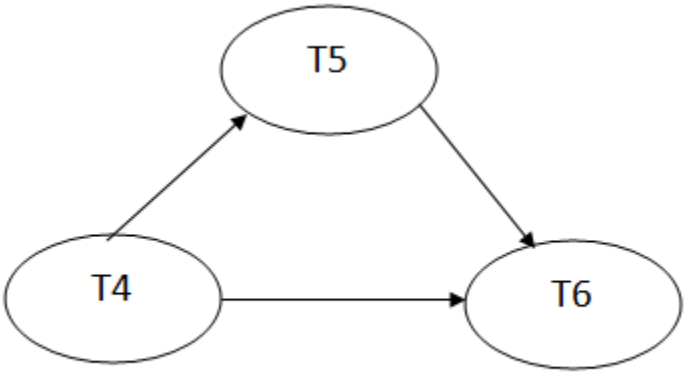
Schedule S2

Explanation:

Read(A): In T4,no subsequent writes to A, so no new edges
Read(C): In T4, no subsequent writes to C, so no new edges

Write(A): A is subsequently read by T5, so add edge T4 → T5
Read(B): In T5,no subsequent writes to B, so no new edges
Write(C): C is subsequently read by T6, so add edge T4 → T6
Write(B): A is subsequently read by T6, so add edge T5 → T6
Write(C): In T6, no subsequent reads to C, so no new edges
Write(A): In T5, no subsequent reads to A, so no new edges
Write(B): In T6, no subsequent reads to B, so no new edges

Precedence graph for schedule S2:



The precedence graph for schedule S2 contains no cycle that's why ScheduleS2 is serializable.

7.Lock-Based Protocol

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

1. Shared lock:

- **It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.**
- **It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.**

2. Exclusive lock:

- **In the exclusive lock, the data item can be both reads as well as written by the transaction.**

- **This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.**

There are four types of lock protocols available:

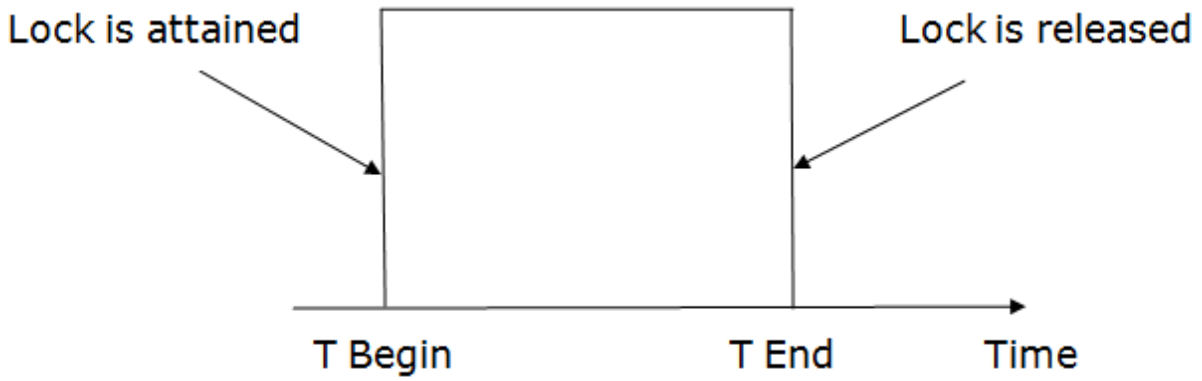
1. Simplistic lock protocol

It is the simplest way of locking the data while transaction. Simplistic lock-based protocols allow all the transactions to get the lock on the data before insert or delete or update on it. It will unlock the data item after completing the transaction.

2. Pre-claiming Lock Protocol

- **Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.**

- **Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.**
- **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.**

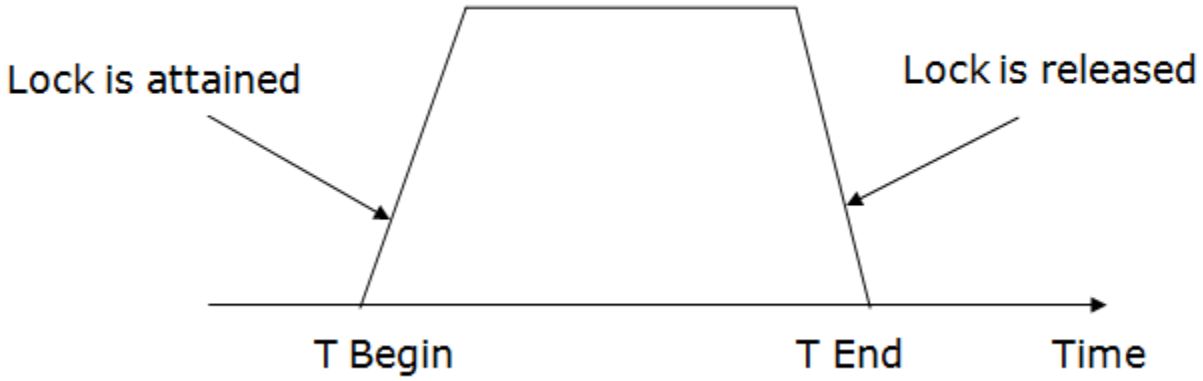


3. Two-phase locking (2PL)

- **The two-phase locking protocol divides the execution phase of the transaction into three parts.**

- **In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.**
- **In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.**

- **In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.**



There are two phases of 2PL:

Growing phase: In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

Shrinking phase: In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

In the below example, if lock conversion is allowed then the following phase can happen:

1. **Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.**
2. **Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.**

Example:

	T1	T2
0	LOCK-S(A)	
1		LOCK-S(A)
2	LOCK-X(B)	
3	——	——
4	UNLOCK(A)	
5		LOCK-X(C)
6	UNLOCK(B)	
7		UNLOCK(A)
8		UNLOCK(C)
9	——	——

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

Transaction T1:

- **Growing phase: from step 1-3**
- **Shrinking phase: from step 5-7**
- **Lock point: at 3**

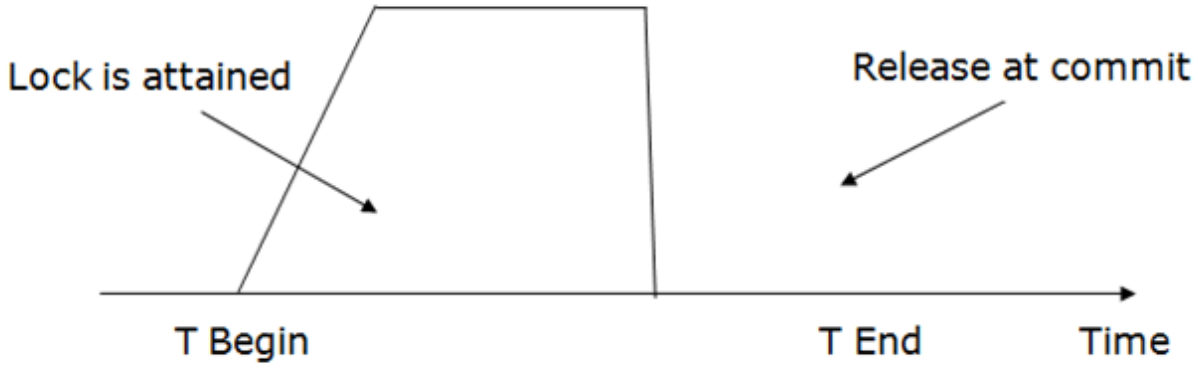
Transaction T2:

- **Growing phase: from step 2-6**
- **Shrinking phase: from step 8-9**
- **Lock point: at 6**

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

- **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.**



It does not have cascading abort as 2PL does.

8. Timestamp Based Protocols-
Concurrency Control can be implemented in different ways. One way to implement it is by using Locks. Now, let us discuss Time Stamp Ordering Protocol.

As earlier introduced, **Timestamp** is a unique identifier created by the DBMS to identify a transaction. They are usually assigned in the order in which they are submitted to the system. Refer to the timestamp of a transaction *T* as ***TS(T)***. For the basics of Timestamp, you may refer here.

Timestamp Ordering Protocol –
The main idea for this protocol is to order the transactions based on their Timestamps. A schedule in which the transactions participate is then serializable and the only *equivalent serial schedule permitted* has the transactions in the order of their Timestamp Values. Stating simply, the schedule is equivalent to the particular *Serial Order* corresponding to the *order of the Transaction timestamps*. An algorithm must ensure that, for each item accessed by *Conflicting Operations* in the schedule, the order in which the item is accessed does not violate the

ordering. To ensure this, use two Timestamp Values relating to each database item **X**.

- **W_TS(X)** is the largest timestamp of any transaction that executed **write(X)** successfully.

- **R_TS(X)** is the largest timestamp of any transaction that executed **read(X)** successfully.

Basic Timestamp Ordering –
Every transaction is issued a timestamp based on when it enters the system. Suppose, if an old transaction Ti has timestamp TS(Ti), a new transaction Tj is assigned timestamp TS(Tj) such that **TS(Ti) < TS(Tj)**. The protocol manages concurrent execution such that the timestamps determine the serializability order. The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. Whenever some Transaction *T* tries to issue a R_item(X) or a W_item(X), the Basic TO algorithm compares the timestamp of *T* with **R_TS(X) & W_TS(X)** to ensure that the Timestamp order is not violated. This describes the Basic TO protocol in the following two cases.

1. Whenever a Transaction *T* issues a **W_item(X)** operation, check the following conditions:

    - If ***R_TS(X) > TS(T)*** and if ***W_TS(X) > TS(T)***, then abort and rollback T and reject the operation. else,
    - Execute W_item(X) operation of T and set W_TS(X) to TS(T).
2. Whenever a Transaction *T* issues a **R_item(X)** operation, check the following conditions:
    - If ***W_TS(X) > TS(T)***, then abort and reject T and reject the operation, else
    - If W_TS(X) <= TS(T), then execute the R_item(X) operation of T and set R_TS(X) to the larger of TS(T) and current R_TS(X).

Whenever the Basic TO algorithm detects two conflicting operations that occur in an incorrect order, it rejects the latter of the two operations by aborting the Transaction that issued it. Schedules produced by Basic TO are guaranteed to be *conflict serializable*. Already discussed that using Timestamp can ensure that our schedule will be *deadlock free*.

One drawback of the Basic TO protocol is that **Cascading Rollback** is still possible. Suppose we have a Transaction T1 and T2 has used a value written by

T1. If T1 is aborted and resubmitted to the system then, T2 must also be aborted and rolled back. So the problem of Cascading aborts still prevails.

Let's gist the Advantages and Disadvantages of Basic TO protocol:

- Timestamp Ordering protocol ensures serializability since the precedence graph will be of the form:

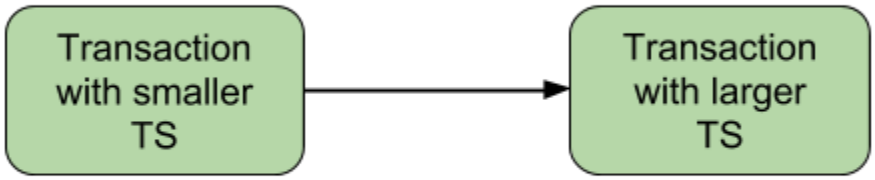


Image – Precedence Graph for TS ordering

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.

- But the schedule may *not be cascade free*, and may not even be recoverable.

Strict Timestamp Ordering –
A variation of Basic TO is called **Strict TO** ensures that the schedules are both Strict and Conflict Serializable. In this variation, a Transaction T that issues a R_item(X) or W_item(X) such that TS(T) > W_TS(X) has its read or write operation delayed until the Transaction **T'** that wrote the values of X has committed or aborted.

Advantages :

High Concurrency: Timestamp-based concurrency control allows for a high degree of concurrency by ensuring that transactions do not interfere with each other.

Efficient: The technique is efficient and scalable, as it does not require locking and can handle a large number of transactions.

No Deadlocks: Since there are no locks involved, there is no possibility of deadlocks occurring.

Improved Performance: By allowing transactions to execute concurrently, the overall performance of the database system can be improved.

Disadvantages:

Limited Granularity: The granularity of timestamp-based concurrency control is limited to the precision of the timestamp. This can lead to situations where transactions are unnecessarily blocked, even if they do not conflict with each other.

Timestamp Ordering: In order to ensure that transactions are executed in the correct order, the timestamps need to be carefully managed. If not managed properly, it can lead to inconsistencies in the database.

Timestamp Synchronization: Timestamp-based concurrency control requires that all transactions have synchronized clocks. If the clocks are not synchronized, it can lead to incorrect ordering of transactions.

Timestamp Allocation: Allocating unique timestamps for each transaction can be challenging, especially in distributed systems where transactions may be initiated at different locations.

8. Validation- Based Protocols

Validation Based Protocol is also called Optimistic Concurrency Control Technique. This protocol is used in DBMS (Database Management System) for avoiding concurrency in transactions. It is called optimistic because of the assumption it makes, i.e. very less interference occurs, therefore, there is no need for checking while the transaction is executed.

In this technique, no checking is done while the transaction is been executed. Until the transaction end is reached updates in the transaction are not applied directly to the database. All updates are applied to local copies of data items kept for the transaction. At the end of transaction execution, while execution of the transaction, a **validation phase** checks whether any of transaction updates violate serializability. If there is no violation of serializability the transaction is committed and the database is updated; or else, the transaction is updated and then restarted.

Optimistic Concurrency Control is a three-phase protocol. The three phases for validation based protocol:

1. **Read Phase:**
   Values of committed data items from the database can be read by a transaction. Updates are only applied to local data versions.

2. **Validation Phase:**
   Checking is performed to make sure that there is no violation of serializability when the transaction updates are applied to the database.

3. **Write Phase:**
   On the success of the validation phase, the transaction updates are applied to the database, otherwise, the updates are discarded and the transaction is slowed down.

10. Multiple Granularity:

Granularity: It is the size of data item allowed to lock.

- It can be defined as hierarchically breaking up the database into blocks which can be locked.
- The Multiple Granularity protocol enhances concurrency and reduces lock overhead.
- It maintains the track of what to lock and how to lock.
- It makes easy to decide either to lock a data item or to unlock a data item. This type of hierarchy can be graphically represented as a tree.

For example: Consider a tree which has four levels of nodes.

- The first level or higher level shows the entire database.
- The second level represents a node of type area. The higher level database consists of exactly these areas.
- The area consists of children nodes which are known as files. No file can be present in more than one area.
- Finally, each file contains child nodes known as records. The file has exactly those records that are its child nodes. No records represent in more than one file.
- Hence, the levels of the tree starting from the top level are as follows:

1. Database
2. Area
3. File
4. Record

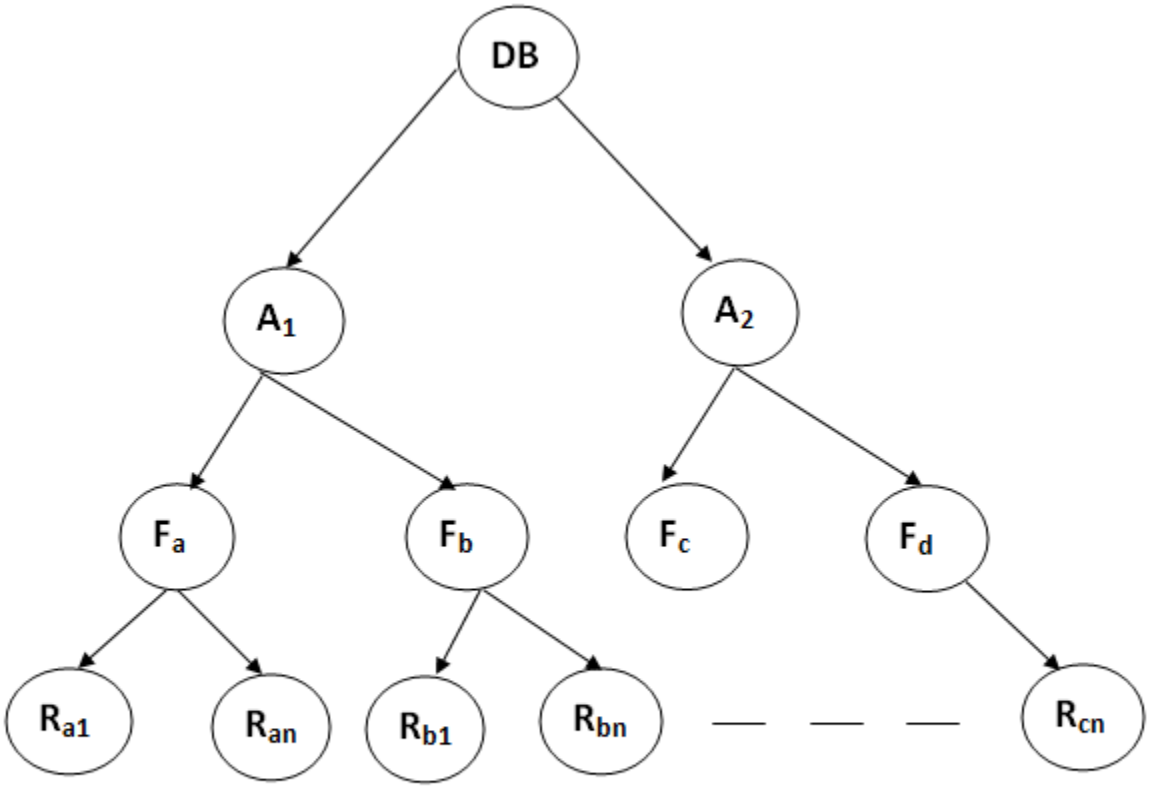


Figure: Multi Granularity tree Hierarchy

In this example, the highest level shows the entire database. The levels below are file, record, and fields.

[redacted]

There are three additional lock modes with multiple granularity:

Intention Mode Lock

Intention-shared (IS): It contains explicit locking at a lower level of the tree but only with shared locks.

Intention-Exclusive (IX): It contains explicit locking at a lower level with exclusive or shared locks.

Shared & Intention-Exclusive (SIX): In this lock, the node is locked in shared mode, and some node is locked in exclusive mode by the same transaction.

Compatibility Matrix with Intention Lock Modes: The below table describes the compatibility matrix for these lock modes:

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	X
IX	✓	✓	X	X	X
S	✓	X	✓	X	X
SIX	✓	X	X	X	X
X	X	X	X	X	X

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.

Observe that in multiple-granularity, the locks are acquired in top-down order, and locks must be released in bottom-up order.

- If transaction T1 reads record R_{a9} in file F_a, then transaction T1 needs to lock the database, area A_1 and file F_a in IX mode. Finally, it needs to lock R_{a2} in S mode.

- If transaction T2 modifies record R_{a9} in file F_a, then it can do so after locking the database, area A_1 and file F_a in IX mode. Finally, it needs to lock the R_{a9} in X mode.

- If transaction T3 reads all the records in file F_a, then transaction T3 needs to lock the database, and area A in IS mode. At last, it needs to lock F_a in S mode.
- If transaction T4 reads the entire database, then T4 needs to lock the database in S mode.

Recovery and Atomicity – Log – Based Recovery – Recovery with Concurrent Transactions

Data may be monitored, stored, and changed rapidly and effectively using a DBMS (Database Management System).A database possesses atomicity, consistency, isolation, and durability qualities. The ability of a system to preserve data and changes made to data defines its durability. A database could fail for any of the following reasons:

- System breakdowns occur as a result of hardware or software issues in the system.
- Transaction failures arise when a certain process dealing with data updates cannot be completed.
- Disk crashes may occur as a result of the system's failure to read the disc.
- Physical damages include issues such as power outages or natural disasters.
- The data in the database must be recoverable to the state they were in prior to the system failure, even if the database system fails. In such situations, database recovery procedures in DBMS are employed to retrieve the data.

The recovery procedures in DBMS ensure the database's atomicity and durability. If a system crashes in the middle of a transaction and all of its data is lost, it is not regarded as durable. If just a portion of the data is updated during the transaction, it is not considered atomic. Data recovery procedures in DBMS make sure that the data is always recoverable to protect the durability property and that its state is retained to protect the atomic property. The procedures listed below are used to recover data from a DBMS,

- Recovery based on logs.
- Recovery through Deferred Update
- Immediate Recovery via Immediate Update

The atomicity attribute of DBMS safeguards the data state. If a data modification is performed, the operation must be completed entirely, or the data's state must be maintained as if the manipulation never occurred. This characteristic may be impacted by DBMS failure brought on by transactions, but DBMS recovery methods will protect it.

What exactly is a Log-Based Recovery?

Every DBMS has its own system logs, which record every system activity and include timestamps for the event's timing. Databases manage several log files for operations such as errors, queries, and other database updates. The log is saved in the following file formats:

- The structure **[start transaction, T]** represents the start of transaction T execution.
- **[write the item, T, X, old value, new value]** indicates that the transaction T changes the value of the variable X from the old value to the new value.
- **[read item, T, X]** indicates that the transaction T reads the value of X.
- **[commit, T]** signifies that the modifications to the data have been committed to the database and cannot be updated further by the transaction. There will be no errors after the database has been committed.
- **[abort, T]** indicates that the transaction, T, has been cancelled.

We may utilize these logs to see how the state of the data changes during a transaction and recover it to the prior or new state.

An undo operation can be used to inspect the [write item, T, X, old value, new value] operation and restore the data state to old data. The only way to restore the previous state of data to the new state that was lost due to a system failure is to do the [commit, T] action.

Consider the following series of transactions: t1, t2, t3, and t4. The system crashes after the fourth transaction; however, the data can still be retrieved to the state it was in before the checkpoint was established during transaction t1.

After all of the records of a transaction are written to logs, a checkpoint is created to transfer all of the logs from local storage to permanent storage for future usage.

What is the Conceded Update Method?

Until the transaction enters its final stage or during the commit operation, the data is not changed using the conceded update mechanism. Following this procedure, the data is updated and permanently placed in the main memory.

In the event of a failure, the logs, which are preserved throughout the process, are utilized to identify the fault's precise moment. We benefit from this because even if the system crashes before the commit step, the database's data is not altered, and the status is managed. If the system fails after the commit stage, we may quickly redo the changes to the new stage, as opposed to the undo procedure.

11. Buffer Management :

Database Buffer

In our previous section, we learned about various types of data storage. But, the goal of a database system is that a minimum number of transfers should take place between the disk and memory. To do so, it can reduce the number of disk accesses by keeping as many blocks in main memory. So, when the user wants to store the data, it can directly search in the main memory, and there will be no requirement of accessing the disk. However, it is difficult to keep so many blocks in main memory; we need to manage the allocation of the space available in the main memory for the storage of blocks.

A database buffer is a temporary storage area in the main memory. It allows storing the data temporarily when moving from one place to another. A database buffer stores a copy of disk blocks. But, the version of block copies on the disk may be older than the version in the buffer.

What is Buffer Manager

- A Buffer Manager is responsible for allocating space to the buffer in order to store data into the buffer.

- If a user request a particular block and the block is available in the buffer, the buffer manager provides the block address in the main memory.
- If the block is not available in the buffer, the buffer manager allocates the block in the buffer.
- If free space is not available, it throws out some existing blocks from the buffer to allocate the required space for the new block.
- The blocks which are thrown are written back to the disk only if they are recently modified when writing on the disk.

- If the user requests such thrown-out blocks, the buffer manager reads the requested block from the disk to the buffer and then passes the address of the requested block to the user in the main memory.
- However, the internal actions of the buffer manager are not visible to the programs that may create any problem in disk-block requests. The buffer manager is just like a virtual machine.

For serving the database system in the best possible way, the buffer manager uses the following methods:

1. **Buffer Replacement Strategy:** If no space is left in the buffer, it is required to remove an existing block from the buffer before allocating the new one. The various operating system uses the LRU (least recently used) scheme. In LRU, the block that was least recently used is removed from the buffer and written back to the disk. Such type of replacement strategy is known as Buffer Replacement Strategy.

2. **Pinned Blocks:** If the user wants to recover any database system from the crashes, it is essential to restrict the time when a block is written back to the disk. In fact, most recovery systems do not allow the blocks to be written on the disk if the block updation being in progress. Such types of blocks that are not allowed to be written on the disk are known as **pinned blocks**. Luckily, many operating systems do not support the pinned blocks.
   - **Forced Output of Blocks:** In some cases, it becomes necessary to write the block back to the disk even though the space occupied by the block in the buffer is not required. When such type of write is required, it is known as the **forced output of a block**. It is because sometimes the data stored on the buffer may get lost in some system crashes, but the data stored on the disk usually does not get affected due to any disk crash.

12. Failure with loss of nonvolatile storage:
The basic measure is to dump the entire contents of the database to stable storage periodically.

One approach to dump the database requires that no transaction is active during the dumping procedure and uses a procedure similar to checkpointing.

a) Output all the log records currently present in the main memory into the stable storage.

b) Output all the buffer blocks into the disk.

c) Copy all the data present in the database to the stable storage.

d) Output a log record <dump> into the stable storage.

A dump of the database contents is also called the ‘archival dump’.

Most database systems support an ‘SQL dump’ as well, which writes out all the SQL DDL statements and SQL insert statements into a file, which can then be re-executed to recreate the database.

12. Advance Recovery systems

Database Systems like any other computer system, are subject to failures but the data stored in them must be available as and when required. When a database fails it must possess the facilities for fast recovery. It must also have atomicity i.e. either transactions are completed successfully and committed (the effect is recorded permanently in the database) or the transaction should have no effect on the database.

Types of Recovery Techniques in DBMS

Database recovery techniques are used in database management systems (DBMS) to restore a database to a consistent state after a failure or error has occurred. The main goal of recovery techniques is to ensure data integrity and consistency and prevent data loss.

There are mainly two types of recovery techniques used in DBMS

- **Rollback/Undo Recovery Technique**
- **Commit/Redo Recovery Technique**

Rollback/Undo Recovery Technique

The rollback/undo recovery technique is based on the principle of backing out or undoing the effects of a transaction that has not been completed successfully due to a system failure or error. This technique is accomplished by undoing the changes made by the transaction using the log records stored in the transaction log. The transaction log contains a record of all the transactions that have been performed on the database. The system uses the log records to undo the changes made by the failed transaction and restore the database to its previous state.

Commit/Redo Recovery Technique

The commit/redo recovery technique is based on the principle of reapplying the changes made by a transaction that has been completed successfully to the database. This technique is accomplished by using the log records stored in the transaction log to redo the changes made by the transaction that was in progress at the time of the failure or error. The system uses the log records to reapply the

changes made by the transaction and restore the database to its most recent consistent state.

In addition to these two techniques, there is also a third technique called checkpoint recovery.

Checkpoint Recovery is a technique used to reduce the recovery time by periodically saving the state of the database in a checkpoint file. In the event of a failure, the system can use the checkpoint file to restore the database to the most recent consistent state before the failure occurred, rather than going through the entire log to recover the database.

Remote backup systems provide a wide range of availability, allowing the transaction processing to continue even if the primary site is destroyed by a fire, flood or earthquake.

Data and log records from a primary site are continuously backed up into a remote backup site.

One can achieve 'wide range availability' of data by performing transaction processing at one site, called the '**primary site**', and having a '**remote backup**' site where all the data from the primary site are duplicated.

The remote site is also called '**secondary site**'.

The remote site must be **synchronized** with the primary site, as updates are performed at the primary.

In designing a remote backup system, the following points are important.

a) Detection of failure: It is important for the remote backup system to detect when the primary has failed.

b) Transfer of control: When the primary site fails, the backup site takes over the processing and becomes the new primary site.

c) Time to recover: If the log at the remote backup becomes large, recovery will take a long time.

d) Time to commit: To ensure that the updates of a committed transaction are durable, a transaction should not be announced committed until its log records have reached the backup site.