**UNIT-III**

**What is a Database Transaction?**

A **Database Transaction** is a logical unit of processing in a DBMS which entails one or more database access operation. In a nutshell, database transactions represent real-world events of any enterprise.

All types of database access operation which are held between the beginning and end transaction statements are considered as a single logical transaction in DBMS. During the transaction the database is inconsistent. Only once the database is committed the state is changed from one consistent state to another.

## States of Transactions

The various states of a transaction concept in DBMS are listed below:

|  |  |
| --- | --- |
| **State** | **Transaction types** |
| Active State | A transaction enters into an active state when the execution process begins. During this state read or write operations can be performed. |
| Partially Committed | A transaction goes into the partially committed state after the end of a transaction. |
| Committed State | When the transaction is committed to state, it has already completed its execution successfully. Moreover, all of its changes are recorded to the database permanently. |
| Failed State | A transaction considers failed when any one of the checks fails or if the transaction is aborted while it is in the active state. |
| Terminated State | State of transaction reaches terminated state when certain transactions which are leaving the system can’t be restarted. |

State Transition Diagram for a Database Transaction

Let’s study a [state transition diagram](https://www.guru99.com/state-transition-testing.html) that highlights how a transaction moves between these various states.

1. Once a transaction states execution, it becomes active. It can issue READ or WRITE operation.
2. Once the READ and WRITE operations complete, the transactions becomes partially committed state.
3. Next, some recovery protocols need to ensure that a system failure will not result in an inability to record changes in the transaction permanently. If this check is a success, the transaction commits and enters into the committed state.
4. If the check is a fail, the transaction goes to the Failed state.
5. If the transaction is aborted while it’s in the active state, it goes to the failed state. The transaction should be rolled back to undo the effect of its write operations on the database.
6. The terminated state refers to the transaction leaving the system.

# ACID Properties in DBMS

DBMS is the management of data that should remain integrated when any changes are done in it. It is because if the integrity of the data is affected, whole data will get disturbed and corrupted. Therefore, to maintain the integrity of the data, there are four properties described in the database management system, which are known as the **ACID** properties. The ACID properties are meant for the transaction that goes through a different group of tasks, and there we come to see the role of the ACID properties.

In this section, we will learn and understand about the ACID properties. We will learn what these properties stand for and what does each property is used for. We will also understand the ACID properties with the help of some examples.

## ACID Properties

The expansion of the term ACID defines for:



**1) Atomicity:** The term atomicity defines that the data remains atomic. It means if any operation is performed on the data, either it should be performed or executed completely or should not be executed at all. It further means that the operation should not break in between or execute partially. In the case of executing operations on the transaction, the operation should be completely executed and not partially.

**Example:** If Remo has account A having $30 in his account from which he wishes to send $10 to Sheero's account, which is B. In account B, a sum of $ 100 is already present. When $10 will be transferred to account B, the sum will become $110. Now, there will be two operations that will take place. One is the amount of $10 that Remo wants to transfer will be debited from his account A, and the same amount will get credited to account B, i.e., into Sheero's account. Now, what happens - the first operation of debit executes successfully, but the credit operation, however, fails. Thus, in Remo's account A, the value becomes $20, and to that of Sheero's account, it remains $100 as it was previously present.



In the above diagram, it can be seen that after crediting $10, the amount is still $100 in account B. So, it is not an atomic transaction.

The below image shows that both debit and credit operations are done successfully. Thus the transaction is atomic.



Thus, when the amount loses atomicity, then in the bank systems, this becomes a huge issue, and so the atomicity is the main focus in the bank systems.

**2) Consistency:** The word **consistency** means that the value should remain preserved always. In [DBMS](https://www.javatpoint.com/dbms-tutorial), the integrity of the data should be maintained, which means if a change in the database is made, it should remain preserved always. In the case of transactions, the integrity of the data is very essential so that the database remains consistent before and after the transaction. The data should always be correct.

**Example:**



In the above figure, there are three accounts, A, B, and C, where A is making a transaction T one by one to both B & C. There are two operations that take place, i.e., Debit and Credit. Account A firstly debits $50 to account B, and the amount in account A is read $300 by B before the transaction. After the successful transaction T, the available amount in B becomes $150. Now, A debits $20 to account C, and that time, the value read by C is $250 (that is correct as a debit of $50 has been successfully done to B). The debit and credit operation from account A to C has been done successfully. We can see that the transaction is done successfully, and the value is also read correctly. Thus, the data is consistent. In case the value read by B and C is $300, which means that data is inconsistent because when the debit operation executes, it will not be consistent.

**4) Isolation:** The term 'isolation' means separation. In DBMS, Isolation is the property of a database where no data should affect the other one and may occur concurrently. In short, the operation on one database should begin when the operation on the first database gets complete. It means if two operations are being performed on two different databases, they may not affect the value of one another. In the case of transactions, when two or more transactions occur simultaneously, the consistency should remain maintained. Any changes that occur in any particular transaction will not be seen by other transactions until the change is not committed in the memory.

**Example:** If two operations are concurrently running on two different accounts, then the value of both accounts should not get affected. The value should remain persistent. As you can see in the below diagram, account A is making T1 and T2 transactions to account B and C, but both are executing independently without affecting each other. It is known as Isolation.



**4) Durability:** Durability ensures the permanency of something. In DBMS, the term durability ensures that the data after the successful execution of the operation becomes permanent in the database. The durability of the data should be so perfect that even if the system fails or leads to a crash, the database still survives. However, if gets lost, it becomes the responsibility of the recovery manager for ensuring the durability of the database. For committing the values, the COMMIT command must be used every time we make changes.

Therefore, the ACID property of DBMS plays a vital role in maintaining the consistency and availability of data in the database.

Thus, it was a precise introduction of ACID properties in DBMS. We have discussed these properties in the transaction section also.

**Seriability:**

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.

In case of transaction the term ACID has been used significantly to state some of important properties that a transaction must follow. We all know ACID stands for Atomicity, Consistency, Isolation and Durability and these properties collectively called as ACID Properties.

## Isolation

It determines the visibility of transactions of other systems. A lower level allows every user to access the same data. Therefore, it involves high risk of data privacy and security of the system. However, a higher isolation level reduces the type of concurrency over the data but requires more resources and is slower than lower isolation levels.

The isolation protocols help safeguards the data from unwanted transactions. They maintain the integrity of every data by defining how and when the changes made by one operation are visible to others.

## Levels of isolation

## Isolation

It determines the visibility of transactions of other systems. A lower level allows every user to access the same data. Therefore, it involves high risk of data privacy and security of the system. However, a higher isolation level reduces the type of concurrency over the data but requires more resources and is slower than lower isolation levels.

The isolation protocols help safeguards the data from unwanted transactions. They maintain the integrity of every data by defining how and when the changes made by one operation are visible to others.

There are four levels of isolations which are explained below −

* **Read Uncommitted** − It is the lowest level of isolation. At this level; the dirty reads are allowed, which means one can read the uncommitted changes made by another.
* **Read committed** − It allows no dirty reads, and clearly states that any uncommitted data is committed now it is read.
* **Repeatable Read** − This is the most restricted level of isolation. The transaction holds read locks on all the rows it references and write locks over all the rows it updates/inserts/deletes. So, there is no chance of non-repeatable reads.
* **Serializable** − The highest level of civilization. It determines that all concurrent transactions be executed serially.

## Example

Consider an example of isolation.

What is the isolation level of transaction E?

session begins

SET GLOBAL TRANSACTION

ISOLATION LEVEL SERIALIZABLE;

session ends

session begins

SET SESSION TRANSACTION

ISOLATION LEVEL REPEATABLE READ;

transaction A

transaction B

SET TRANSACTION

ISOLATION LEVEL READ UNCOMMITTED;

transaction C

SET TRANSACTION

ISOLATION LEVEL READ COMMITTED;

transaction D

transaction E

session ends

Check which option −

A- Serializable

B- Repeatable read

C- Read uncommitted

* **Step 1** − In the above program, the first session starts and ends without doing any transaction.
* **Step 2** − The second session begins at session-level with isolation level "Repeatable Read". Transaction A & B gets executed with these settings.
* **Step 3** − Once again a new transaction begins with isolation level "Read uncommitted". This setting is used only for "Transaction C" since "Set transaction" alone is mentioned. If the "SET transaction" is used without global or session keywords, then these particular settings will work only for a single transaction.
* **Step 4** − Once again "Set Transaction" with isolation level Read committed works only for Transaction D. (Refer step 3 for reason)
* **Step 5** − "Transaction E" gets continued at the "Repeatable Read" since the transaction started at step 2 has not ended still. Transaction isolation level set at Step 3 and Step 4 vanishes once a single transaction is executed. So, automatically "Transaction E" will refer to the prior transaction settings.

Concurrency control concept comes under the Transaction in database management system (DBMS). It is a procedure in DBMS which helps us for the management of two simultaneous processes to execute without conflicts between each other, these conflicts occur in multi user systems.

Concurrency can simply be said to be executing multiple transactions at a time. It is required to increase time efficiency. If many transactions try to access the same data, then inconsistency arises. Concurrency control required to maintain consistency data.

For example, if we take ATM machines and do not use concurrency, multiple persons cannot draw money at a time in different places. This is where we need concurrency.

## Advantages

The advantages of concurrency control are as follows −

* Waiting time will be decreased.
* Response time will decrease.
* Resource utilization will increase.
* System performance & Efficiency is increased.

# DBMS Concurrency Control

Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on a database.

But before knowing about concurrency control, we should know about concurrent execution.

## Concurrent Execution in DBMS

* In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.
* While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
* The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database. Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

### Concurrency Control Protocols

The concurrency control protocols ensure the atomicity, consistency, isolation, durability and serializability of the concurrent execution of the database transactions. Therefore, these protocols are categorized as:

* Lock Based Concurrency Control Protocol
* Time Stamp Concurrency Control Protocol
* Validation Based Concurrency Control Protocol

NEED FOR THE CONCURRENCY:

If transactions are executed serially, i.e., sequentially with no overlap in time, no transaction concurrency exists. However, if concurrent transactions with interleaving operations are allowed in an uncontrolled manner, some unexpected, undesirable result may occur, such as:

* The lost update problem: A second transaction writes a second value of a data-item (datum) on top of a first value written by a first concurrent transaction, and the first value is lost to other transactions running concurrently which need, by their precedence, to read the first value. The transactions that have read the wrong value end with incorrect results.
* The dirty read problem: Transactions read a value written by a transaction that has been later aborted. This value disappears from the database upon abort, and should not have been read by any transaction (“dirty read”). The reading transactions end with incorrect results.
* The incorrect summary problem: While one transaction takes a summary over the values of all the instances of a repeated data-item, a second transaction updates some instances of that data-item. The resulting summary does not reflect a correct result for any (usually needed for correctness) precedence order between the two transactions (if one is executed before the other), but rather some random result, depending on the timing of the updates, and whether certain update results have been included in the summary or not.

Most high-performance transactional systems need to run transactions concurrently to meet their performance requirements. Thus, without concurrency control such systems can neither provide correct results nor maintain their databases consistently.

# 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:**



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.

## What are Deadlocks?

Deadlock is a state of a database system having two or more transactions, when each transaction is waiting for a data item that is being locked by some other transaction. A deadlock can be indicated by a cycle in the wait-for-graph. This is a directed graph in which the vertices denote transactions and the edges denote waits for data items.

For example, in the following wait-for-graph, transaction T1 is waiting for data item X which is locked by T3. T3 is waiting for Y which is locked by T2 and T2 is waiting for Z which is locked by T1. Hence, a waiting cycle is formed, and none of the transactions can proceed executing.



## Deadlock Handling in Centralized Systems

There are three classical approaches for deadlock handling, namely −

* Deadlock prevention.
* Deadlock avoidance.
* Deadlock detection and removal.

All of the three approaches can be incorporated in both a centralized and a distributed database system.

### Deadlock Prevention

The deadlock prevention approach does not allow any transaction to acquire locks that will lead to deadlocks. The convention is that when more than one transactions request for locking the same data item, only one of them is granted the lock.

One of the most popular deadlock prevention methods is pre-acquisition of all the locks. In this method, a transaction acquires all the locks before starting to execute and retains the locks for the entire duration of transaction. If another transaction needs any of the already acquired locks, it has to wait until all the locks it needs are available. Using this approach, the system is prevented from being deadlocked since none of the waiting transactions are holding any lock.

### Deadlock Avoidance

The deadlock avoidance approach handles deadlocks before they occur. It analyzes the transactions and the locks to determine whether or not waiting leads to a deadlock.

The method can be briefly stated as follows. Transactions start executing and request data items that they need to lock. The lock manager checks whether the lock is available. If it is available, the lock manager allocates the data item and the transaction acquires the lock. However, if the item is locked by some other transaction in incompatible mode, the lock manager runs an algorithm to test whether keeping the transaction in waiting state will cause a deadlock or not. Accordingly, the algorithm decides whether the transaction can wait or one of the transactions should be aborted.

There are two algorithms for this purpose, namely **wait-die** and **wound-wait**. Let us assume that there are two transactions, T1 and T2, where T1 tries to lock a data item which is already locked by T2. The algorithms are as follows −

* **Wait-Die** − If T1 is older than T2, T1 is allowed to wait. Otherwise, if T1 is younger than T2, T1 is aborted and later restarted.
* **Wound-Wait** − If T1 is older than T2, T2 is aborted and later restarted. Otherwise, if T1 is younger than T2, T1 is allowed to wait.

### Deadlock Detection and Removal

The deadlock detection and removal approach runs a deadlock detection algorithm periodically and removes deadlock in case there is one. It does not check for deadlock when a transaction places a request for a lock. When a transaction requests a lock, the lock manager checks whether it is available. If it is available, the transaction is allowed to lock the data item; otherwise the transaction is allowed to wait.

Since there are no precautions while granting lock requests, some of the transactions may be deadlocked. To detect deadlocks, the lock manager periodically checks if the wait-forgraph has cycles. If the system is deadlocked, the lock manager chooses a victim transaction from each cycle. The victim is aborted and rolled back; and then restarted later. Some of the methods used for victim selection are −

* Choose the youngest transaction.
* Choose the transaction with fewest data items.
* Choose the transaction that has performed least number of updates.
* Choose the transaction having least restart overhead.
* Choose the transaction which is common to two or more cycles.

This approach is primarily suited for systems having transactions low and where fast response to lock requests is needed.

## Deadlock Handling in Distributed Systems

Transaction processing in a distributed database system is also distributed, i.e. the same transaction may be processing at more than one site. The two main deadlock handling concerns in a distributed database system that are not present in a centralized system are **transaction location** and **transaction control**. Once these concerns are addressed, deadlocks are handled through any of deadlock prevention, deadlock avoidance or deadlock detection and removal.

### Transaction Location

Transactions in a distributed database system are processed in multiple sites and use data items in multiple sites. The amount of data processing is not uniformly distributed among these sites. The time period of processing also varies. Thus the same transaction may be active at some sites and inactive at others. When two conflicting transactions are located in a site, it may happen that one of them is in inactive state. This condition does not arise in a centralized system. This concern is called transaction location issue.

This concern may be addressed by Daisy Chain model. In this model, a transaction carries certain details when it moves from one site to another. Some of the details are the list of tables required, the list of sites required, the list of visited tables and sites, the list of tables and sites that are yet to be visited and the list of acquired locks with types. After a transaction terminates by either commit or abort, the information should be sent to all the concerned sites.

### Transaction Control

Transaction control is concerned with designating and controlling the sites required for processing a transaction in a distributed database system. There are many options regarding the choice of where to process the transaction and how to designate the center of control, like −

* One server may be selected as the center of control.
* The center of control may travel from one server to another.
* The responsibility of controlling may be shared by a number of servers.

### Distributed Deadlock Prevention

Just like in centralized deadlock prevention, in distributed deadlock prevention approach, a transaction should acquire all the locks before starting to execute. This prevents deadlocks.

The site where the transaction enters is designated as the controlling site. The controlling site sends messages to the sites where the data items are located to lock the items. Then it waits for confirmation. When all the sites have confirmed that they have locked the data items, transaction starts. If any site or communication link fails, the transaction has to wait until they have been repaired.

Though the implementation is simple, this approach has some drawbacks −

* Pre-acquisition of locks requires a long time for communication delays. This increases the time required for transaction.
* In case of site or link failure, a transaction has to wait for a long time so that the sites recover. Meanwhile, in the running sites, the items are locked. This may prevent other transactions from executing.
* If the controlling site fails, it cannot communicate with the other sites. These sites continue to keep the locked data items in their locked state, thus resulting in blocking.

### Distributed Deadlock Avoidance

As in centralized system, distributed deadlock avoidance handles deadlock prior to occurrence. Additionally, in distributed systems, transaction location and transaction control issues needs to be addressed. Due to the distributed nature of the transaction, the following conflicts may occur −

* Conflict between two transactions in the same site.
* Conflict between two transactions in different sites.

In case of conflict, one of the transactions may be aborted or allowed to wait as per distributed wait-die or distributed wound-wait algorithms.

Let us assume that there are two transactions, T1 and T2. T1 arrives at Site P and tries to lock a data item which is already locked by T2 at that site. Hence, there is a conflict at Site P. The algorithms are as follows −

* **Distributed Wound-Die**
	+ If T1 is older than T2, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has either committed or aborted successfully at all sites.
	+ If T1 is younger than T2, T1 is aborted. The concurrency control at Site P sends a message to all sites where T1 has visited to abort T1. The controlling site notifies the user when T1 has been successfully aborted in all the sites.
* **Distributed Wait-Wait**
	+ If T1 is older than T2, T2 needs to be aborted. If T2 is active at Site P, Site P aborts and rolls back T2 and then broadcasts this message to other relevant sites. If T2 has left Site P but is active at Site Q, Site P broadcasts that T2 has been aborted; Site L then aborts and rolls back T2 and sends this message to all sites.
	+ If T1 is younger than T1, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has completed processing.

### Distributed Deadlock Detection

Just like centralized deadlock detection approach, deadlocks are allowed to occur and are removed if detected. The system does not perform any checks when a transaction places a lock request. For implementation, global wait-for-graphs are created. Existence of a cycle in the global wait-for-graph indicates deadlocks. However, it is difficult to spot deadlocks since transaction waits for resources across the network.

Alternatively, deadlock detection algorithms can use timers. Each transaction is associated with a timer which is set to a time period in which a transaction is expected to finish. If a transaction does not finish within this time period, the timer goes off, indicating a possible deadlock.

Another tool used for deadlock handling is a deadlock detector. In a centralized system, there is one deadlock detector. In a distributed system, there can be more than one deadlock detectors. A deadlock detector can find deadlocks for the sites under its control. There are three alternatives for deadlock detection in a distributed system, namely.

* **Centralized Deadlock Detector** − One site is designated as the central deadlock detector.
* **Hierarchical Deadlock Detector** − A number of deadlock detectors are arranged in hierarchy.
* **Distributed Deadlock Detector** − All the sites participate in detecting deadlocks and removing them.

[**Previous Page**](https://www.tutorialspoint.com/distributed_dbms/distributed_dbms_controlling_concurrency.htm)

The entire DBMS is a very complex structure with multiple transactions being performed and carried out every second. The toughness and strength of a system depend not only on the complex and secured architecture of a system but also in the way how data are managed and maintained in the worst cases. If the underlying architecture fails or crashes, then there must be some techniques and procedures by which the lost data during a transaction gets recovered.

## What is Data Recovery?

It is the method of restoring the database to its correct state in the event of a failure at the time of the transaction or after the end of a process. Earlier, you have been given the concept of database recovery as a service that should be provided by all the DBMS for ensuring that the database is dependable and remains in a consistent state in the presence of failures. In this context, dependability refers to both the flexibility of the DBMS to various kinds of failure and its ability to recover from those failures. In this chapter, you will gather a brief knowledge of how this service can be provided. To gain a better understanding of the possible problems you may encounter in providing a consistent system, you will first learn about the need for recovery and its types of failure, which usually occurs in a database environment.

### What is the Need for Recovery of data?

The storage of data usually includes four types of media with an increasing amount of reliability: the main memory, the magnetic disk, the magnetic tape, and the optical disk. Many different forms of failure can affect database processing and/or transaction, and each of them has to be dealt with differently. Some data failures can affect the main memory only, while others involve non-volatile or secondary storage also. Among the sources of failure are:

* Due to hardware or software errors, the system crashes, which ultimately resulting in loss of main memory.
* Failures of media, such as head crashes or unreadable media that results in the loss of portions of secondary storage.
* There can be application software errors, such as logical errors that are accessing the database that can cause one or more transactions to abort or fail.
* Natural physical disasters can also occur, such as fires, floods, earthquakes, or power failures.
* Carelessness or unintentional destruction of data or directories by operators or users.
* Damage or intentional corruption or hampering of data (using malicious software or files) hardware or software facilities.

Whatever the grounds of the failure are, there are two principal things that you have to consider:

* Failure of main memory, including that database buffers.
* Failure of the disk copy of that database.

### Recovery Facilities

Every DBMS should offer the following facilities to help out with the recovery mechanism:

* **Backup mechanism** makes backup copies at a specific interval for the database.
* **Logging facilities** keep tracing the current state of transactions and any changes made to the database.
* **Checkpoint facility** allows updates to the database for getting the latest patches to be made permanent and keep secure from vulnerability.
* **Recovery manager** allows the database system for restoring the database to a reliable and steady-state after any failure occurs.

# Failure Classification

To find that where the problem has occurred, we generalize a failure into the following categories:

1. Transaction failure
2. System crash
3. Disk failure

### 1. Transaction failure

The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transaction or process is hurt, then this is called as transaction failure.

Reasons for a transaction failure could be -

* 1. **Logical errors:** If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.
	2. **Syntax error:** It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. **For example,** The system aborts an active transaction, in case of deadlock or resource unavailability.

### 2. System Crash

* 1. System failure can occur due to power failure or other hardware or software failure. **Example:** Operating system error.

**Fail-stop assumption:** In the system crash, non-volatile storage is assumed not to be corrupted.

### 3. Disk Failure

* 1. It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.
	2. Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.