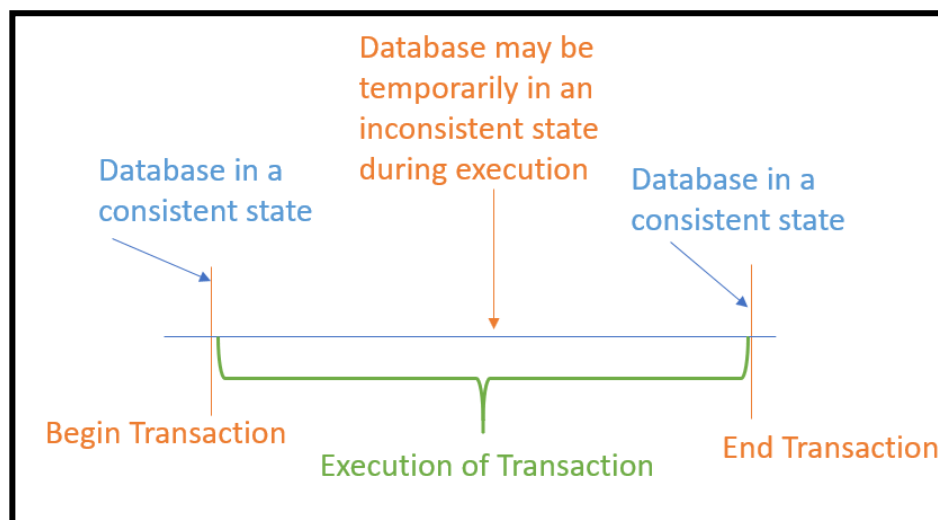


**Syllabus:** : Transaction State, Implementation of Atomicity and Durability, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for Serializability, Failure Classification, Storage, Recovery and Atomicity, Recovery algorithm. **Indexing Techniques:** B+ Trees: Search, Insert, Delete algorithms, File Organization and Indexing, Cluster Indexes, Primary and Secondary Indexes , Index data Structures, Hash Based Indexing: Tree base Indexing, Comparison of File Organizations, Indexes and Performance Tuning.

## Transaction

A database transaction is a set of logically related operation. It contains a group of tasks. A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.

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.



**Example:** Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:

### X's Account

1. Open\_Account(X)
2. Old\_Balance = X.balance
3. New\_Balance = Old\_Balance - 800
4. X.balance = New\_Balance
5. Close\_Account(X)

### Y's Account

1. Open\_Account(Y)
2. Old\_Balance = Y.balance
3. New\_Balance = Old\_Balance + 800
4. Y.balance = New\_Balance
5. Close\_Account(Y)

Operations of Transaction:

Following are the main operations of transaction:

**Read(X):** Read operation is used to read the value of X from the database and stores it in a buffer in main memory.

**Write(X):** Write operation is used to write the value back to the database from the buffer.

Let's take an example to debit transaction from an account which consists of following operations:

1. R(X);
2.  $X = X - 500$ ;
3. W(X);

Let's assume the value of X before starting of the transaction is 4000.

- The first operation reads X's value from database and stores it in a buffer.
- The second operation will decrease the value of X by 500. So, buffer will contain 3500.
- The third operation will write the buffer's value to the database. So, x's final value will be 3500.

But it may be possible that because of the failure of hardware, software or power, etc. that transaction may fail before finished all the operations in the set.

**For example:** If in the above transaction, the debit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the bank.

To solve this problem, we have two important operations:

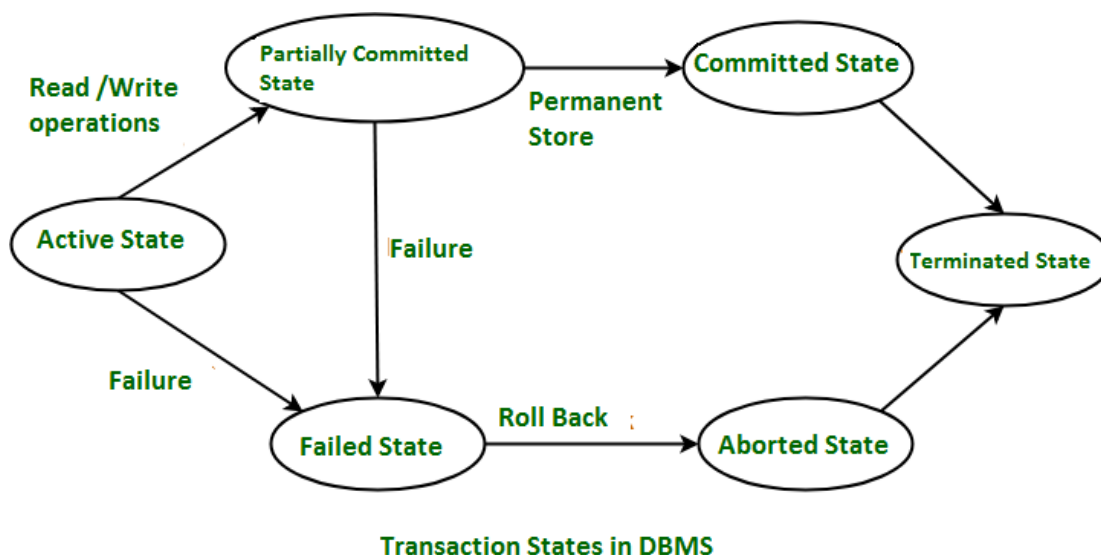
**Commit:** It is used to save the work done permanently.

**Rollback:** It is used to undo the work done.

## Transaction State

A transaction passes through many different states in its life cycle. These states are known as transaction states. A transaction can be in one of the following states in the database:

1. Active state
2. Partially committed state
3. Committed state
4. Failed state
5. Terminated state



### 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".

### Partially Committed

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

## 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.

## 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.

## Committed State

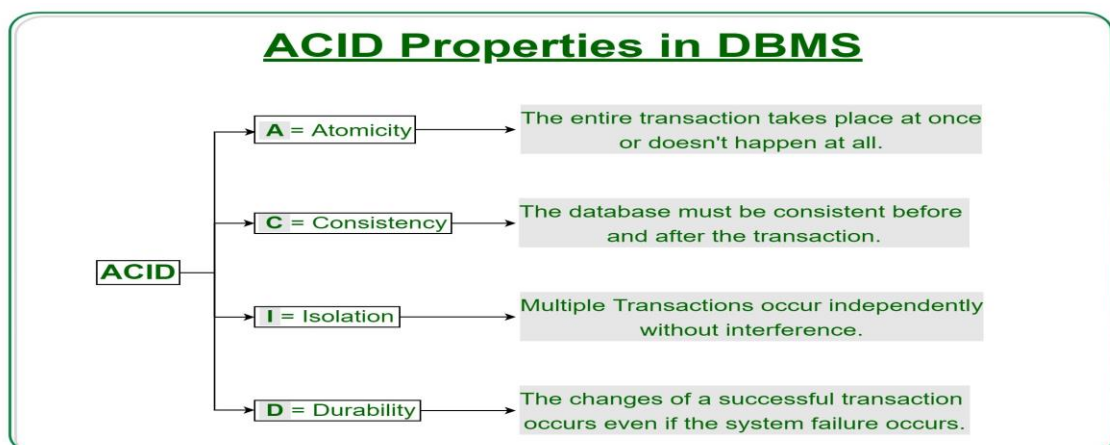
It is the state when the changes are made permanent on the Data Base and the transaction is complete and therefore terminated in the “terminated state”.

## 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.

## 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.

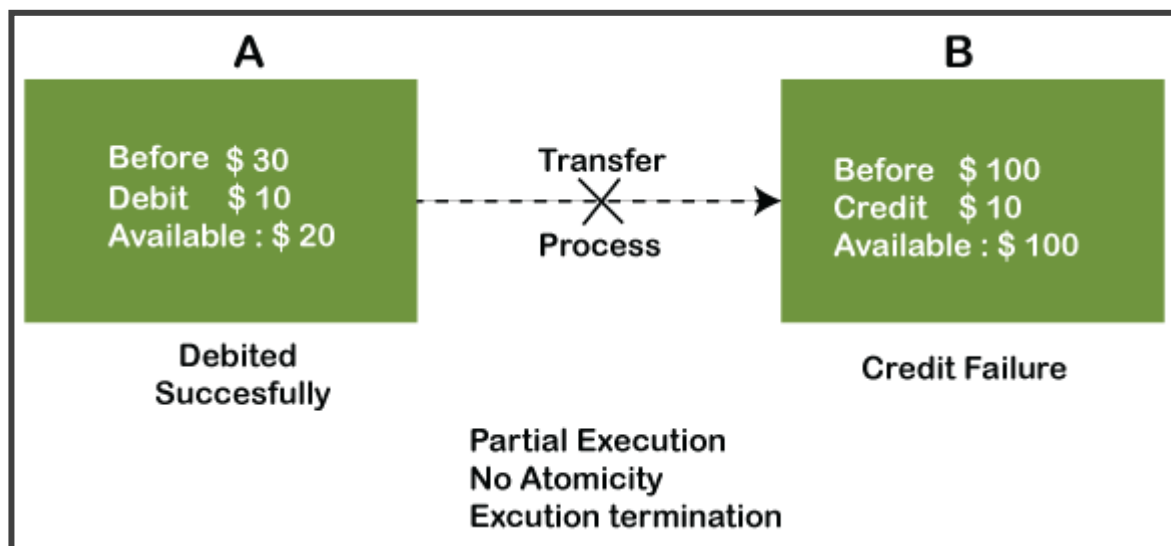


## Atomicity:

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.

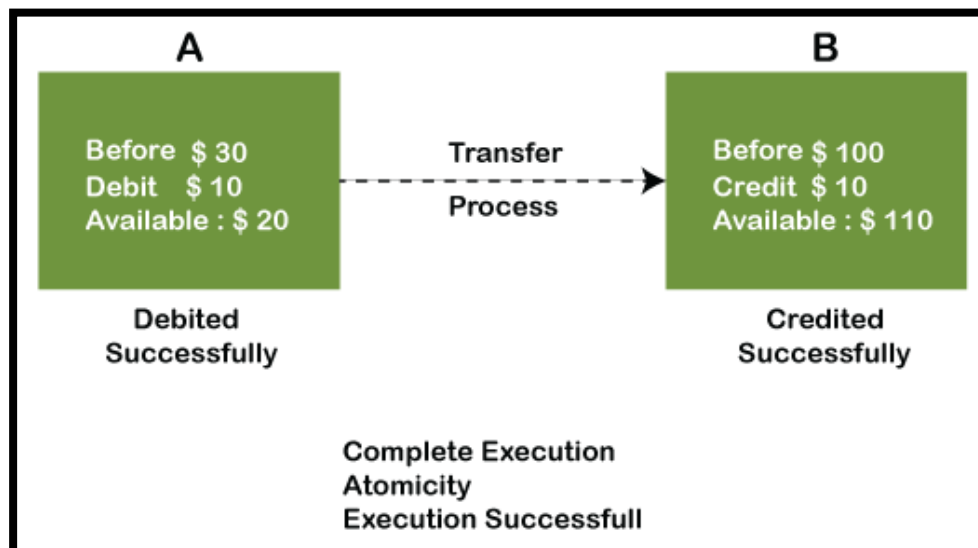
## Example:

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.

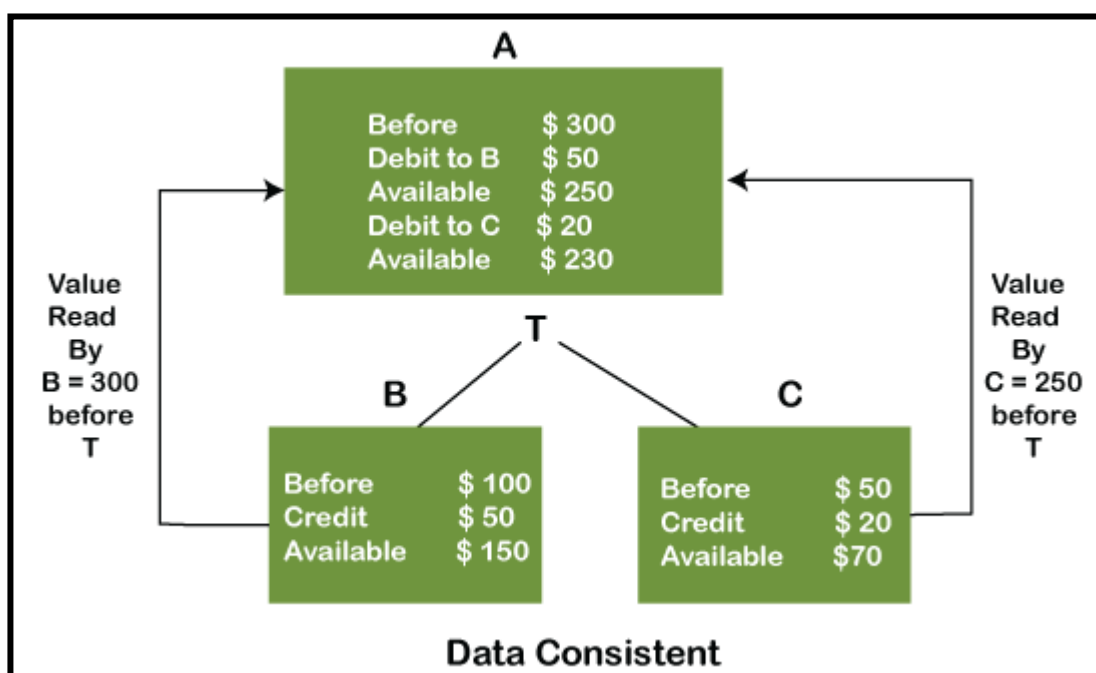


Thus, when the amount loses atomicity, then in the bank systems, this becomes a huge issue, and so the atomicity is the main focus in the bank systems.

### Consistency:

The word **consistency** means that the value should remain preserved always. In DBMS, 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:



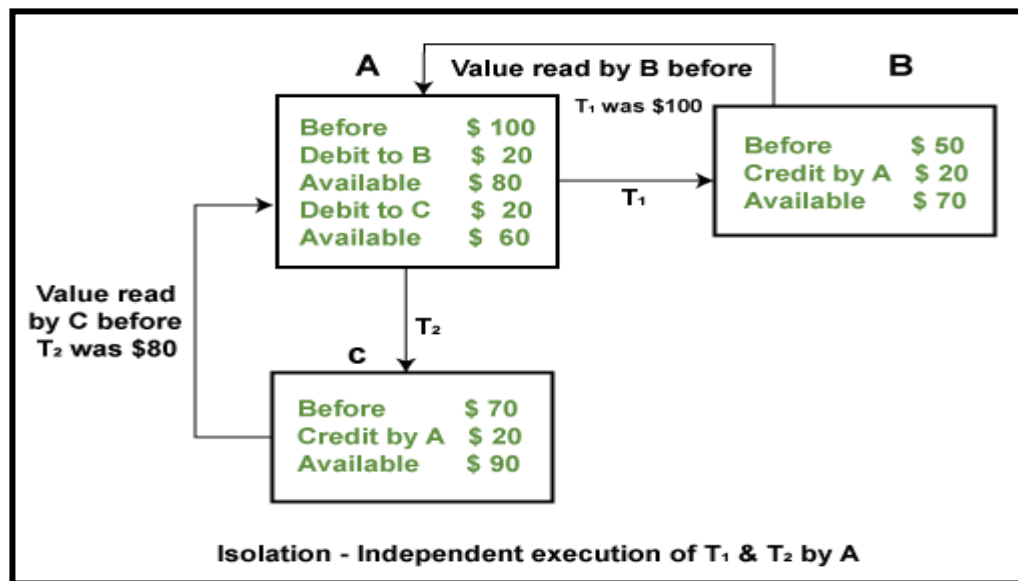
In the above figure, there are three accounts, A, B, and C, where A is making a transaction T one by one to both B & C. There are two operations that take place, i.e., Debit and Credit. Account A firstly debits \$50 to account B, and the amount in account A is read \$300 by B before the transaction. 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.

**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.



### Durability:

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.

### **Concurrent Executions**

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.

## Problems with Concurrent Execution

In a database transaction, the two main operations are **READ** and **WRITE** operations. So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent. So, the following problems occur with the Concurrent Execution of the operations:

- **Lost Update Problems (W - W Conflict)**
- **Dirty Read Problems (W-R Conflict)**
- **Unrepeatable Read Problem (W-R Conflict)**

### Lost Update Problems (W - W Conflict)

The problem occurs when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent.

#### For example:

Consider the below diagram where two transactions  $T_x$  and  $T_y$ , are performed on the same account A where the balance of account A is \$300.

Time	$T_x$	$T_y$
$t_1$	READ (A)	—
$t_2$	$A = A - 50$	—
$t_3$	—	READ (A)
$t_4$	—	$A = A + 100$
$t_5$	—	—
$t_6$	WRITE (A)	—
$t_7$	—	WRITE (A)

**LOST UPDATE PROBLEM**

- At time  $t_1$ , transaction  $T_x$  reads the value of account A, i.e., \$300 (only read).
- At time  $t_2$ , transaction  $T_x$  deducts \$50 from account A that becomes \$250 (only deducted and not updated/write).
- Alternately, at time  $t_3$ , transaction  $T_y$  reads the value of account A that will be \$300 only because  $T_x$  didn't update the value yet.
- At time  $t_4$ , transaction  $T_y$  adds \$100 to account A that becomes \$400 (only added but not updated/write).
- At time  $t_6$ , transaction  $T_x$  writes the value of account A that will be updated as \$250 only, as  $T_y$  didn't update the value yet.
- Similarly, at time  $t_7$ , transaction  $T_y$  writes the values of account A, so it will write as done at time  $t_4$  that will be \$400. It means the value written by  $T_x$  is lost, i.e., \$250 is lost.

Hence data becomes incorrect, and database sets to inconsistent.

## Dirty Read Problems (W-R Conflict)

The dirty read problem occurs when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.

### For example:

Consider two transactions  $T_x$  and  $T_y$  in the below diagram performing read/write operations on account A where the available balance in account A is \$300:

Time	$T_x$	$T_y$
$t_1$	READ (A)	—
$t_2$	$A = A + 50$	—
$t_3$	WRITE (A)	—
$t_4$	—	READ (A)
$t_5$	SERVER DOWN ROLLBACK	—

**DIRTY READ PROBLEM**

- At time  $t_1$ , transaction  $T_x$  reads the value of account A, i.e., \$300.
- At time  $t_2$ , transaction  $T_x$  adds \$50 to account A that becomes \$350.
- At time  $t_3$ , transaction  $T_x$  writes the updated value in account A, i.e., \$350.
- Then at time  $t_4$ , transaction  $T_y$  reads account A that will be read as \$350.
- Then at time  $t_5$ , transaction  $T_x$  rollbacks due to server problem and the value changes back to \$300 (as initially).
- But the value for account A remains \$350 for transaction  $T_y$  as committed, which is the dirty read and therefore known as the Dirty Read Problem.

**Unrepeatable Read Problem (W-R Conflict)**

Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

**For example:**

Consider two transactions,  $T_x$  and  $T_y$ , performing the read/write operations on account A, having an available balance = \$300. The diagram is shown below:

Time	$T_x$	$T_y$
$t_1$	READ (A)	—
$t_2$	—	READ (A)
$t_3$	—	$A = A + 100$
$t_4$	—	WRITE (A)
$t_5$	READ (A)	—

**UNREPEATABLE READ PROBLEM**

- At time  $t_1$ , transaction  $T_x$  reads the value from account A, i.e., \$300.
- At time  $t_2$ , transaction  $T_y$  reads the value from account A, i.e., \$300.

- At time  $t_3$ , transaction  $T_Y$  updates the value of account A by adding \$100 to the available balance, and then it becomes \$400.
- At time  $t_4$ , transaction  $T_Y$  writes the updated value, i.e., \$400.
- After that, at time  $t_5$ , transaction  $T_X$  reads the available value of account A, and that will be read as \$400.
- It means that within the same transaction  $T_X$ , it reads two different values of account A, i.e., \$ 300 initially, and after updation made by transaction  $T_Y$ , it reads \$400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.

### DBMS Serializability

When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state. Serializability is a concept that helps us to check which schedules are serializable. A serializable schedule is the one that always leaves the database in consistent state.

#### What is a serializable schedule?

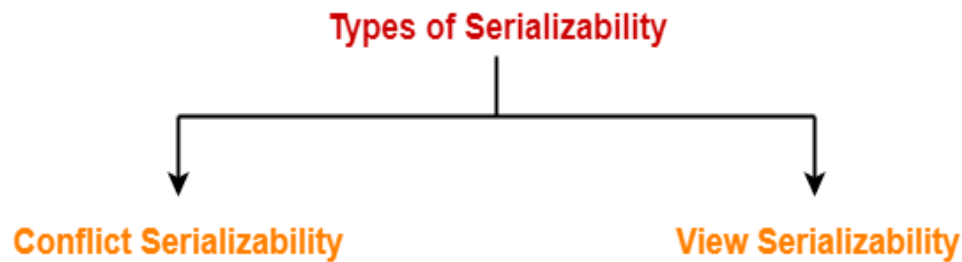
A serializable schedule always leaves the database in consistent state. A serial schedule is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution. However, a non-serial schedule needs to be checked for Serializability.

A non-serial schedule of  $n$  number of transactions is said to be serializable schedule, if it is equivalent to the serial schedule of those  $n$  transactions. A serial schedule doesn't allow concurrency, only one transaction executes at a time and the other starts when the already running transaction finished.

#### Types of Serializability

There are two types of Serializability.

1. Conflict Serializability
2. View Serializability



### Conflict Serializability

In the DBMS Schedules, there are two types of schedules - Serial & Non-Serial. A Serial schedule doesn't support concurrent execution of transactions while a non-serial schedule supports concurrency. In Serializability that a non-serial schedule may leave the database in inconsistent state so we need to check these non-serial schedules for the Serializability.

A schedule is said to be conflict serializable if it can transform into a serial schedule after swapping of non-conflicting operations. It is a type of serializability that can be used to check whether the non-serial schedule is conflict serializable or not.

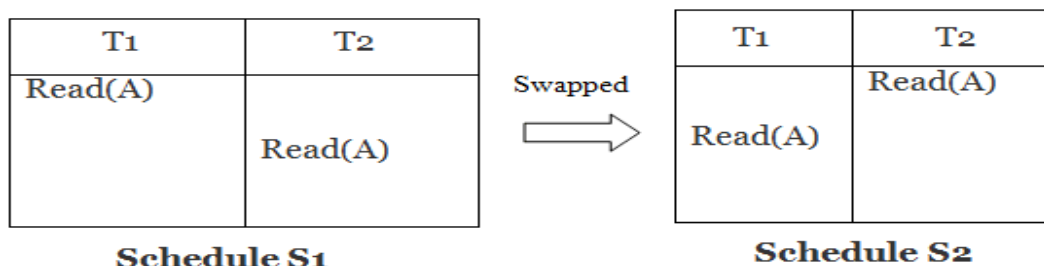
### Conflicting operations

The two operations are called conflicting operations, if all the following three conditions are satisfied:

- Both the operation belongs to separate transactions.
- Both works on the same data item.
- At least one of them contains one write operation.

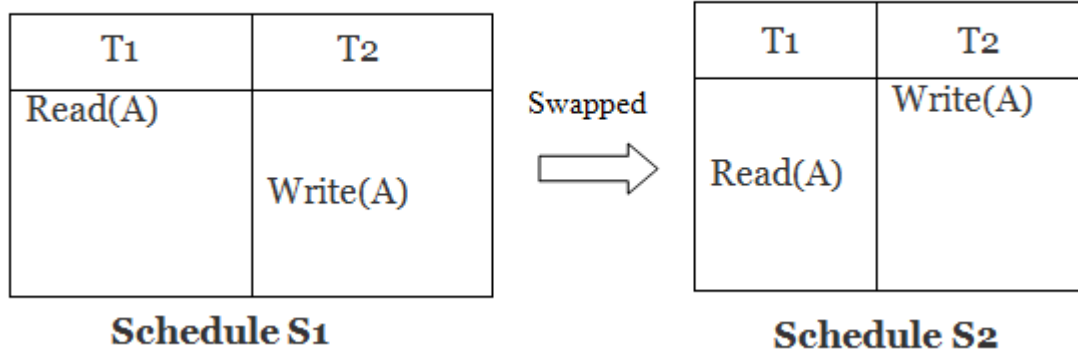
**Example:** Swapping is possible only if S1 and S2 are logically equal.

**1. T1: Read(A)   T2: Read(A)**



Here, S1 = S2. That means it is non-conflict.

**2. T1: Read(A) T2: Write(A)**



Here,  $S1 \neq S2$ . That means it is conflict.

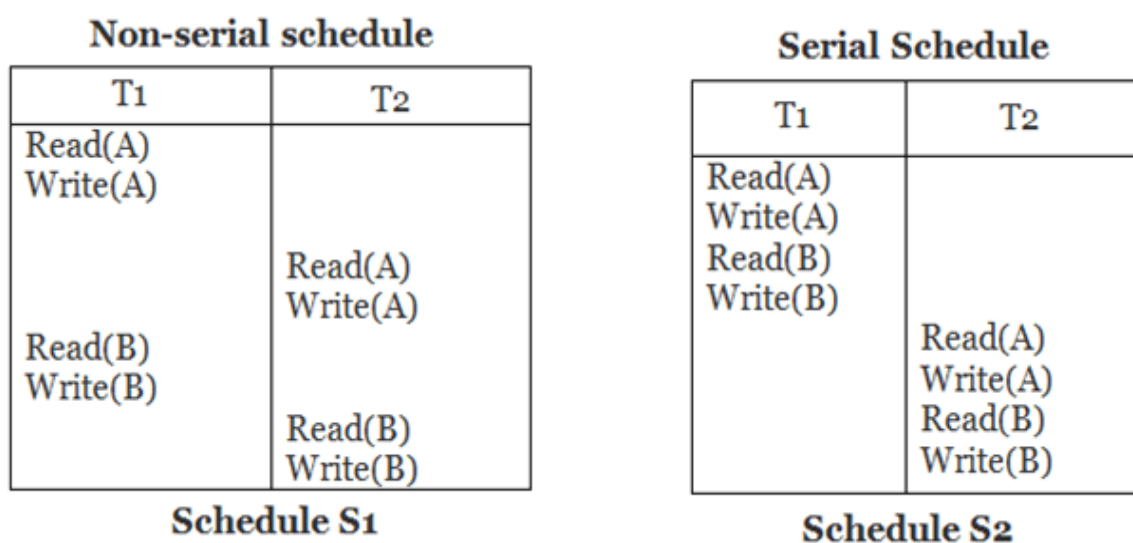
**Conflict Equivalent**

In the conflict equivalent, one can be transformed to another by swapping non-conflicting operations. In the given example, S2 is conflict equivalent to S1 (S1 can be converted to S2 by swapping non-conflicting operations).

Two schedules are said to be conflict equivalent if and only if:

- They contain the same set of the transaction.
- If each pair of conflict operations are ordered in the same way.

**Example:**



Schedule S2 is a serial schedule because, in this, all operations of T1 are performed before starting any operation of T2. Schedule S1 can be transformed into a serial schedule by swapping non-conflicting operations of S1.

After swapping of non-conflict operations, the schedule S1 becomes:

<b>Serial Schedule</b>	
<b>T1</b>	<b>T2</b>
Read(A) Write(A) Read(B) Write(B)	Read(A) Write(A) Read(B) Write(B)

Since, **S1** is conflict serializable.

### **View Serializability**

It is a type of serializability that can be used to check whether the given schedule is view serializable or not. A schedule called as a view serializable if it is view equivalent to a serial schedule.

### **View Equivalent**

Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:

- **Initial Read**
- **Updated Read**
- **Final Write**

### **Initial Read**

An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.

<b>T1</b>	<b>T2</b>	<b>T1</b>	<b>T2</b>
Read(A)	Write(A)	Read(A)	Write(A)

**Schedule S1**
**Schedule S2**

Above two schedules are view equivalent because Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

### Updated Read

In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.

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

**Schedule S1**

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

**Schedule S2**

Above two schedules are not view equal because, in S1, T3 is reading A updated by T2 and in S2, T3 is reading A updated by T1.

### Final Write

A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.

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

**Schedule S1**

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

**Schedule S2**

Above two schedules are view equal because Final write operation in S1 is done by T3 and in S2, the final write operation is also done by T3.

### **Recoverability**

Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback.



But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		
Failure Point				
Commit;				

The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1, but T2 can't be rollback because it already committed. So this type of schedule is known as irrecoverable schedule.

**Irrecoverable schedule:** The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
Failure Point				
Commit;				
		Commit;		

The above table shows a schedule with two transactions. Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, we have to rollback T1. T2 should be rollback because T2 has read the value written by T1. As it has not committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with cascade rollback.

**Recoverable with cascading rollback:** The schedule will be recoverable with cascading rollback if  $T_j$  reads the updated value of  $T_i$ . Commit of  $T_j$  is delayed till commit of  $T_i$ .

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
Commit;		Read(A);	A = 6000	A = 6000
		A = A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		

The above Table shows a schedule with two transactions. Transaction **T1** reads and write **A** and commits, and that value is read and written by **T2**. So this is a cascade less recoverable schedule.

### Testing for 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  $T_i \rightarrow T_j$  for which one of the three conditions holds:

1. Create a node  $T_i \rightarrow T_j$  if  $T_i$  executes write (Q) before  $T_j$  executes read (Q).
2. Create a node  $T_i \rightarrow T_j$  if  $T_i$  executes read (Q) before  $T_j$  executes write (Q).
3. Create a node  $T_i \rightarrow T_j$  if  $T_i$  executes write (Q) before  $T_j$  executes write (Q).

### Precedence graph for Schedule S



- If a precedence graph contains a single edge  $T_i \rightarrow T_j$ , then all the instructions of  $T_i$  are executed before the first instruction of  $T_j$  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:

	T1	T2	T3
Time ↓	Read(A)	Read(B)	
	A := f <sub>1</sub> (A)		Read(C)
		B := f <sub>2</sub> (B) Write(B)	C := f <sub>3</sub> (C) Write(C)
	Write(A)		Read(B)
	Read(C)	Read(A) A := f <sub>4</sub> (A)	
	C := f <sub>5</sub> (C) Write(C)	Write(A)	B := f <sub>6</sub> (B) Write(B)

Schedule S1

Explanation:

**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  $T2 \rightarrow T3$

**Write(C):** C is subsequently read by T1, so add edge  $T3 \rightarrow T1$

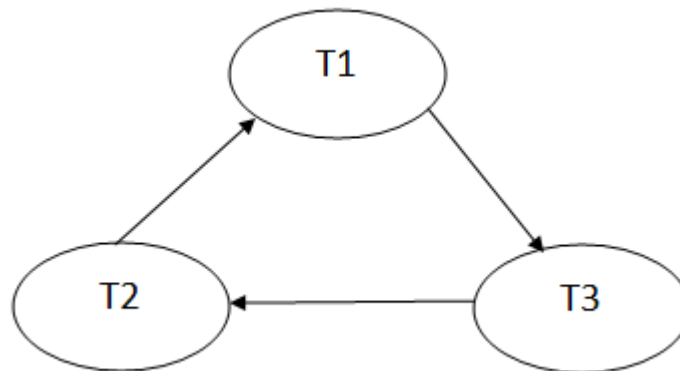
**Write(A):** A is subsequently read by T2, so add edge  $T1 \rightarrow T2$

**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.

	T4	T5	T6
Time ↓	Read(A)		
	A:= f1(A)		
	Read(C)		
	Write(A)		
	A:= f2(C)		
	Write(C)	Read(B)	
		Read(A)	
		B:= f3(B)	
		Write(B)	
			Read(C)
			C:= f4(C)
			Read(B)
			Write(C)
		A:= f5(A)	
		Write(A)	
			B:= f6(B)
			Write(B)

**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

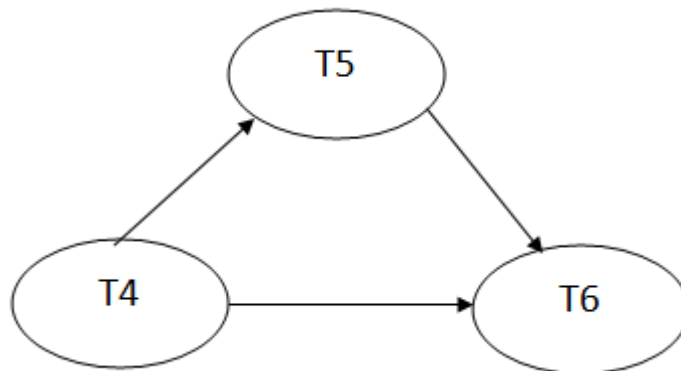
**Write(B):** A is subsequently read by T6, so add edge T5 → T6

**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 Schedule S2 is serializable.

## Implementation of Isolation

The execution of every transaction must be done in an isolated manner, such that execution of a transaction is not known to any other transaction i.e., every transaction must execute independently. The intermediate results generated by the transactions should not be available to other transactions.

The isolation property can be ensured by concurrency control component of a database system. According to this property, the transactions that are being executed concurrently obtain same result even if they are executed serially. The transactions are executed concurrently, because such execution enhances the system performance, but at the same time results in an inconsistent state due to the occurrence of various abnormalities.

During the concurrent execution of transactions, different concurrency control mechanisms can be used which guarantee that only acceptable schedules are generated.

Let us consider one such concurrency control mechanisms, suppose that a transaction  $T_A$ , acquires a lock on a database. If another transaction  $T_B$ , wishes to access

that database, then it has to wait until the first transaction  $T_A$  that hold the lock releases it. The transaction releases the lock only when the entire transaction is committed.

This locking policy decreases the performance of the system as only a single transaction is executed at a time, because of this execution only serial schedules are generated. This concurrency control mechanism provides poor concurrency level.

The major objective of concurrency control scheme is to provide high degree of concurrency and to generate conflict, view and cascadeless schedules.

### **Failure Classification**

A failure in DBMS is categorized into the following three classifications to ease the process of determining the exact nature of the problem:

- **Transaction failure**
- **Disk failure**
- **System crash**

#### **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.

#### **System Crash**

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.

**Disk Failure**

- 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.
- 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.

**Storage System in DBMS**

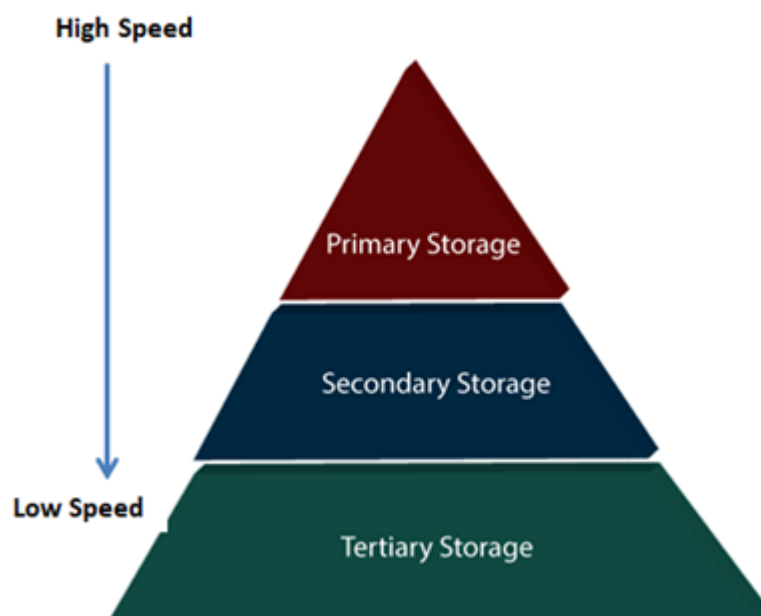
A database system provides an ultimate view of the stored data. However, data in the form of bits, bytes get stored in different storage devices.

We will take an overview of various types of storage devices that are used for accessing and storing data.

**Types of Data Storage**

For storing the data, there are different types of storage options available. These storage types differ from one another as per the speed and accessibility. There are the following types of storage devices used for storing the data:

- **Primary Storage**
- **Secondary Storage**
- **Tertiary Storage**



**Primary Storage**

It is the primary area that offers quick access to the stored data. We also know the primary storage as volatile storage. It is because this type of memory does not permanently store the data. As soon as the system leads to a power cut or a crash, the data also get lost. Main memory and cache are the types of primary storage.

- **Main Memory**: It is the one that is responsible for operating the data that is available by the storage medium. The main memory handles each instruction of a computer machine. This type of memory can store gigabytes of data on a system but is small enough to carry the entire database. At last, the main memory loses the whole content if the system shuts down because of power failure or other reasons.
- **Cache**: It is one of the costly storage media. On the other hand, it is the fastest one. A cache is a tiny storage media which is maintained by the computer hardware usually. While designing the algorithms and query processors for the data structures, the designers keep concern on the cache effects.

**Secondary Storage**

Secondary storage is also called as online storage. It is the storage area that allows the user to save and store data permanently. This type of memory does not lose the data due to any power failure or system crash. That's why we also call it non-volatile storage.

There are some commonly described secondary storage media which are available in almost every type of computer system:

- **Flash Memory**: A flash memory stores data in USB (Universal Serial Bus) keys which are further plugged into the USB slots of a computer system. These USB keys help transfer data to a computer system, but it varies in size limits. Unlike the main memory, it is possible to get back the stored data which may be lost due to a power cut or other reasons. This type of memory storage is most commonly used in the server systems for caching the frequently used data. This leads the systems towards high performance and is capable of storing large amounts of databases than the main memory.
- **Magnetic Disk Storage**: This type of storage media is also known as online storage media. A magnetic disk is used for storing the data for a long time. It is capable of



storing an entire database. It is the responsibility of the computer system to make availability of the data from a disk to the main memory for further accessing. Also, if the system performs any operation over the data, the modified data should be written back to the disk. The tremendous capability of a magnetic disk is that it does not affect the data due to a system crash or failure, but a disk failure can easily ruin as well as destroy the stored data.

### **Tertiary Storage**

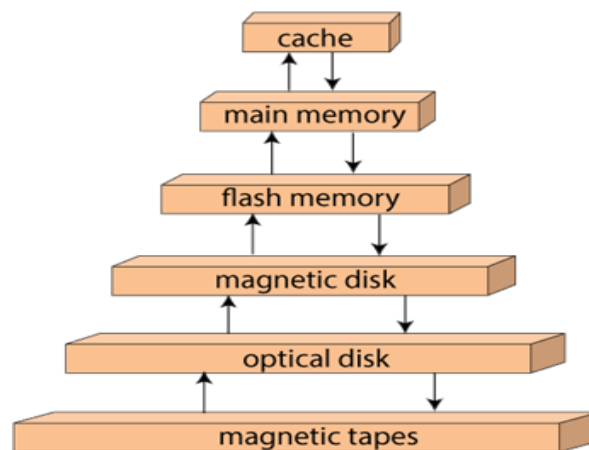
It is the storage type that is external from the computer system. It has the slowest speed. But it is capable of storing a large amount of data. It is also known as Offline storage. Tertiary storage is generally used for data backup. There are following tertiary storage devices available:

- **Optical Storage:** An optical storage can store megabytes or gigabytes of data. A Compact Disk (CD) can store 700 megabytes of data with a playtime of around 80 minutes. On the other hand, a Digital Video Disk or a DVD can store 4.7 or 8.5 gigabytes of data on each side of the disk.
- **Tape Storage:** It is the cheapest storage medium than disks. Generally, tapes are used for archiving or backing up the data. It provides slow access to data as it accesses data sequentially from the start. Thus, tape storage is also known as sequential-access storage. Disk storage is known as direct-access storage as we can directly access the data from any location on disk.

### **Storage Hierarchy**

Besides the above, various other storage devices reside in the computer system. These storage media are organized on the basis of data accessing speed, cost per unit of data to buy the medium, and by medium's reliability. Thus, we can create a hierarchy of storage media on the basis of its cost and speed.

Thus, on arranging the above-described storage media in a hierarchy according to its speed and cost, we conclude the below-described image:



**Storage device hierarchy**

In the image, the higher levels are expensive but fast. On moving down, the cost per bit is decreasing, and the access time is increasing. Also, the storage media from the main memory to up represents the volatile nature, and below the main memory, all are non-volatile devices.

### **Recovery and Atomicity**

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following –

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction –

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

### **Log-based Recovery**

Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to the actual modification and stored on a stable storage media, which is failsafe.

Log-based recovery works as follows –

- The log file is kept on a stable storage media.
- When a transaction enters the system and starts execution, it writes a log about it.

**<T<sub>n</sub>, Start>**

- When the transaction modifies an item X, it write logs as follows –

**<T<sub>n</sub>, X, V<sub>1</sub>, V<sub>2</sub>>**

It reads T<sub>n</sub> has changed the value of X, from V<sub>1</sub> to V<sub>2</sub>.

- When the transaction finishes, it logs –

**<T<sub>n</sub>, commit>**

The database can be modified using two approaches –

- **Deferred database modification** – All logs are written on to the stable storage and the database is updated when a transaction commits.
- **Immediate database modification** – Each log follows an actual database modification. That is, the database is modified immediately after every operation.

### **Recovery with Concurrent Transactions**

When more than one transaction is being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system to

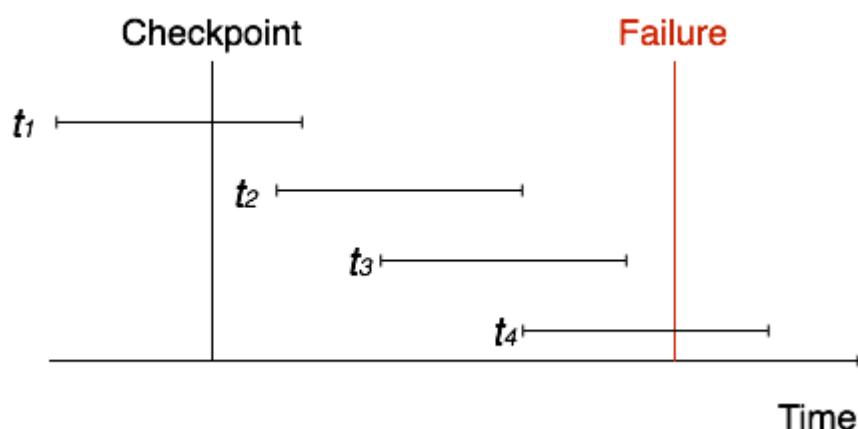
backtrack all logs, and then start recovering. To ease this situation, most modern DBMS use the concept of 'checkpoints'.

## Checkpoint

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

## Recovery

When a system with concurrent transactions crashes and recovers, it behaves in the following manner -



- 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  $\langle T_n, \text{Start} \rangle$  and  $\langle T_n, \text{Commit} \rangle$  or just  $\langle T_n, \text{Commit} \rangle$ , it puts the transaction in the redo-list.
- If the recovery system sees a log with  $\langle T_n, \text{Start} \rangle$  but no commit or abort log found, it puts the transaction in undo-list.

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.

## Indexing in DBMS

Indexing is a data structure technique which allows you to quickly retrieve records from a database file. An Index is a small table having only two columns. The first column comprises a copy of the primary or candidate key of a table. Its second column contains a set of **pointers** for holding the address of the disk block where that specific key value stored.

### Index structure:

Indexes can be created using some database columns.

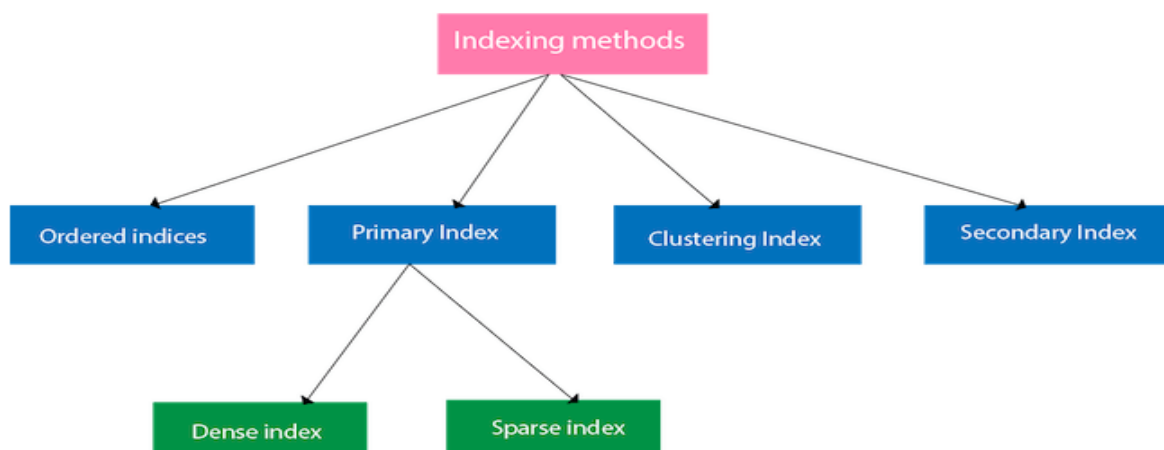
Search key	Data Reference
------------	-------------------

**Fig: Structure of Index**

- The first column of the database is the search key that contains a copy of the primary key or candidate key of the table. The values of the primary key are stored in sorted order so that the corresponding data can be accessed easily.
- The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.

### Types of Indexing

Indexing is defined based on its indexing attributes. Indexing can be of the following types –



### **Ordered indices**

The indices are usually sorted to make searching faster. The indices which are sorted are known as ordered indices.

**Example:** Suppose we have an employee table with thousands of record and each of which is 10 bytes long. If their **IDs** start with 1, 2, 3....and so on and we have to search student with **ID-543**.

- ✓ In the case of a database with no index, we have to search the disk block from starting till it reaches 543. The DBMS will read the record after reading  $543 \times 10 = 5430$  bytes.
- ✓ In the case of an index, we will search using indexes and the DBMS will read the record after reading  $542 \times 2 = 1084$  bytes which are very less compared to the previous case.

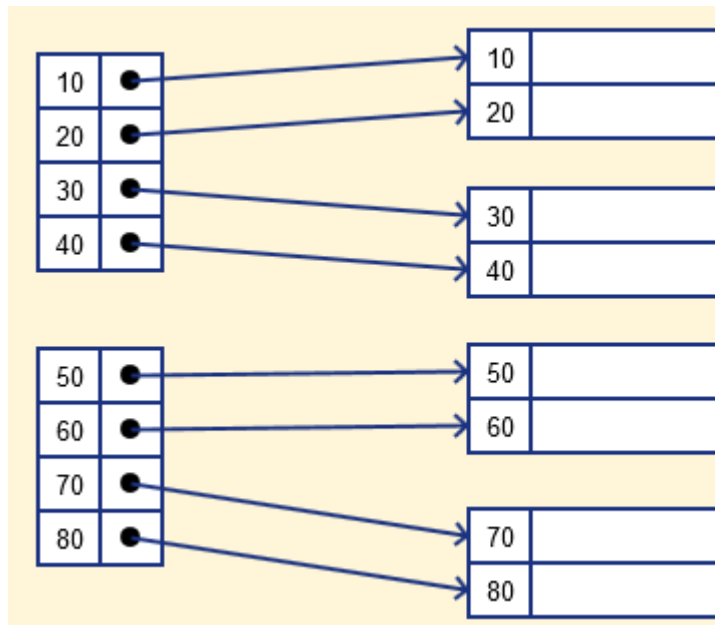
### **Primary Index**

- If the index is created on the basis of the primary key of the table, then it is known as primary indexing. These primary keys are unique to each record and contain 1:1 relation between the records.
- Primary Index is an ordered file which is fixed length size with two fields. The first field is the same a primary key and second, filed is pointed to that specific data block.
- As primary keys are stored in sorted order, the performance of the searching operation is quite efficient.
- The primary index can be classified into two types.
  - ❖ Dense Index
  - ❖ Sparse Index

### **Dense Index**

- The dense index contains an index record for every search key value in the data file. It makes searching faster.
- In this, the number of records in the index table is same as the number of records in the main table.

- It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.

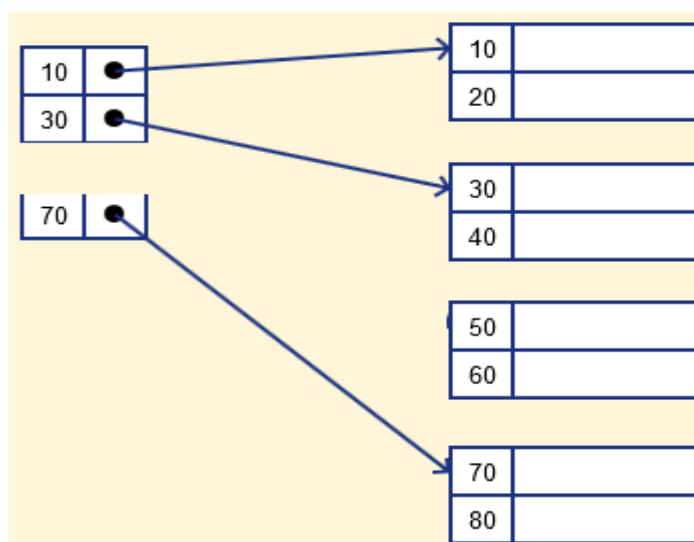


## Sparse Index

It is an index record that appears for only some of the values in the file. Sparse Index helps you to resolve the issues of dense Indexing in DBMS. In this method of indexing technique, a range of index columns stores the same data block address, and when data needs to be retrieved, the block address will be fetched.

However, sparse Index stores index records for only some search-key values. It needs less space, less maintenance overhead for insertion, and deletions but It is slower compared to the dense Index for locating records.

Below is a database index Example of Sparse Index

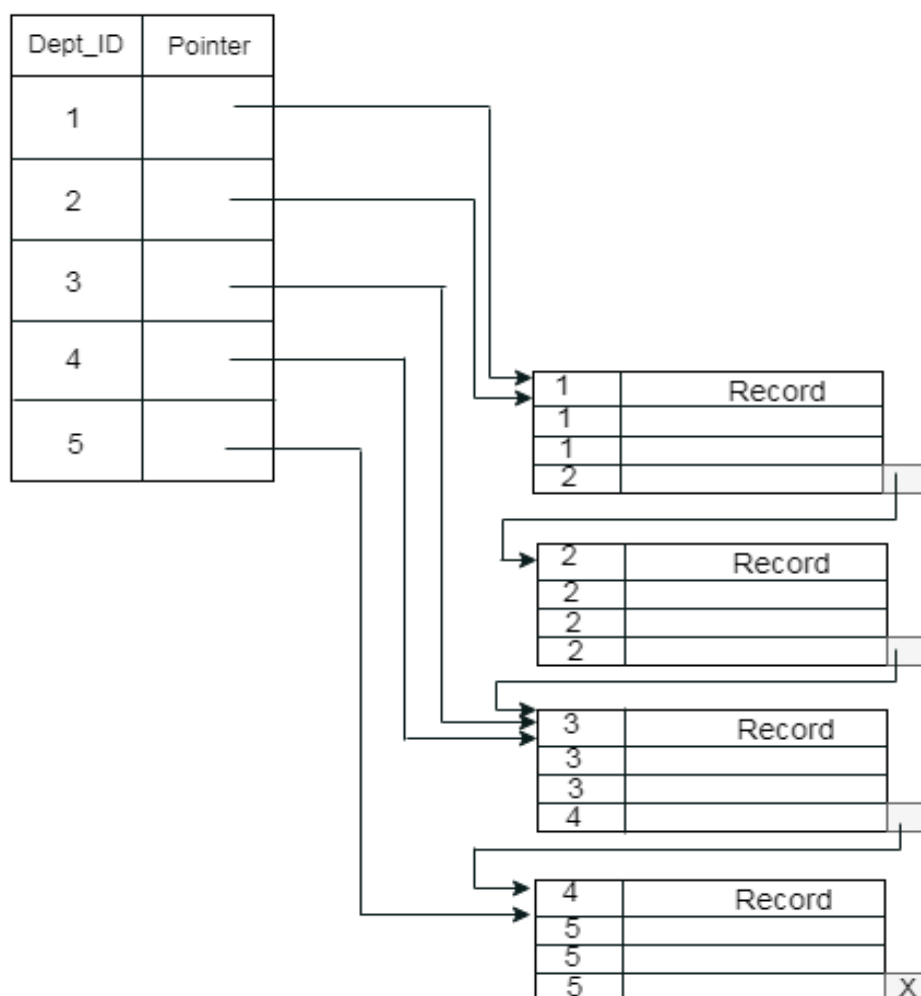


## Clustering Index

- A clustered index can be defined as an ordered data file. Sometimes the index is created on non-primary key columns which may not be unique for each record.
- In this case, to identify the record faster, we will group two or more columns to get the unique value and create index out of them. This method is called a clustering index.
- The records which have similar characteristics are grouped, and indexes are created for these group.

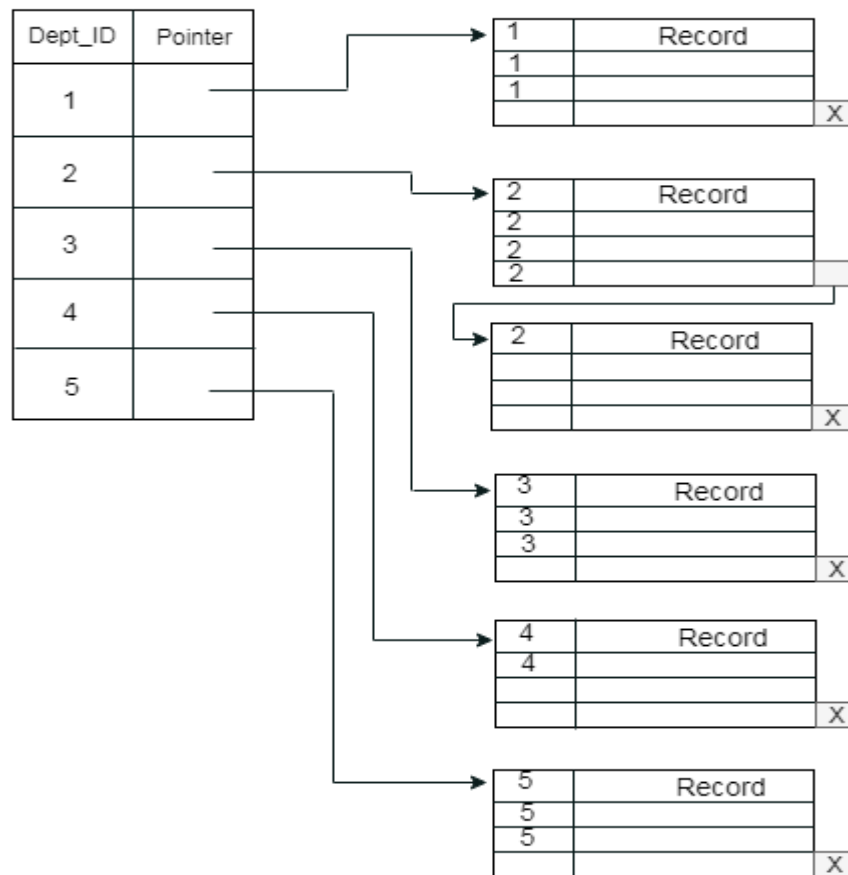
### Example:

Suppose a company contains several employees in each department. Suppose we use a clustering index, where all employees which belong to the same **Dept\_ID** are considered within a single cluster, and index pointers point to the cluster as a whole. Here **Dept\_Id** is a non-unique key.





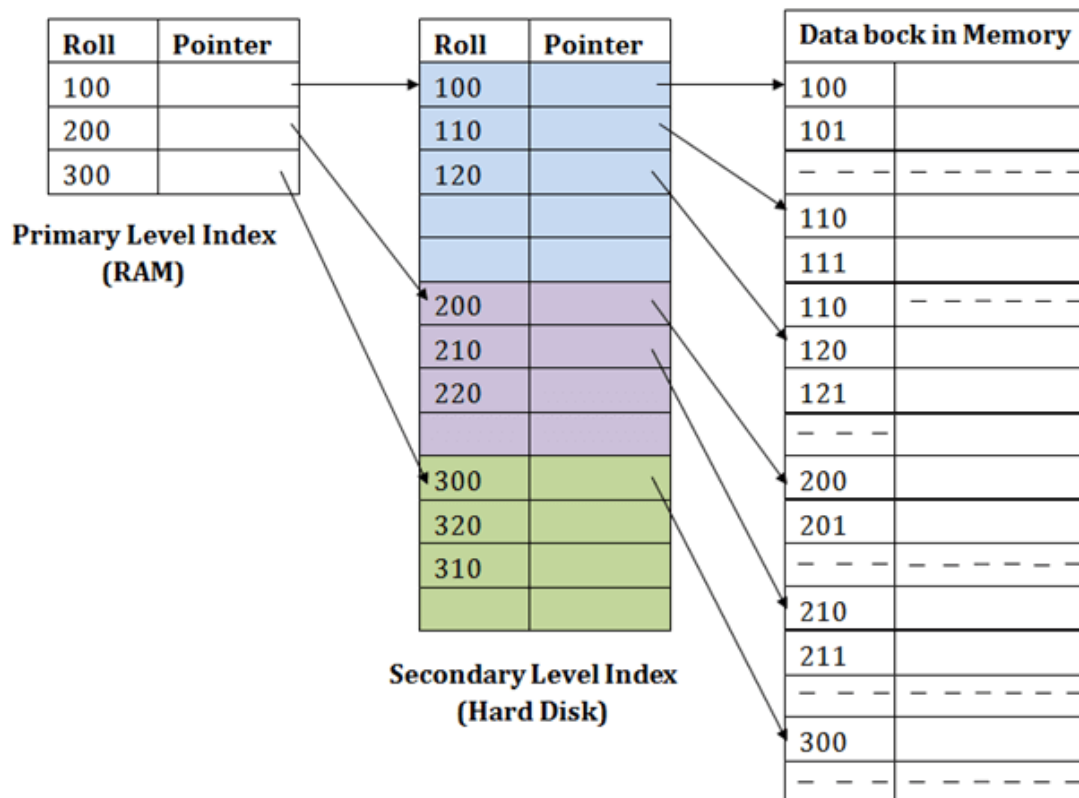
The previous schema is little confusing because one disk block is shared by records which belong to the different cluster. If we use separate disk block for separate clusters, then it is called better technique.



## Secondary Index

In the sparse indexing, as the size of the table grows, the size of mapping also grows. These mappings are usually kept in the primary memory so that address fetch should be faster. Then the secondary memory searches the actual data based on the address got from mapping. If the mapping size grows then fetching the address itself becomes slower. In this case, the sparse index will not be efficient. To overcome this problem, secondary indexing is introduced.

In secondary indexing, to reduce the size of mapping, another level of indexing is introduced. In this method, the huge range for the columns is selected initially so that the mapping size of the first level becomes small. Then each range is further divided into smaller ranges. The mapping of the first level is stored in the primary memory, so that address fetch is faster. The mapping of the second level and actual data are stored in the secondary memory (hard disk).



### For example:

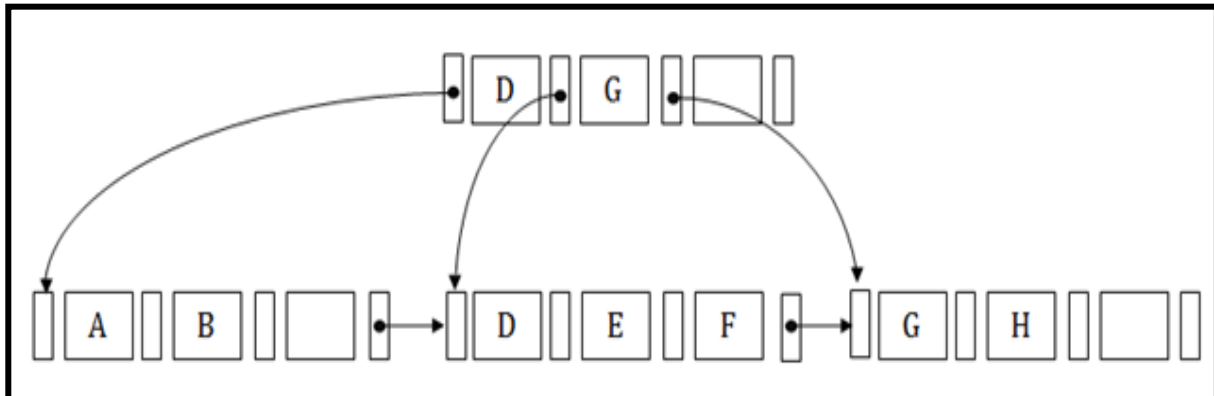
- If you want to find the record of roll 111 in the diagram, then it will search the highest entry which is smaller than or equal to 111 in the first level index. It will get 100 at this level.
- Then in the second index level, again it does  $\max(111) \leq 111$  and gets 110. Now using the address 110, it goes to the data block and starts searching each record till it gets 111.
- This is how a search is performed in this method. Inserting, updating or deleting is also done in the same manner.

### Introduction of B+ Trees

- The B+ tree is a balanced binary search tree. It follows a multi-level index format.
- In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.
- In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.

## Structure of B+ Tree

- In the B+ tree, every leaf node is at equal distance from the root node. The B+ tree is of the order  $n$  where  $n$  is fixed for every B+ tree.
- It contains an internal node and leaf node.



## Internal node

- An internal node of the B+ tree can contain at least  $n/2$  record pointers except the root node.
- At most, an internal node of the tree contains  $n$  pointers.

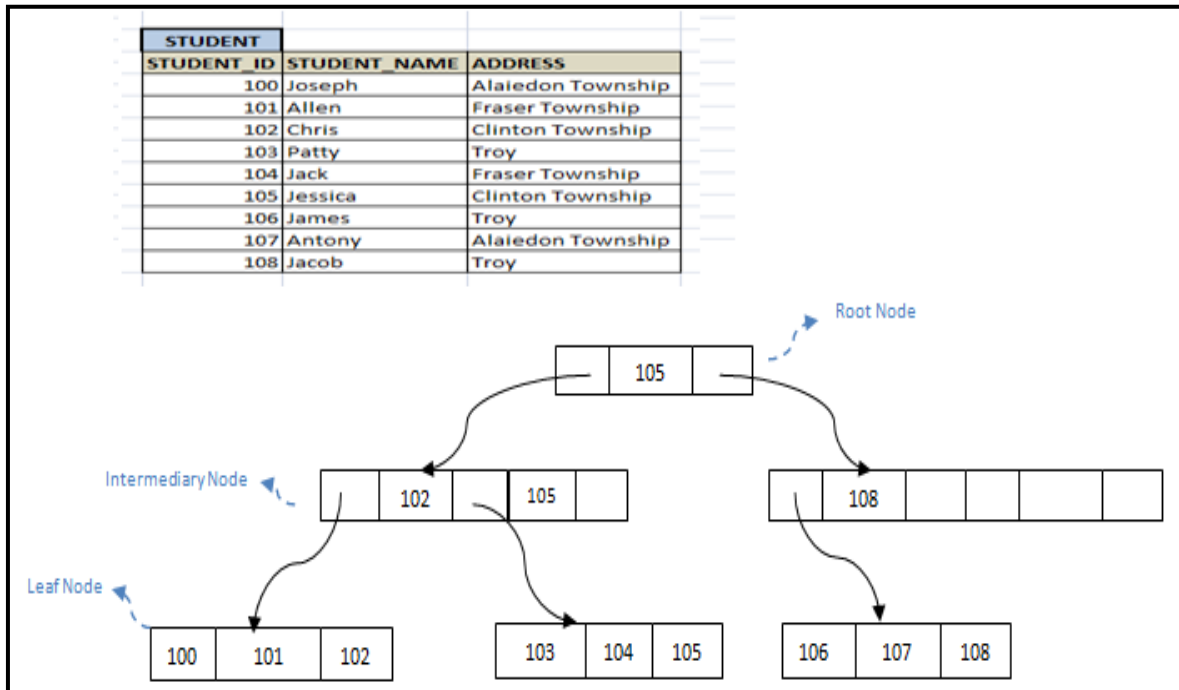
## Leaf node

- The leaf node of the B+ tree can contain at least  $n/2$  record pointers and  $n/2$  key values.
- At most, a leaf node contains  $n$  record pointer and  $n$  key values.
- Every leaf node of the B+ tree contains one block pointer  $P$  to point to next leaf node.

Consider the **STUDENT** table below. This can be stored in B+ tree structure as shown below. We can observe here that it divides the records into two and splits into **left** node and **right** node.

Left node will have all the values less than or equal to root node and the right node will have values greater than root node. The intermediary nodes at level 2 will have only the pointers to the leaf nodes.

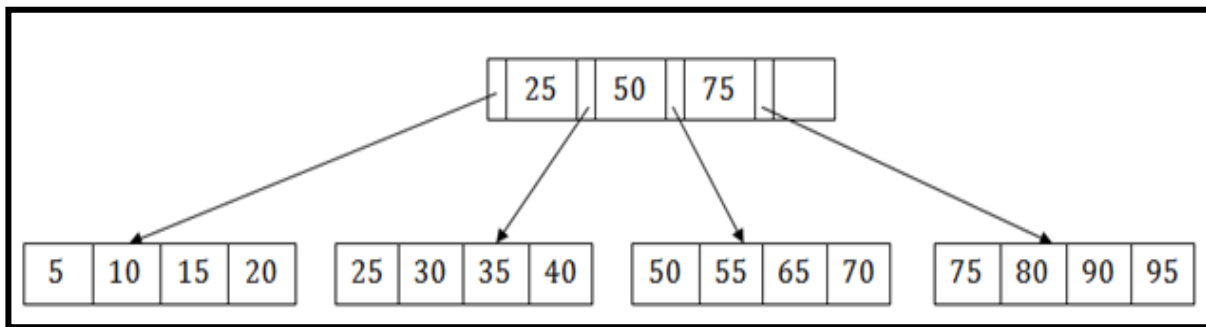
The values shown in the intermediary nodes are only the pointers to next level. All the leaf nodes will have the actual records in a sorted order.



If we have to search for any record, they are all found at leaf node. Hence searching any record will take same time because of equidistance of the leaf nodes. Also they are all sorted. Hence searching a record is like a sequential search and does not take much time.

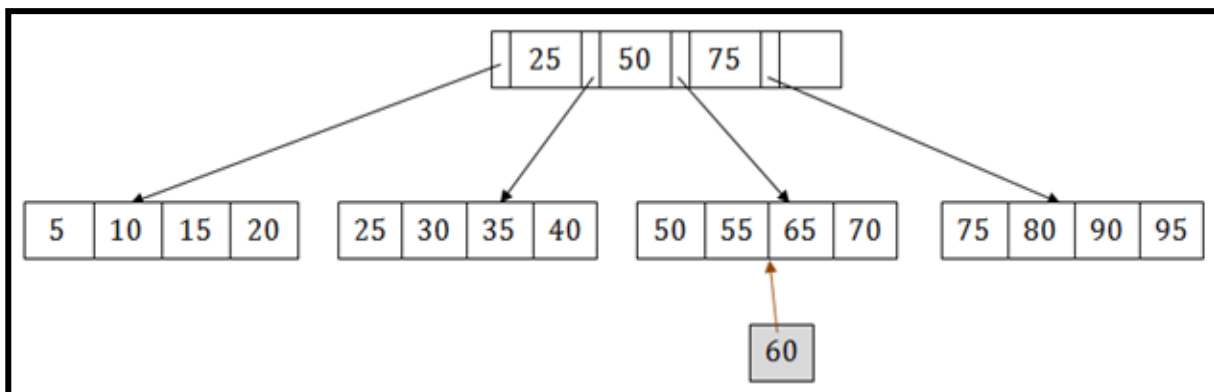
## Searching a record in B+ Tree

Suppose we want to search 65 in the below B+ tree structure. First we will fetch for the intermediary node which will direct to the leaf node that can contain record for 65. So we find branch between 50 and 75 nodes in the intermediary node. Then we will be redirected to the third leaf node at the end. Here DBMS will perform sequential search to find 65. Suppose, instead of 65, we have to search for 60. What will happen in this case? We will not be able to find in the leaf node. No insertions/update/delete is allowed during the search in B+ tree.

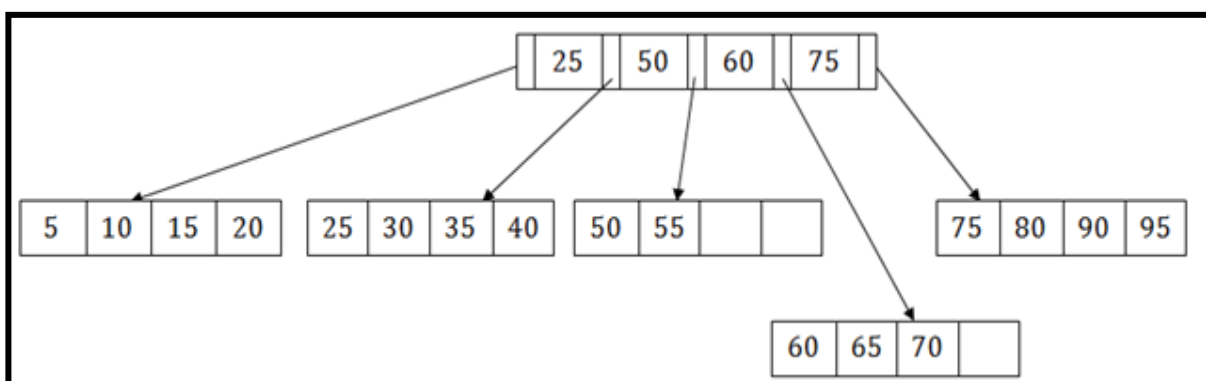


**Insertion in B+ tree**

Suppose we have to insert a record 60 in below structure. It will go to 3rd leaf node after 55. Since it is a balanced tree and that leaf node is already full, we cannot insert the record there. But it should be inserted there without affecting the fill factor, balance and order. So the only option here is to split the leaf node. But how do we split the nodes?



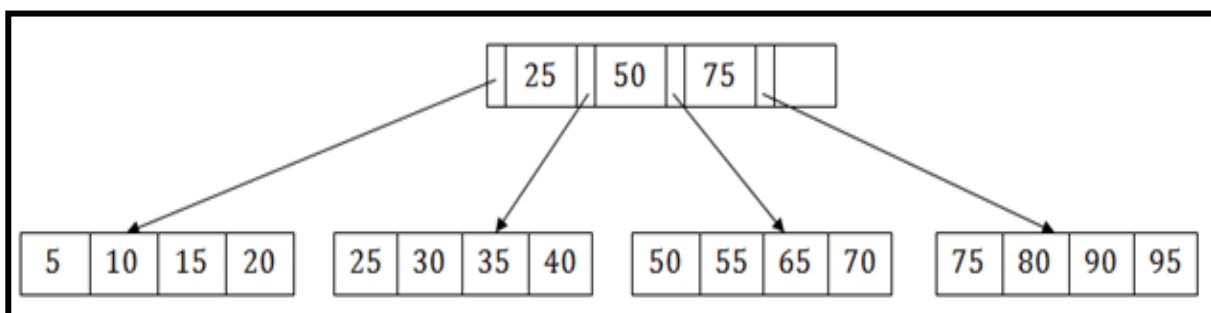
The 3rd leaf node should have values (50, 55, 60, 65, 70) and its current root node is 50. We will split the leaf node in the middle so that its balance is not altered. So we can group (50, 55) and (60, 65, 70) into 2 leaf nodes. If these two has to be leaf nodes, the intermediary node cannot branch from 50. It should have 60 added to it and then we can have pointers to new leaf node.



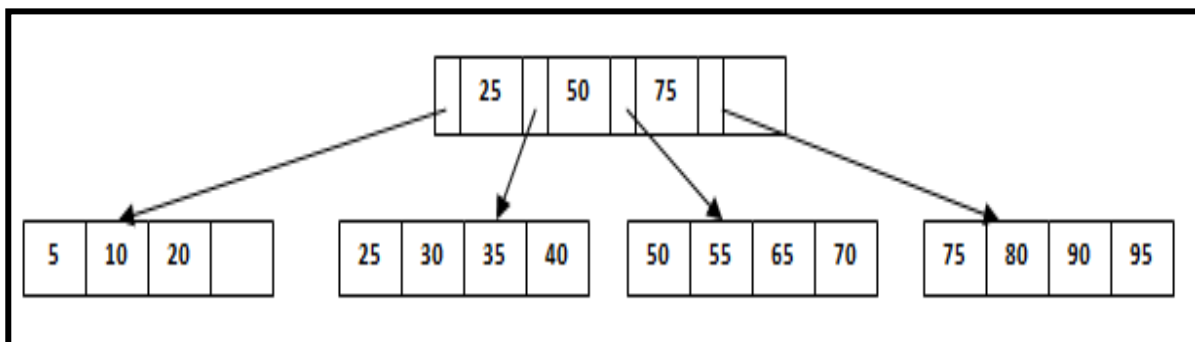
This is how we insert a new entry when there is overflow. In normal scenario, it is simple to find the node where it fits and place it in that leaf node.

## Delete in B+ tree

Suppose we have to delete 60 from the above example. What will happen in this case? We have to remove 60 from 4th leaf node as well as from the intermediary node too. If we remove it from intermediary node, the tree will not satisfy B+ tree rules. So we need to modify it have a balanced tree. After deleting 60 from above B+ tree and re-arranging nodes, it will appear as below.



Suppose we have to delete 15 from above tree. We will traverse to the 1st leaf node and simply delete 15 from that node. There is no need for any re-arrangement as the tree is balanced and 15 do not appear in the intermediary node.



## File Organization and Indexing

- The **File** is a collection of records. Using the primary key, we can access the records. The type and frequency of access can be determined by the type of file organization which was used for a given set of records.
- File organization is a logical relationship among various records. This method defines how file records are mapped onto disk blocks.

- File organization is used to describe the way in which the records are stored in terms of blocks, and the blocks are placed on the storage medium.
- The first approach to map the database to the file is to use the several files and store only one fixed length record in any given file. An alternative approach is to structure our files so that we can contain multiple lengths for records.
- Files of fixed length records are easier to implement than the files of variable length records.

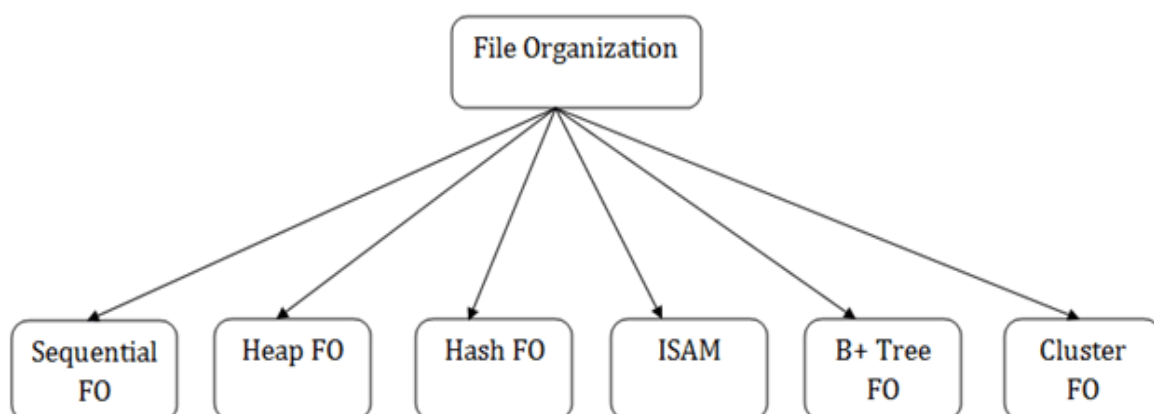
### **Objective of file organization**

- It contains an optimal selection of records, i.e., records can be selected as fast as possible.
- To perform insert, delete or update transaction on the records should be quick and easy.
- The duplicate records cannot be induced as a result of insert, update or delete.
- For the minimal cost of storage, records should be stored efficiently.

### **Types of file organization:**

Various methods have been introduced to Organize files. These particular methods have advantages and disadvantages on the basis of access or selection. Thus, it is all upon the programmer to decide the best suited file Organization method according to his requirements.

Some types of File Organizations are:



- ❖ Sequential File Organization
- ❖ Heap File Organization
- ❖ Hash File Organization
- ❖ B+ Tree File Organization
- ❖ Clustered File Organization

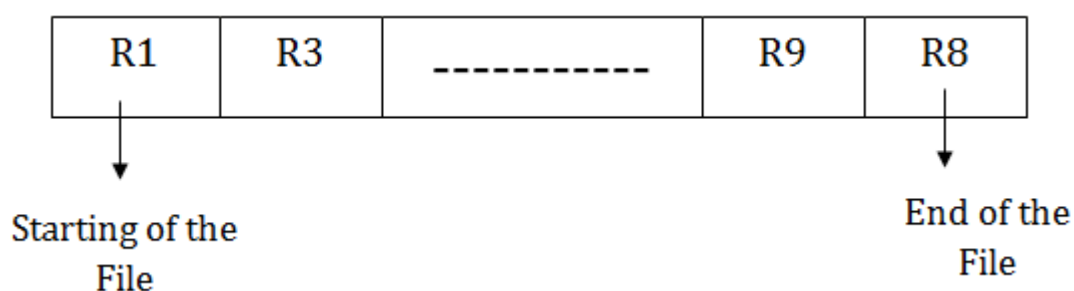
### Sequential File Organization

This method is the easiest method for file organization. In this method, files are stored sequentially. This method can be implemented in two ways:

- ❖ Pile File Method.
- ❖ Sorted File Method

#### 1. Pile File Method:

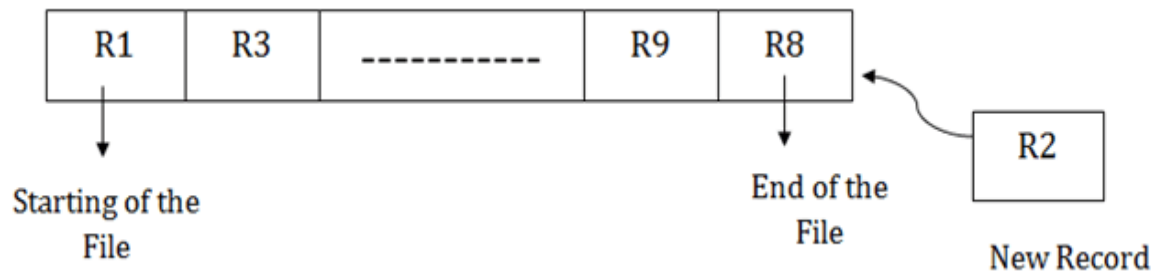
- It is a quite simple method. In this method, we store the record in a sequence, i.e., one after another. Here, the record will be inserted in the order in which they are inserted into tables.
- In case of updating or deleting of any record, the record will be searched in the memory blocks. When it is found, then it will be marked for deleting, and the new record is inserted.



#### Insertion of the new record:

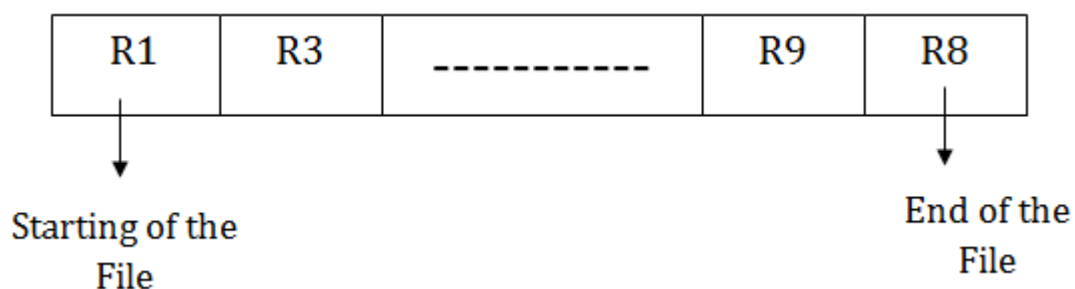
Suppose we have four records R1, R3 and so on up to R9 and R8 in a sequence. Hence, records are nothing but a row in the table. Suppose we want to insert a new record R2 in the sequence, then it will be placed at the end of the file. Here, records are nothing but a row in any table.





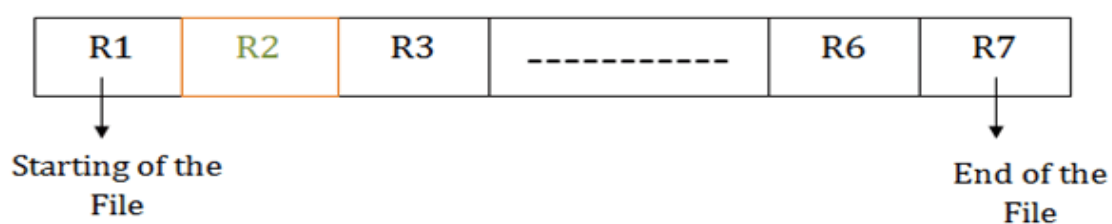
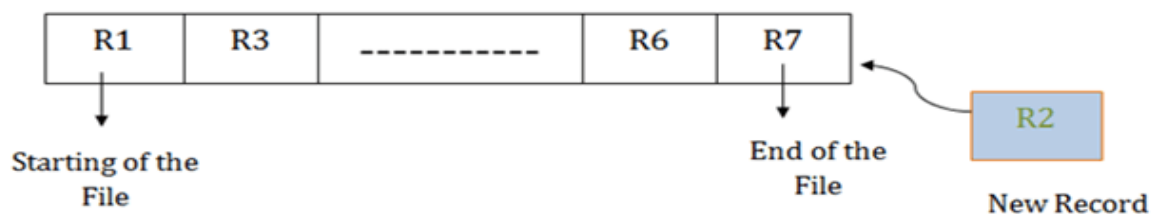
## 2. Sorted File Method:

- In this method, the new record is always inserted at the file's end, and then it will sort the sequence in ascending or descending order. Sorting of records is based on any primary key or any other key.
- In the case of modification of any record, it will update the record and then sort the file, and lastly, the updated record is placed in the right place.



## Insertion of the new record:

Suppose there is a preexisting sorted sequence of four records R1, R3 and so on upto R6 and R7. Suppose a new record R2 has to be inserted in the sequence, then it will be inserted at the end of the file, and then it will sort the sequence.



**Pros of sequential file organization**

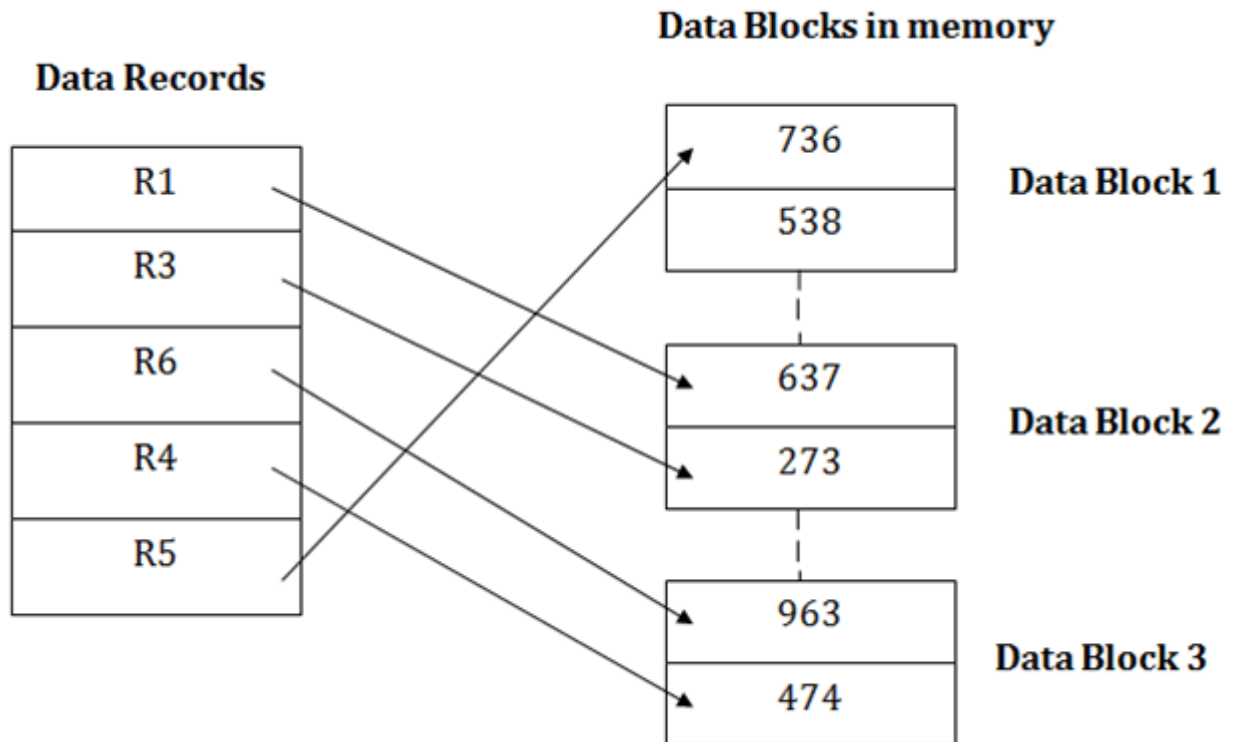
- It contains a fast and efficient method for the huge amount of data.
- In this method, files can be easily stored in cheaper storage mechanism like magnetic tapes.
- It is simple in design. It requires no much effort to store the data.
- This method is used when most of the records have to be accessed like grade calculation of a student, generating the salary slip, etc.
- This method is used for report generation or statistical calculations.

**Cons of sequential file organization**

- It will waste time as we cannot jump on a particular record that is required but we have to move sequentially which takes our time.
- Sorted file method takes more time and space for sorting the records.

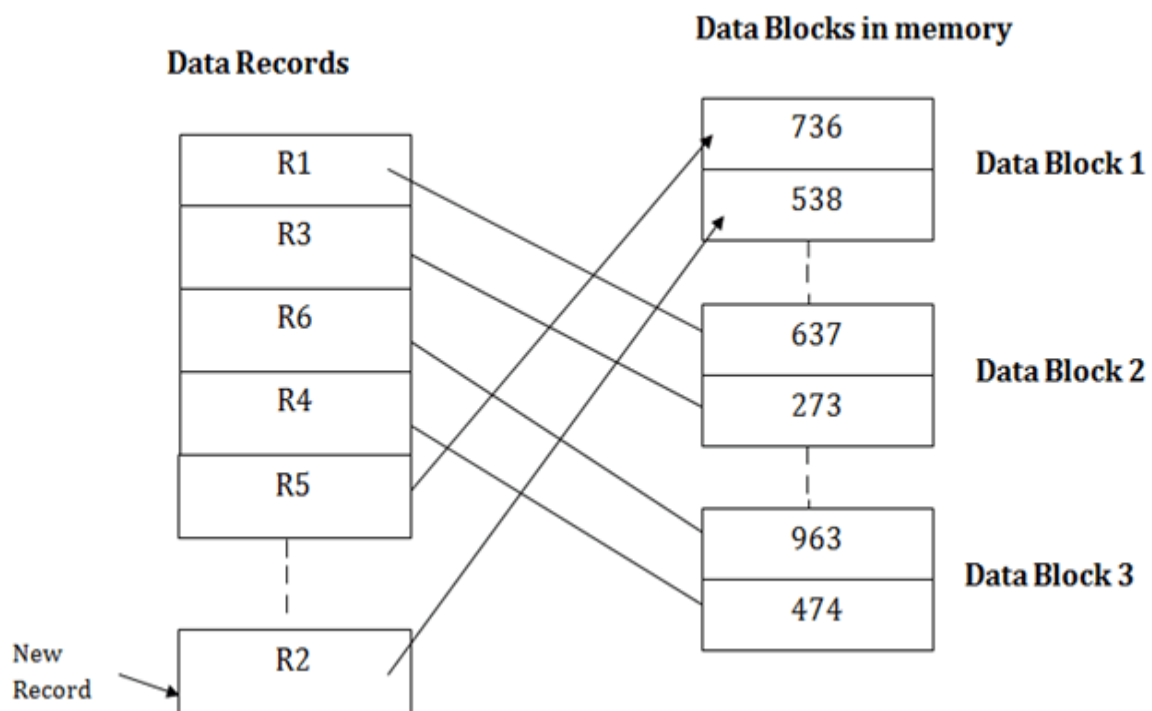
**Heap file organization**

- It is the simplest and most basic type of organization. It works with data blocks. In heap file organization, the records are inserted at the file's end. When the records are inserted, it doesn't require the sorting and ordering of records.
- When the data block is full, the new record is stored in some other block. This new data block need not to be the very next data block, but it can select any data block in the memory to store new records. The heap file is also known as an unordered file.
- In the file, every record has a unique id, and every page in a file is of the same size. It is the DBMS responsibility to store and manage the new records.



**Insertion of a new record**

Suppose we have five records R1, R3, R6, R4 and R5 in a heap and suppose we want to insert a new record R2 in a heap. If the data block 3 is full then it will be inserted in any of the database selected by the DBMS, let's say data block 1.



If we want to search, update or delete the data in heap file organization, then we need to traverse the data from starting of the file till we get the requested record.

If the database is very large then searching, updating or deleting of record will be time-consuming because there is no sorting or ordering of records. In the heap file organization, we need to check all the data until we get the requested record.

### **Pros of Heap file organization**

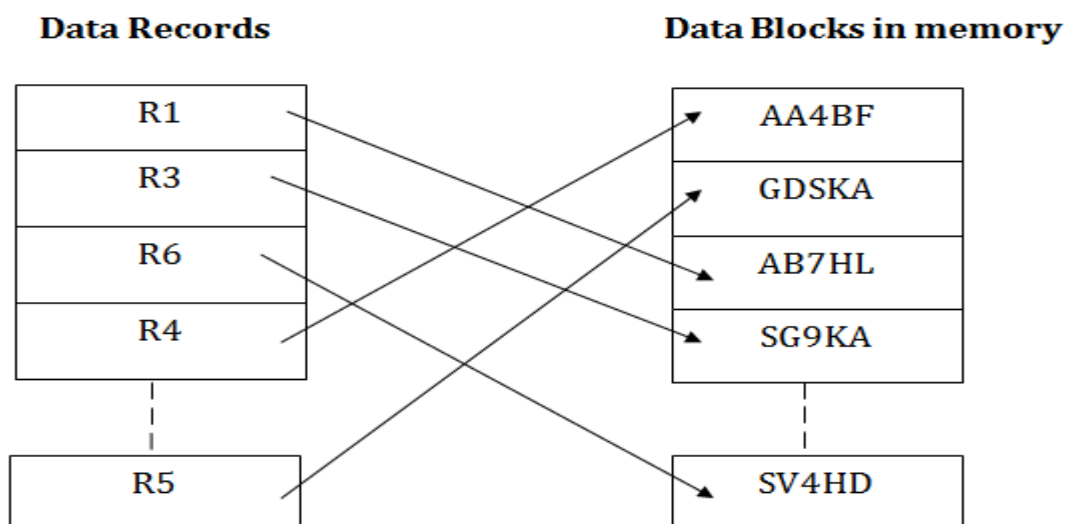
- It is a very good method of file organization for bulk insertion. If there is a large number of data which needs to load into the database at a time, then this method is best suited.
- In case of a small database, fetching and retrieving of records is faster than the sequential record.

### **Cons of Heap file organization**

- This method is inefficient for the large database because it takes time to search or modify the record.
- This method is inefficient for large databases.

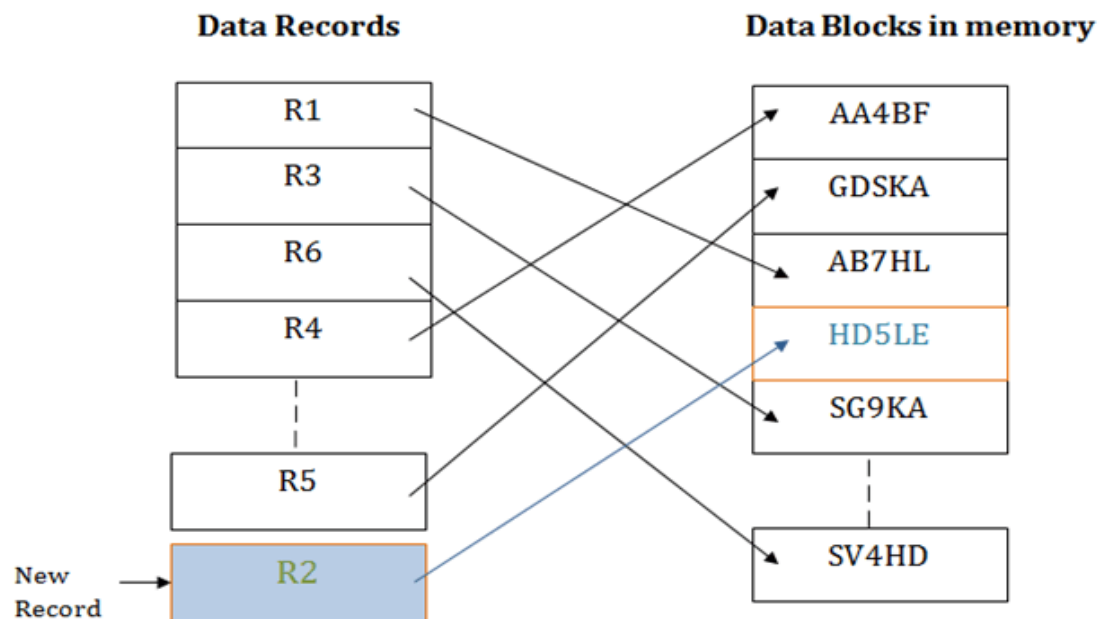
### **Hash File Organization**

Hash File Organization uses the computation of hash function on some fields of the records. The hash function's output determines the location of disk block where the records are to be placed.



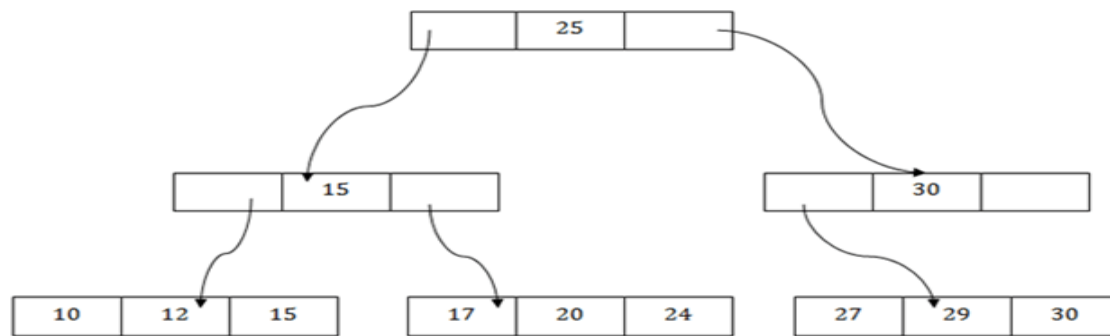
When a record has to be received using the hash key columns, then the address is generated, and the whole record is retrieved using that address. In the same way, when a new record has to be inserted, then the address is generated using the hash key and record is directly inserted. The same process is applied in the case of delete and update.

In this method, there is no effort for searching and sorting the entire file. In this method, each record will be stored randomly in the memory.



## B+ File Organization

- B+ tree file organization is the advanced method of an indexed sequential access method. It uses a tree-like structure to store records in File.
- It uses the same concept of key-index where the primary key is used to sort the records. For each primary key, the value of the index is generated and mapped with the record.
- The B+ tree is similar to a binary search tree (BST), but it can have more than two children. In this method, all the records are stored only at the leaf node. Intermediate nodes act as a pointer to the leaf nodes. They do not contain any records.



**The above B+ tree shows that:**

- There is one root node of the tree, i.e., 25.
- There is an intermediary layer with nodes. They do not store the actual record. They have only pointers to the leaf node.
- The nodes to the left of the root node contain the prior value of the root and nodes to the right contain next value of the root, i.e., 15 and 30 respectively.
- There is only one leaf node which has only values, i.e., 10, 12, 17, 20, 24, 27 and 29.
- Searching for any record is easier as all the leaf nodes are balanced.
- In this method, searching any record can be traversed through the single path and accessed easily.

**Pros of B+ tree file organization**

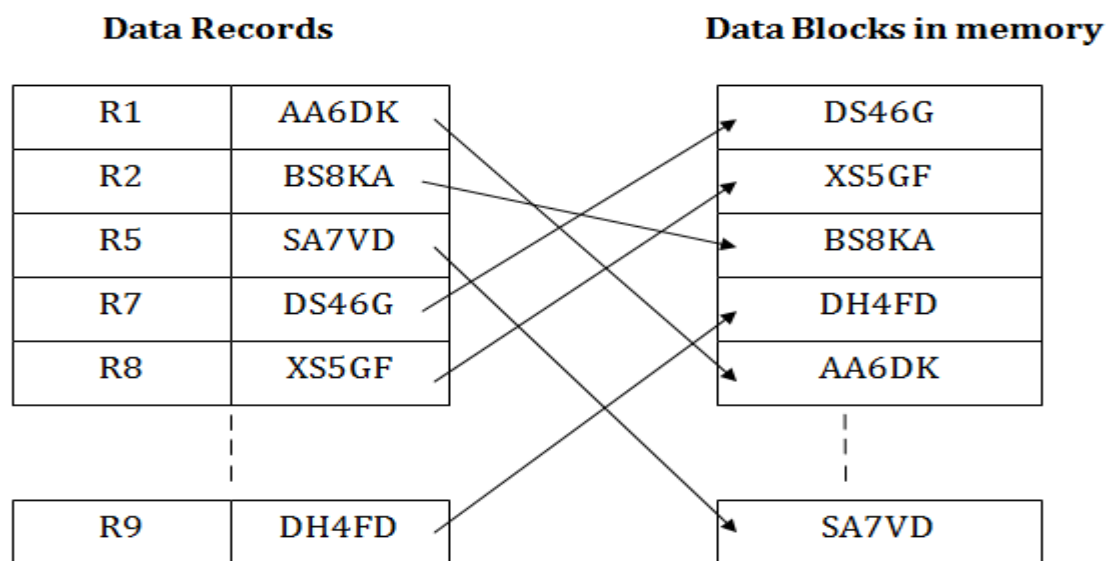
- In this method, searching becomes very easy as all the records are stored only in the leaf nodes and sorted the sequential linked list.
- Traversing through the tree structure is easier and faster.
- The size of the B+ tree has no restrictions, so the number of records can increase or decrease and the B+ tree structure can also grow or shrink.
- It is a balanced tree structure, and any insert/update/delete does not affect the performance of tree.

**Cons of B+ tree file organization**

- This method is inefficient for the static method.

**Indexed sequential access method (ISAM)**

ISAM method is an advanced sequential file organization. In this method, records are stored in the file using the primary key. An index value is generated for each primary key and mapped with the record. This index contains the address of the record in the file.



If any record has to be retrieved based on its index value, then the address of the data block is fetched and the record is retrieved from the memory.

**Pros of ISAM:**

- In this method, each record has the address of its data block, searching a record in a huge database is quick and easy.
- This method supports range retrieval and partial retrieval of records. Since the index is based on the primary key values, we can retrieve the data for the given range of value. In the same way, the partial value can also be easily searched, i.e., the student's name starting with 'JA' can be easily searched.

**Cons of ISAM**

- This method requires extra space in the disk to store the index value.

- When the new records are inserted, then these files have to be reconstructed to maintain the sequence.
- When the record is deleted, then the space used by it needs to be released. Otherwise, the performance of the database will slow down.

## Cluster file organization

- When the two or more records are stored in the same file, it is known as clusters. These files will have two or more tables in the same data block, and key attributes which are used to map these tables together are stored only once.
- This method reduces the cost of searching for various records in different files.
- The cluster file organization is used when there is a frequent need for joining the tables with the same condition. These joins will give only a few records from both tables. In the given example, we are retrieving the record for only particular departments. This method can't be used to retrieve the record for the entire department.


**EMPLOYEE**

EMP_ID	EMP_NAME	ADDRESS	DEP_ID
1	John	Delhi	14
2	Robert	Gujarat	12
3	David	Mumbai	15
4	Amelia	Meerut	11
5	Kristen	Noida	14
6	Jackson	Delhi	13
7	Amy	Bihar	10
8	Sonoo	UP	12

**DEPARTMENT**

DEP_ID	DEP_NAME
10	Math
11	English
12	Java
13	Physics
14	Civil
15	Chemistry

**Cluster Key**



DEP_ID	DEP_NAME	EMP_ID	EMP_NAME	ADDRESS
10	Math	7	Amy	Bihar
11	English	4	Amelia	Meerut
12	Java	2	Robert	Gujarat
12		8	Sonoo	UP
13	Physics	6	Jackson	Delhi
14	Civil	1	John	Delhi
14		5	Kristen	Noida
15	Chemistry	3	David	Mumbai



In this method, we can directly insert, update or delete any record. Data is sorted based on the key with which searching is done. Cluster key is a type of key with which joining of the table is performed.

### **Types of Cluster file organization:**

Cluster file organization is of two types:

#### **1. Indexed Clusters:**

In indexed cluster, records are grouped based on the cluster key and stored together. The above EMPLOYEE and DEPARTMENT relationship is an example of an indexed cluster. Here, all the records are grouped based on the cluster key- DEP\_ID and all the records are grouped.

#### **2. Hash Clusters:**

It is similar to the indexed cluster. In hash cluster, instead of storing the records based on the cluster key, we generate the value of the hash key for the cluster key and store the records with the same hash key value.

### **Pros of Cluster file organization**

- The cluster file organization is used when there is a frequent request for joining the tables with same joining condition.
- It provides the efficient result when there is a 1:M mapping between the tables.

### **Cons of Cluster file organization**

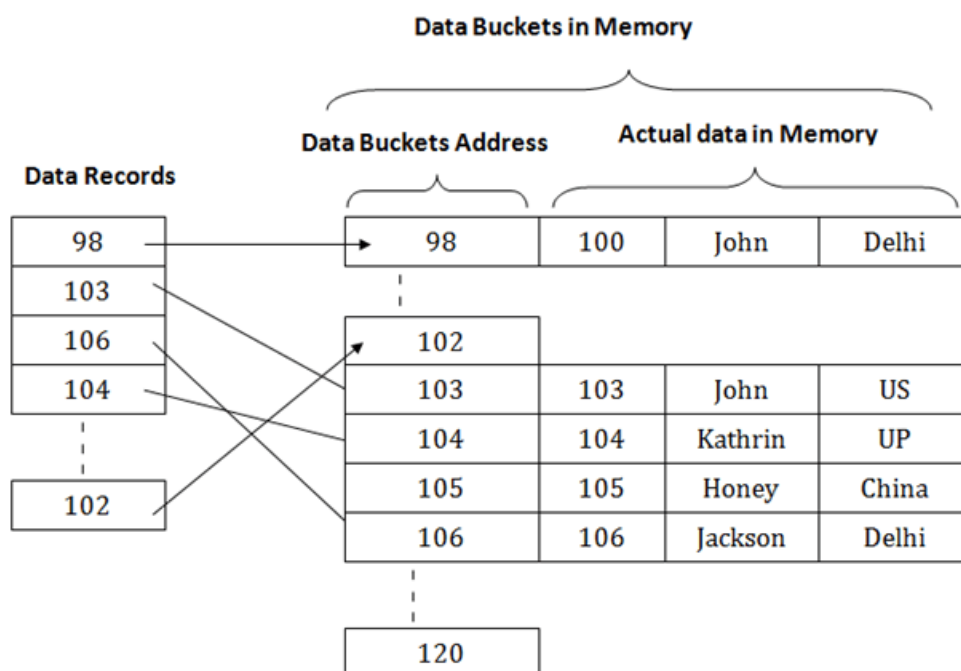
- This method has the low performance for the very large database.
- If there is any change in joining condition, then this method cannot use. If we change the condition of joining then traversing the file takes a lot of time.
- This method is not suitable for a table with a 1:1 condition.

## Hash Based Indexing

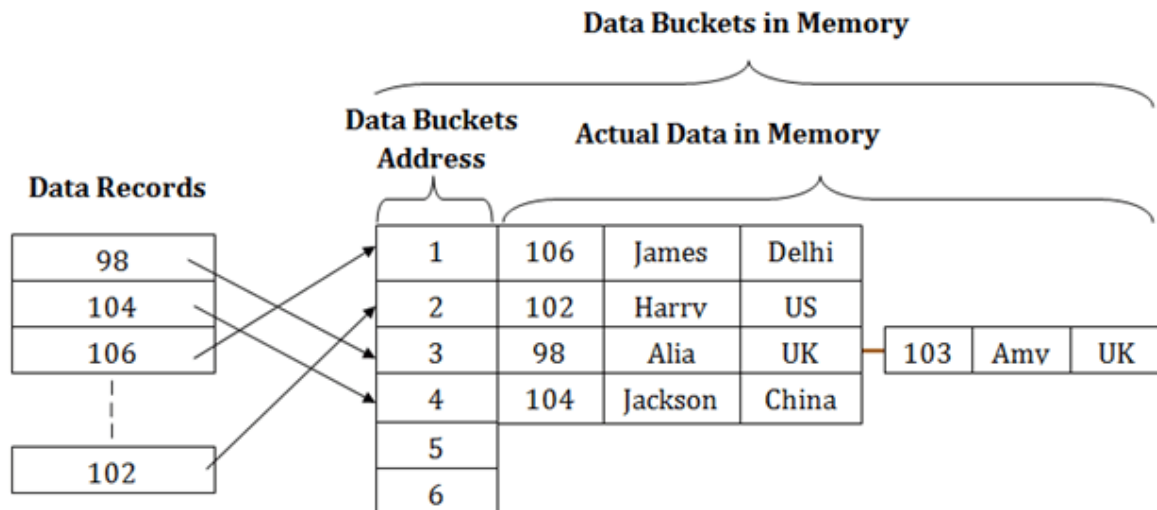
In a huge database structure, it is very inefficient to search all the index values and reach the desired data. Hashing technique is used to calculate the direct location of a data record on the disk without using index structure.

In this technique, data is stored at the data blocks whose address is generated by using the hashing function. The memory location where these records are stored is known as data bucket or data blocks.

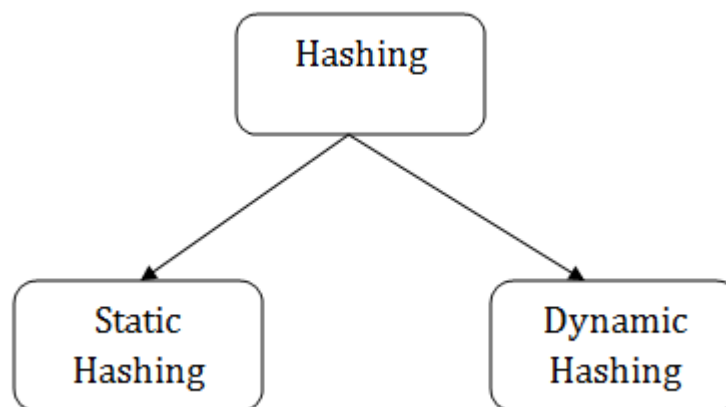
In this, a hash function can choose any of the column value to generate the address. Most of the time, the hash function uses the primary key to generate the address of the data block. A hash function is a simple mathematical function to any complex mathematical function. We can even consider the primary key itself as the address of the data block. That means each row whose address will be the same as a primary key stored in the data block.



The above diagram shows data block addresses same as primary key value. This hash function can also be a simple mathematical function like exponential, mod, cos, sin, etc. Suppose we have mod (5) hash function to determine the address of the data block. In this case, it applies mod (5) hash function on the primary keys and generates 3, 3, 1, 4 and 2 respectively, and records are stored in those data block addresses.



### Types of Hashing:

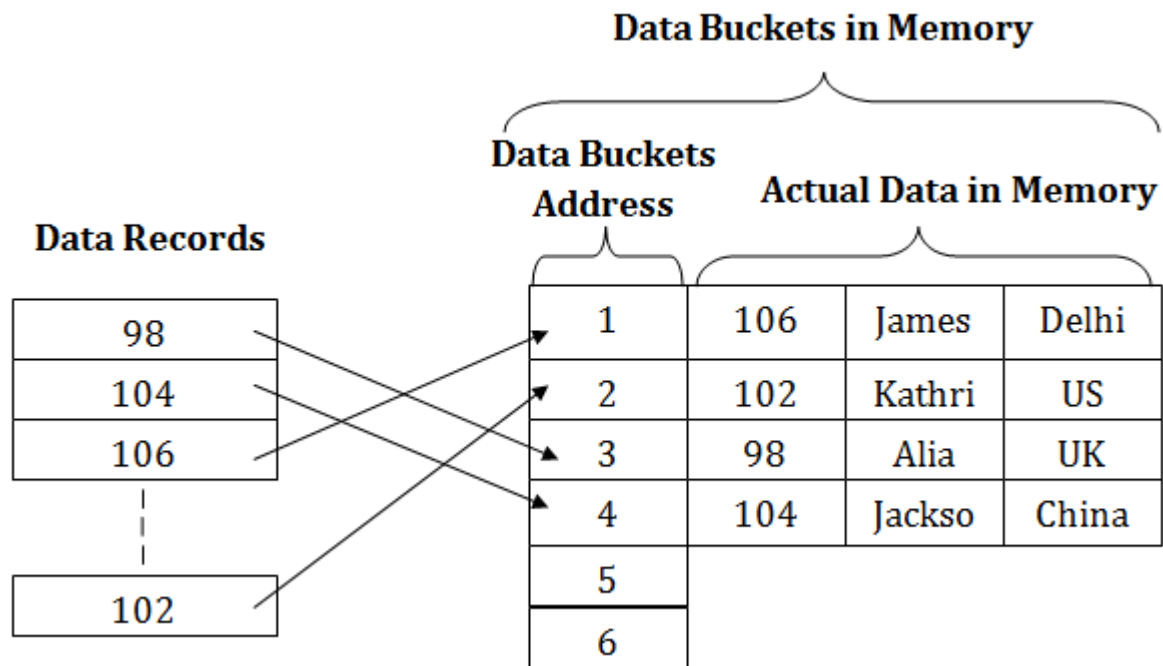


- ❖ Static Hashing
- ❖ Dynamic Hashing

### Static Hashing

In static hashing, the resultant data bucket address will always be the same. That means if we generate an address for EMP\_ID = 103 using the hash function mod (5) then it will always result in same bucket address 3. Here, there will be no change in the bucket address.

Hence in this static hashing, the number of data buckets in memory remains constant throughout. In this example, we will have five data buckets in the memory used to store the data.



### Operations of Static Hashing

#### Searching a record

When a record needs to be searched, then the same hash function retrieves the address of the bucket where the data is stored.

#### Insert a Record

When a new record is inserted into the table, then we will generate an address for a new record based on the hash key and record is stored in that location.

#### Delete a Record

To delete a record, we will first fetch the record which is supposed to be deleted. Then we will delete the records for that address in memory.

#### Update a Record

To update a record, we will first search it using a hash function, and then the data record is updated.

If we want to insert some new record into the file but the address of a data bucket generated by the hash function is not empty, or data already exists in that address. This

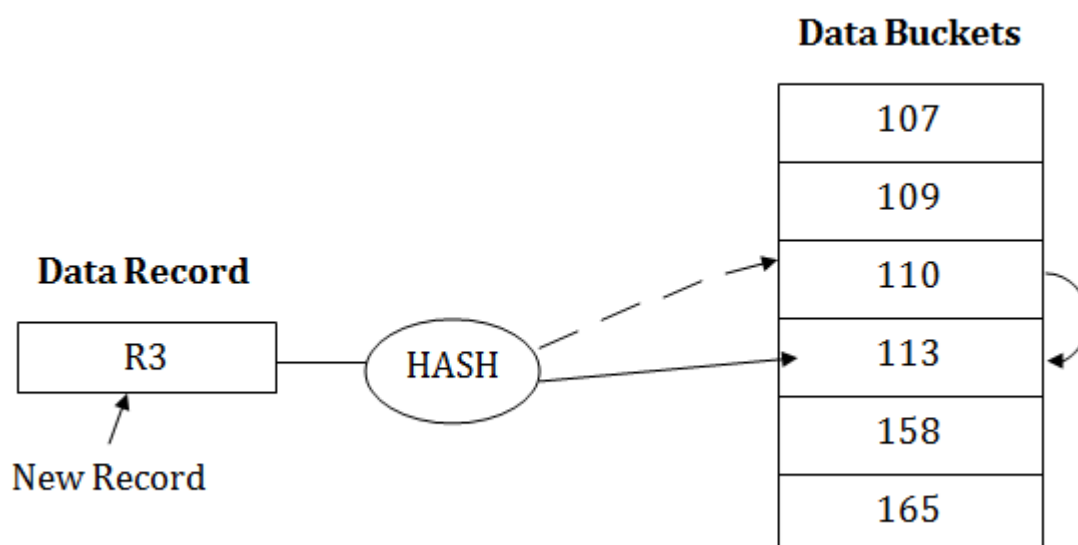
situation in the static hashing is known as **bucket overflow**. This is a critical situation in this method.

To overcome this situation, there are various methods. Some commonly used methods are as follows:

### 1. Open Hashing

When a hash function generates an address at which data is already stored, then the next bucket will be allocated to it. This mechanism is called as **Linear Probing**.

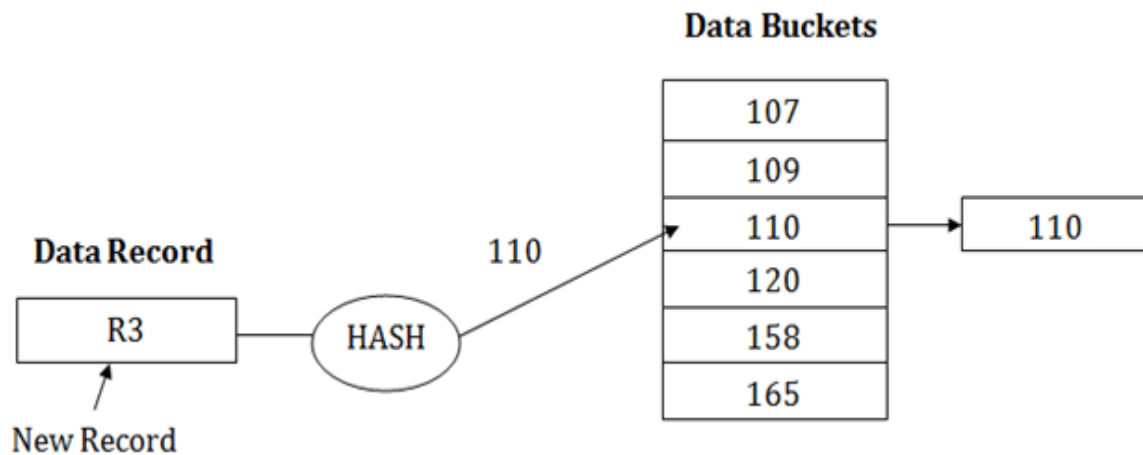
**For example:** suppose R3 is a new address which needs to be inserted, the hash function generates address as 112 for R3. But the generated address is already full. So the system searches next available data bucket, 113 and assigns R3 to it.



### 2. Close Hashing

When buckets are full, then a new data bucket is allocated for the same hash result and is linked after the previous one. This mechanism is known as **Overflow chaining**.

**For example:** Suppose R3 is a new address which needs to be inserted into the table, the hash function generates address as 110 for it. But this bucket is full to store the new data. In this case, a new bucket is inserted at the end of 110 buckets and is linked to it.



### Dynamic Hashing

- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increase or decrease. This method is also known as Extendable hashing method.
- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

### How to search a key

- First, calculate the hash address of the key.
- Check how many bits are used in the directory, and these bits are called as i.
- Take the least significant i bits of the hash address. This gives an index of the directory.
- Now using the index, go to the directory and find bucket address where the record might be.

### How to insert a new record

- Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.

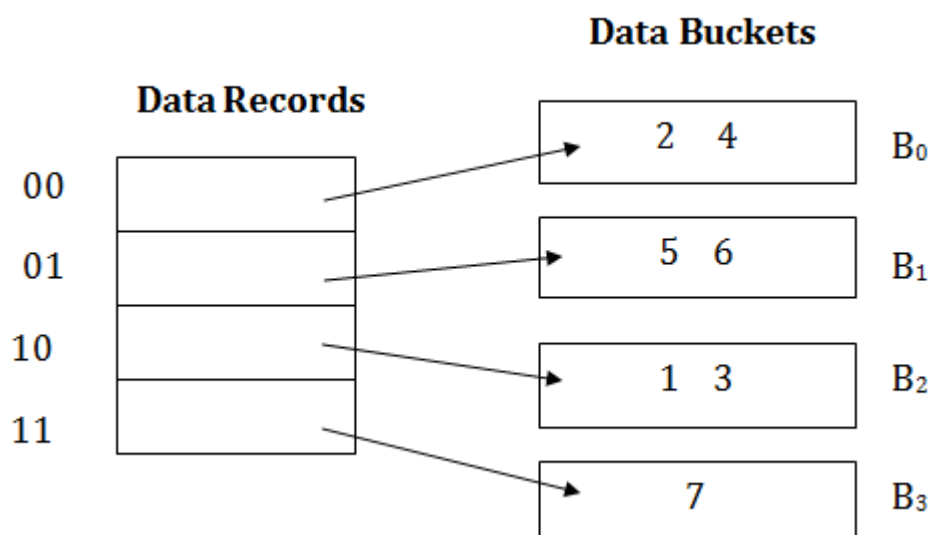
- If there is still space in that bucket, then place the record in it.
- If the bucket is full, then we will split the bucket and redistribute the records.

**For example:**

Consider the following grouping of keys into buckets, depending on the prefix of their hash address:

Key	Hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111

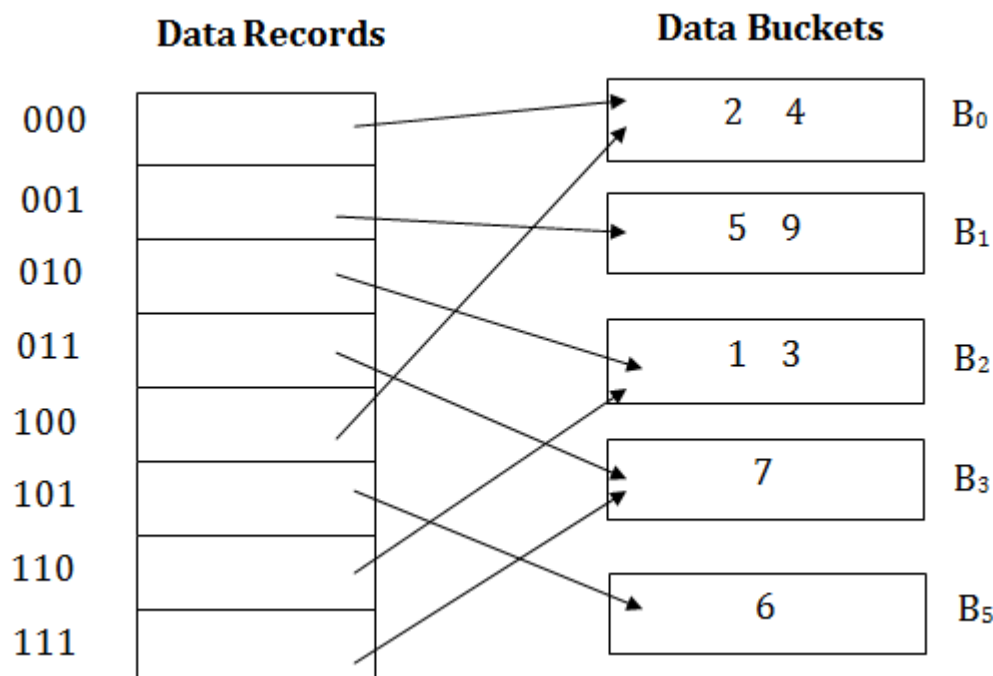
The last two bits of 2 and 4 are 00. So it will go into bucket B<sub>0</sub>. The last two bits of 5 and 6 are 01, so it will go into bucket B<sub>1</sub>. The last two bits of 1 and 3 are 10, so it will go into bucket B<sub>2</sub>. The last two bits of 7 are 11, so it will go into bucket B<sub>3</sub>.



**Insert key 9 with hash address 10001 into the above structure:**

- Since key 9 has hash address 10001, it must go into the first bucket. But bucket B<sub>1</sub> is full, so it will get split.

- The splitting will separate 5, 9 from 6 since last three bits of 5, 9 are 001, so it will go into bucket B1, and the last three bits of 6 are 101, so it will go into bucket B5.
- Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00.
- Keys 1 and 3 are still in B2. The record in B2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.
- Key 7 are still in B3. The record in B3 pointed by the 111 and 011 entry because last two bits of both the entry are 11.



## Advantages of dynamic hashing

- In this method, the performance does not decrease as the data grows in the system. It simply increases the size of memory to accommodate the data.
- In this method, memory is well utilized as it grows and shrinks with the data. There will not be any unused memory lying.
- This method is good for the dynamic database where data grows and shrinks frequently.



**Disadvantages of dynamic hashing**

- In this method, if the data size increases then the bucket size is also increased. These addresses of data will be maintained in the bucket address table. This is because the data address will keep changing as buckets grow and shrink. If there is a huge increase in data, maintaining the bucket address table becomes tedious.
- In this case, the bucket overflow situation will also occur. But it might take little time to reach this situation than static hashing.