SUBJECT (CODE): DBMS (KCS-501) COURSE: B.Tech (CSE)

UNIT – IV (TRANSACTION SYSTEM)

A transaction can be defined as a group of tasks. A single task is the minimum processing unit which cannot be divided further. A transaction is a very small unit of a program and it may contain several low-level tasks.

A set of logically related operations is known as a transaction. The main operations of a transaction are:
Read (A): Read operations Read (A) or R (A) reads the value of A from the database and stores it in a buffer in the main memory.
Write (A): Write operation Write (A) or W (A) writes the value back to the database from the buffer.
(Note: It doesn't always need to write it to a database back it just writes the changes to buffer this is the reason where dirty read comes into the picture)

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

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

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

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

But it may also be possible that the transaction may fail after executing some of its operations. The failure can be because of hardware, software or power, etc. For example, if the debit transaction discussed above fails after executing operation 2, the value of A will remain 5000 in the database which is not acceptable by the bank. To avoid this, Database has two important operations:
Commit: After all instructions of a transaction are successfully executed, the changes made by a transaction are made permanent in the database.
Rollback: If a transaction is not able to execute all operations successfully, all the changes made by a transaction are undone.

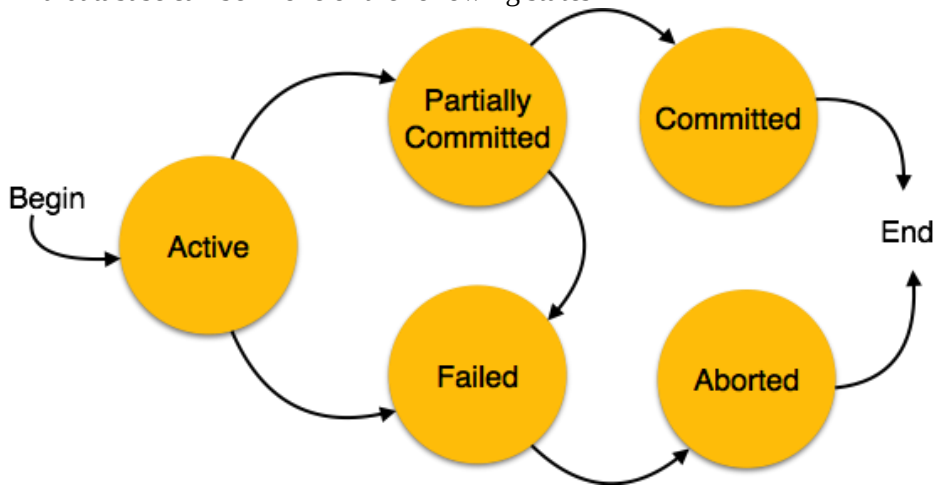
PROPERTY OF A TRANSACTION

A transaction in a database system must maintain Atomicity, Consistency, Isolation, and Durability − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

- Atomicity − this property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- Consistency − the database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- Durability − the database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- Isolation − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

States of Transactions

A transaction in a database can be in one of the following states −



- **Active** − In this state, the transaction is being executed. This is the initial state of every transaction.
- **Partially** Committed − When a transaction executes its final operation, it is said to be in a partially committed state.
- **Failed** − A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.
- **Aborted** − If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts −
    - Re-start the transaction
    - Kill the transaction
- **Committed** − If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

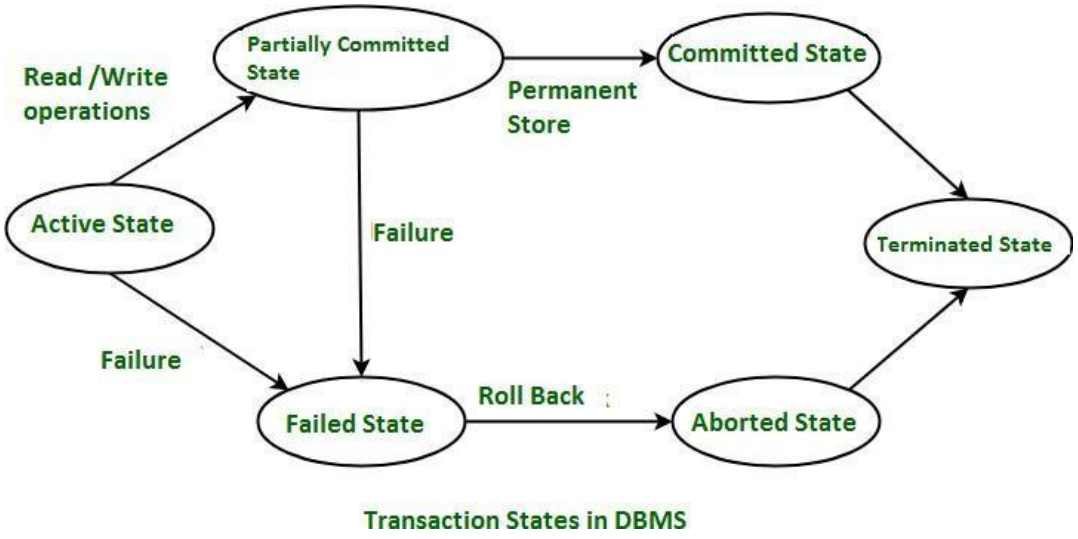
Uses of Transaction Management:

- The DBMS is used to schedule the access of data concurrently. It means that the user can access multiple data from the database without being interfered with each other. Transactions are used to manage concurrency.

- It is also used to satisfy ACID properties.
  It is used to solve Read/Write Conflict.

- It is used to implement Recoverability, Serializability, and Cascading.
  Transaction Management is also used for Concurrency Control Protocols and Locking of data.

Transaction States :
Transactions can be implemented using SQL queries and Server. In the below-given diagram, you can see how transaction states works.



Transaction States

Disadvantage of using a Transaction :

- It may be difficult to change the information within the transaction database by end-users.
- We need to always roll back and start from the beginning rather than continue from the previous state.

Testing of Serializability

In a database management system (DBMS), a serializable schedule is a sequence of database actions (read and write operations) that does not violate the serializability property. This property ensures that each transaction appears to execute atomically and is isolated from other transactions' effects. In order for serializability of schedules in DBMS, it must be equivalent to some serial schedule of the same transactions. In other words, the schedule must produce the same results as if the transactions were executed one at a time in some order.

There are several different algorithms that can be used to check for a serializable schedule in DBMS. One such algorithm is the conflict serializability algorithm, which checks for potential conflicts between transactions. A conflict occurs when two transactions try to access the same data item in conflicting ways (e.g., one transaction tries to read a data item while another transaction is writing to it). If there are no conflicts, then the schedule is guaranteed to be serializable. However, if there are conflicts, then the schedule may or may not be serializable.

Another algorithm that can be used to check for serializability is the view serializability in the DBMS algorithm, which checks for potential mutual dependencies between transactions. A mutual dependency exists when two transactions depend on each other for their respective outputs to be correct. If there are no mutual dependencies, then the schedule is guaranteed to be serializable. However, if there are mutual dependencies, then the schedule may or may not be serializable.

1. Conflict Serializability

Conflict serializability is a type of serializability in which conflicting operations on the same data items are executed in an order that preserves database consistency. Each transaction is assigned a unique number, and the operations within each transaction are executed in order based on that number. This ensures that no two conflicting operations are executed concurrently. For example, consider a database with two tables: Customers and Orders. A customer can have multiple orders, but each order can only be associated with one customer.

Key conditions for conflict serializability.

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

2. View Serializability

View serializability is a type of serializability in which each transaction produces results that are equivalent to some well-defined sequential execution of all transactions in the system. Unlike conflict serializability, which focuses on preventing inconsistencies within the database, view serializability in DBMS focuses on providing users with consistent views of the database.

In order to better understand view serialization in a database management system, it is important to consider schedules S1 and S2. These schedules are created with two transactions in mind, T1 and T2. In order for these schedules to be viewed as equivalent, the following three conditions must be met.

The first condition is that both schedules must have the same set of committed transactions. This simply means that both schedules S1 and S2, cannot have different committed transactions. If one schedule has a committed transaction that the other does not, then the schedules are not viewed as equivalent.

The second condition is that both schedules cannot have different numbers of read/write operations on the same data item. In other words, if schedule S1 has two write operations on data item A while schedule S2 only has one write operation on data item A, the schedules are not viewed as equivalent. The number of read operations can differ between the schedules as long as the number of write operations is equal.

The last and final condition is that both schedules cannot have conflicting orders of execution for the same data items. For example, suppose in schedule S1 transaction T1 writes to data item A before transaction T2 does while in schedule S2 transaction T2 writes to data item A before transaction T1 does. In that case, the schedules are not viewed as equivalent. This conflicts with condition 2, which states that both schedules must have the same number of write operations on each data item. However, if all three Benefits of Serializability in DBMS

Now that we understand the serializability in DBMS let's look at some benefits of Serializability in DBMS.

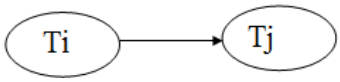
1. Predictable Executions: Since all threads are executed one at a time, there are no surprises. All variables are updated as expected, and no data is lost or corrupted.
2. Easier to Reason about & Debug: As each thread is executed alone, it is easier to reason about what each thread is doing and why. This can make debugging much easier since you don't have to worry about concurrency issues.
3. Reduced Costs: Serialization in DBMS can help reduce hardware costs by allowing fewer resources to be used for a given computation (e.g., only one CPU instead of two). Additionally, it can help reduce software development costs by making it easier to reason about code and reducing the need for extensive testing with multiple threads running concurrently.
4. Increased Performance: In some cases, serializable executions can perform better than their non-serializable counterparts since they allow the developer to optimize their code for performance.

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

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

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

Precedence graph for Schedule S



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

For example:

Time

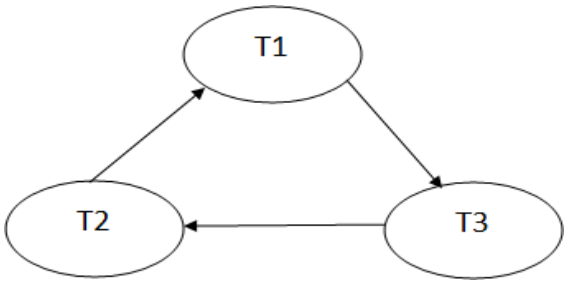
T1	T2	T3
Read(A)		
	Read(B)	
A:= f_1(A)		
		Read(C)
	B:= f_2(B)	
	Write(B)	
		C:= f_3(C)
		Write(C)
Write(A)		
		Read(B)
	Read(A)	
	A:= f_4(A)	
Read(C)		
	Write(A)	
C:= f_5(C)		
Write(C)		
		B:= f_6(B)
		Write(B)

Schedule S1

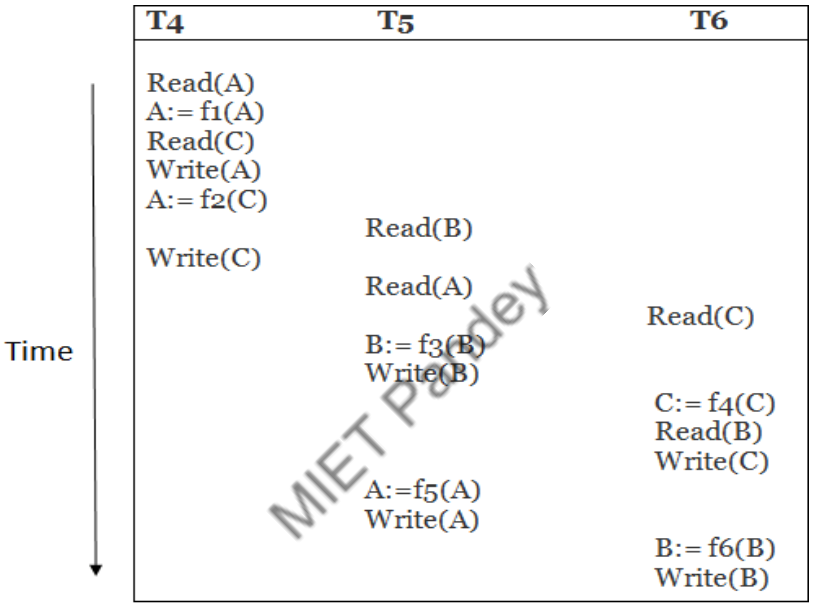
Explanation:

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

Precedence graph for schedule S1:



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

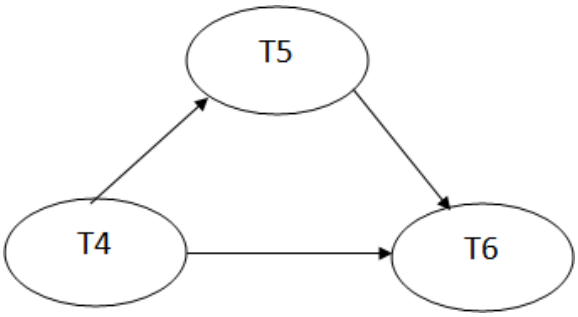


Schedule S2

Explanation:

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

Precedence graph for schedule S2:



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

Irrecoverable schedules

If a transaction does a dirty read operation from an uncommitted transaction and commits before the transaction from where it has read the value, then such a schedule is called an irrecoverable schedule.

Example

Let us consider a two transaction schedules as shown below −

T1	T2
Read(A)	
Write(A)	
-	Read(A) ///Dirty Read
-	Write(A)
-	Commit
-	
Rollback	

The above schedule is a irrecoverable because of the reasons mentioned below −

- The transaction T2 which is performing a dirty read operation on A.
- The transaction T2 is also committed before the completion of transaction T2.
- The transaction T1 fails later and there are rollbacks.
- The transaction T2 reads an incorrect value.
- Finally, the transaction T2 cannot recover because it is already committed.

Recoverable Schedules

If any transaction that performs a dirty read operation from an uncommitted transaction and also its committed operation becomes delayed till the uncommitted transaction is either committed or rollback such type of schedules is called as Recoverable Schedules.

Example

Let us consider two transaction schedules as given below −

T1	T2

T1	T2
Read(A)	
Write(A)	
-	Read(A) ///Dirty Read
-	Write(A)
-	
-	
Commit	
	Commit // delayed

The above schedule is a recoverable schedule because of the reasons mentioned below −

- The transaction T2 performs dirty read operation on A.
- The commit operation of transaction T2 is delayed until transaction T1 commits or rollback.
- Transaction commits later.
- In the above schedule transaction T2 is now allowed to commit whereas T1 is not yet committed.
- In this case transaction T1 is failed, and transaction T2 still has a chance to recover by rollback.

RECOVERY FROM FAILURE

DBMS is a highly complex system with hundreds of transactions being executed every second. The durability and robustness of a DBMS depends on its complex architecture and its underlying hardware and system software. If it fails or crashes amid transactions, it is expected that the system would follow some sort of algorithm or techniques to recover lost data.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

- 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

- 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.

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_n, Start>

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

<T_n, X, V_1, V_2>

It reads T_n has changed the value of X, from V_1 to V_2.

- When the transaction finishes, it logs −

<T_n, 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.

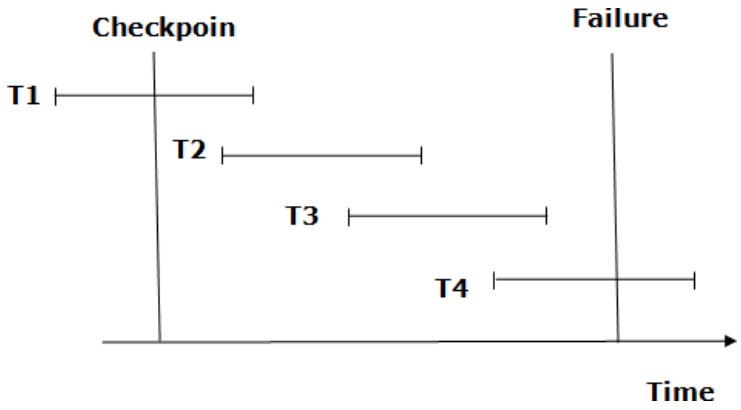
CHECKPOINTS

The DBMS checkpoint is a mechanism of compressing the transaction log file by transferring the old transactions to permanent storage. The checkpoint marks the position till where the consistency of the transactions is maintained. During the execution of the transactions, the curser passes through the marked checkpoint.

- The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.
- The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked, and the transaction is executed then using the steps of the transaction, the log files will be created.
- When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.
- The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

Recovery using Checkpoint

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



- The recovery system reads log files from the end to start. It reads log files from T4 to T1.
- Recovery system maintains two lists, a redo-list, and an undo-list.
- The transaction is put into redo state if the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>. In the redo-list and their previous list, all the transactions are removed and then redone before saving their logs.
- For example: In the log file, transaction T2 and T3 will have <Tn, Start> and <Tn, Commit>. The T1 transaction will have only <Tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts T1, T2 and T3 transaction into redo list.
- The transaction is put into undo state if the recovery system sees a log with <Tn, Start> but no commit or abort log found. In the undo-list, all the transactions are undone, and their logs are removed.
- For example: Transaction T4 will have <Tn, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed amid.

Advantages of using Checkpoints:

- It speeds up data recovery process.
- Most of the dbms products automatically checkpoints themselves.
- Checkpoint records in log file is used to prevent unnecessary redo operations.
- Since dirty pages are flushed out continuously in the background, it has very low overhead and can be done frequently.

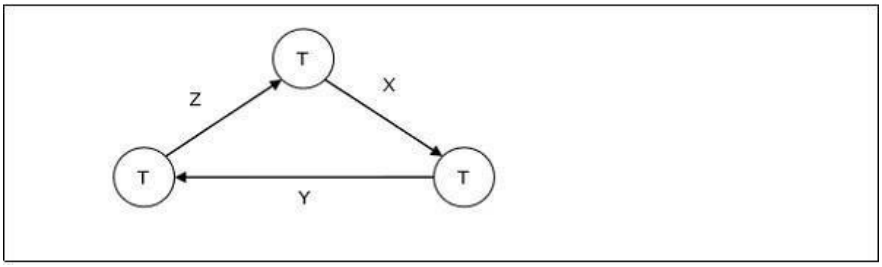
Real-Time Applications of Checkpoints:

- Whenever an application is tested in real-time environment that may have modified the database, it is verified and validated using checkpoints.
- Checkpoints are used to create backups and recovery prior to applying any updates in the database.
- The recovery system is used to return the database to the checkpoint state.

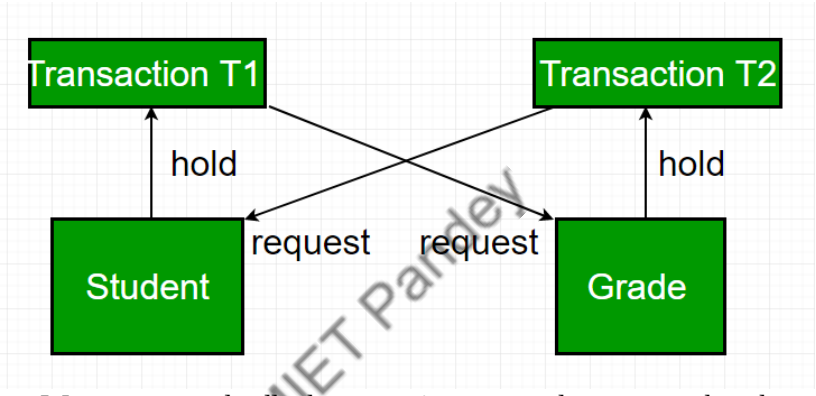
DEADLOCK

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.



Example – let us understand the concept of Deadlock with an example :
Suppose, Transaction T1 holds a lock on some rows in the Students table and needs to update some rows in the Grades table. Simultaneously, Transaction T2 holds locks on those very rows (Which T1 needs to update) in the Grades table but needs to update the rows in the Student table held by Transaction T1.
Now, the main problem arises. Transaction T1 will wait for transaction T2 to give up the lock, and similarly, transaction T2 will wait for transaction T1 to give up the lock. As a consequence, All activity comes to a halt and remains at a standstill forever unless the DBMS detects the deadlock and aborts one of the transactions.



Deadlock prevention: We can use a deadlock-prevention protocol to ensure that the system will never enter a deadlock state.
Deadlock detection: In this case, we can allow the system to enter a deadlock state, and then try to recover using a deadlock detection and deadlock recovery scheme.
Both the above methods may result in transaction rollback. Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise detection and recovery are more efficient.
Deadlock Prevention
We can prevent deadlocks by giving each transaction a priority and ensuring that lower priority transactions are not allowed to wait for higher priority transactions (or vice versa). One way to assign priorities is to give each transaction a timestamp when it starts up. The lower the timestamp, the higher the transaction priority that is, the oldest transaction has the highest priority.

If a transaction Ti requests a lock and transaction Tj holds a conflicting lock, the lock manager can use one of the following two policies:

Wait-die

If Ti has higher priority, it is allowed to wait; otherwise it is aborted. It means when transaction Ti requests a data item currently held by Tj, Ti is allowed to wait only if it has a timestamp smaller than that of T1 (that is Ti is older than Tj). Otherwise Ti is rolled back (dies).

Example

Suppose that transactions T22, T23 and T24 have timestamps 5, 10 and 15 respectively. If T22 requests a data item held by T23 then T22 will wait. If T24 requests a data item held by T23 then T24 will be rolled back.

T22 will Wait T24 will rollback

T22 ——————————-> T23 <——————————- T24

(5) (10) (15)

The wait-die scheme is non-preemptive scheme because only a transaction requesting a lock can be aborted. As a transaction grows older (and its priority increases), it tends to wait for more and more younger transactions.

Wound-wait

If Ti has higher priority, abort Tj otherwise Ti waits. It means when transaction Ti requests a data item currently held by Tj, Ti is allowed to wait only if it has a timestamp larger than that of Tj (that is Ti is younger than Tj). Otherwise Tj is rolled back (Tj is wounded by Ti).

Example:

Returning to our previous example, with transactions T22,T23 and T24, ifT22 requests a data item held by T23 then the data item will be preempted from T23 and T23 will be rolled back. IfT24 requests a data item held by T23, and then T24 will wait.

T22 access T23 will be

Data item rollback

T22 ——————————> T23

(5) (10)

Data item with T23 wait for T23

T23 <—————————— T24

(10) (15)

This scheme is based on a preemptive technique and is a counterpart to the wait-die scheme. In the wait-die scheme, lower priority transactions can never wait for higher priority transactions. In the wound-wait scheme, higher priority transactions never wait for lower priority transactions. In either case no deadlock cycle can develop.

When a transaction is aborted and restarted, it should be given the same timestamp that it had originally. Reissuing timestamps in this way ensures that each transaction will eventually become the oldest transaction, and thus the one with the highest priority, and will get the locks that it requires.

DISTRIBUTED DATABASE

A distributed database is a database that consists of two or more files located in different sites either on the same network or on entirely different networks. Portions of the database are stored in multiple physical locations and processing is distributed among multiple database nodes.

Advantages of Distributed Databases

Following are the advantages of distributed databases over centralized databases.

Modular Development − If the system needs to be expanded to new locations or new units, in centralized database systems, the action requires substantial efforts and disruption in the existing functioning. However, in distributed databases, the work simply requires adding new computers and local data to the new site and finally connecting them to the distributed system, with no interruption in current functions.

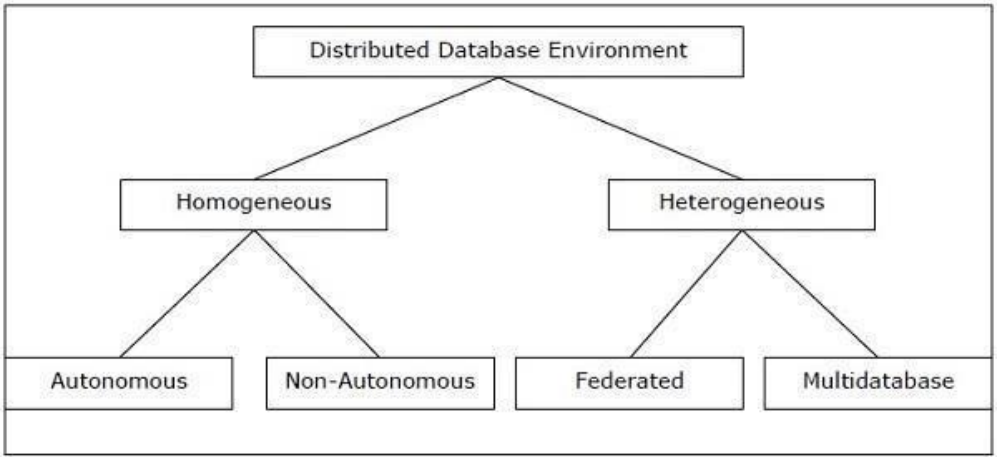
More Reliable − In case of database failures, the total system of centralized databases comes to a halt. However, in distributed systems, when a component fails, the functioning of the system continues may be at a reduced performance. Hence DDBMS is more reliable.

Better Response − If data is distributed in an efficient manner, then user requests can be met from local data itself, thus providing faster response. On the other hand, in centralized systems, all queries have to pass through the central computer for processing, which increases the response time.

Lower Communication Cost − In distributed database systems, if data is located locally where it is mostly used, then the communication costs for data manipulation can be minimized. This is not feasible in centralized systems.

Types of Distributed Databases

Distributed databases can be broadly classified into homogeneous and heterogeneous distributed database environments, each with further sub-divisions, as shown in the following illustration.



Homogeneous Distributed Databases

In a homogeneous distributed database, all the sites use identical DBMS and operating systems. Its properties are −

- The sites use very similar software.
- The sites use identical DBMS or DBMS from the same vendor.
- Each site is aware of all other sites and cooperates with other sites to process user requests.
- The database is accessed through a single interface as if it is a single database.

Types of Homogeneous Distributed Database

There are two types of homogeneous distributed database −

- ***Autonomous*** − Each database is independent that functions on its own. They are integrated by a controlling application and use message passing to share data updates.
- ***Non-autonomous*** − Data is distributed across the homogeneous nodes and a central or master DBMS co-ordinates data updates across the sites.

Heterogeneous Distributed Databases

In a heterogeneous distributed database, different sites have different operating systems, DBMS products and data models. Its properties are −

- Different sites use dissimilar schemas and software.
- The system may be composed of a variety of DBMSs like relational, network, hierarchical or object oriented.
- Query processing is complex due to dissimilar schemas.
- Transaction processing is complex due to dissimilar software.
- A site may not be aware of other sites and so there is limited co-operation in processing user requests.

Types of Heterogeneous Distributed Databases

- Federated − The heterogeneous database systems are independent in nature and integrated together so that they function as a single database system.
- Un-federated − The database systems employ a central coordinating module through which the databases are accessed.

CONCURRENCY CONTROL

A transaction is a single reasonable unit of work that can retrieve or may change the data of a database. Executing each transaction individually increases the waiting time for the other transactions and the overall execution also gets delayed. Hence, to increase the throughput and to reduce the waiting time, transactions are executed concurrently.
Example: Suppose, between two railway stations, A and B, 5 trains have to travel, if all the trains are set in a row and only one train is allowed to move from station A to B and others have to wait for the first train to reach its destination then it will take a lot of time for all the trains to travel from station A to B. To reduce time all the trains should be allowed to move concurrently from station A to B ensuring no risk of collision between them.
When several transactions execute simultaneously, then there is a risk of violation of the data integrity of several databases. Concurrency Control in DBMS is a procedure of managing simultaneous transactions ensuring their atomicity, isolation, consistency and serializability.

Concurrency control techniques
The concurrency control techniques are as follows −

Locking
Lock guaranties exclusive use of data items to a current transaction. It first accesses the data items by acquiring a lock, after completion of the transaction it releases the lock.
Types of Locks

The types of locks are as follows −

- Shared Lock [Transaction can read only the data item values]
- Exclusive Lock [Used for both read and write data item values]

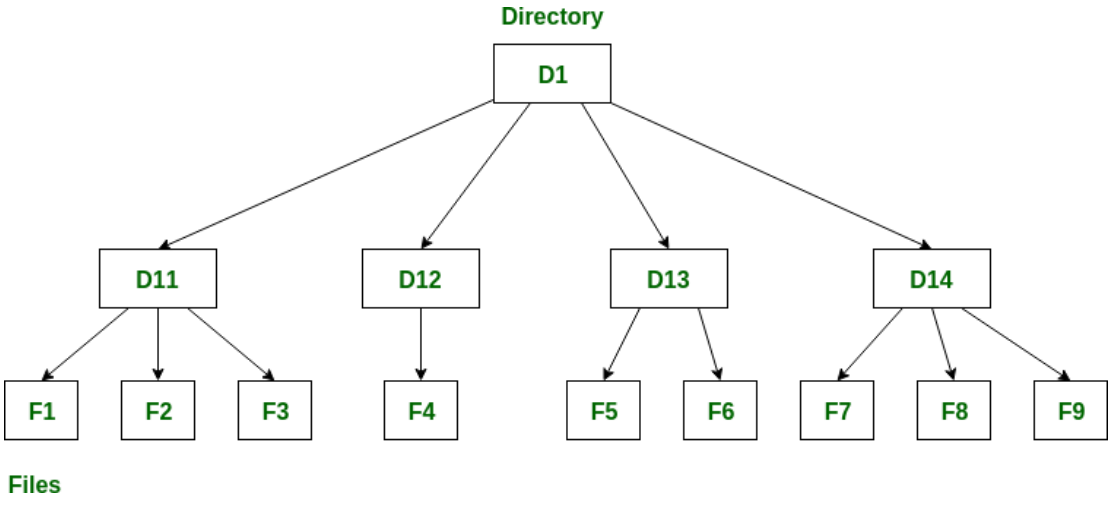
Time Stamping
Time stamp is a unique identifier created by DBMS that indicates relative starting time of a transaction. Whatever transaction we are doing it stores the starting time of the transaction and denotes a specific time.

This can be generated using a system clock or logical counter. This can be started whenever a transaction is started. Here, the logical counter is incremented after a new timestamp has been assigned.

Optimistic
It is based on the assumption that conflict is rare and it is more efficient to allow transactions to proceed without imposing delays to ensure serializability.

DATABASE DIRECTORIES

A directory is a container that is used to contain folders and files. It organizes files and folders in a hierarchical manner.

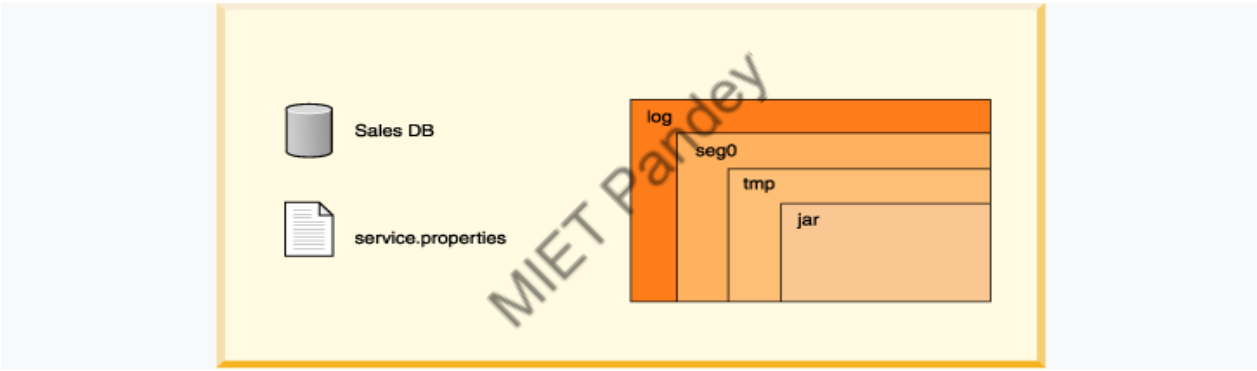


A Derby database is stored in files that live in a directory of the same name as the database. Database directories typically live in system directories.
Note: An in-memory database does not use the file system, but the size limits listed in the table later in this topic still apply. For some limits, the maximum value is determined by the available main memory instead of the available disk space and file system limitations.
A database directory contains the following, as shown in the following figure.

- **log directory**
  Contains files that make up the database transaction log, used internally for data recovery (not the same thing as the error log).
- **seg0 directory**
  Contains one file for each user table, system table, and index (known as conglomerates).
- **service.properties file**
  A text file with internal configuration information.
- **tmp directory**
  (might not exist.) A temporary directory used by Derby for large sorts and deferred updates and deletes. Sorts are used by a variety of SQL statements. For databases on read-only media, you might need to set a property to change the location of this directory. See "Creating Derby Databases for Read-Only Use".

- **jar directory**

  (might not exist.) A directory in which jar files are stored when you use database class loading.



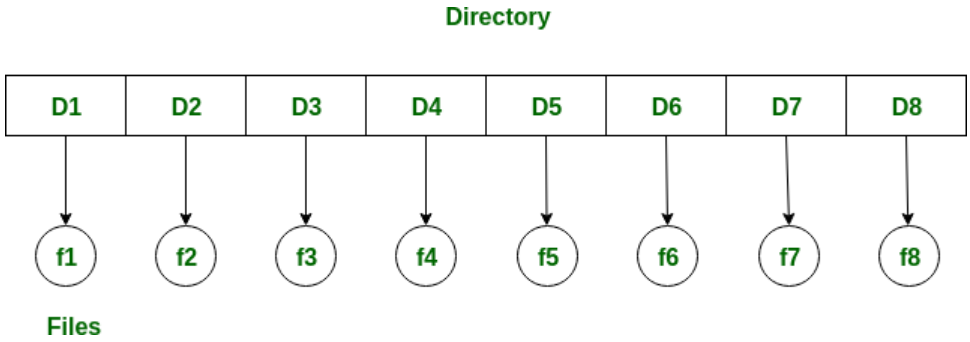
Derby imposes relatively few limitations on the number and size of databases and database objects. The following table shows some size limitations of Derby databases and database objects.

Table 1. Size limits for Derby database objects

Type of Object	Limit
Tables in each database	java.lang.Long.MAX_VALUESome operating systems impose a limit to the number of files allowed in a single directory.
Indexes in each table	32,767 or storage
Columns in each table	1,012
Number of columns on an index key	16
Rows in each table	No limit.
Size of table	No limit. Some operating systems impose a limit on the size of a single file.
Size of row	No limit. Rows can span pages. Rows cannot span tables so some operating systems impose a limit on the size of a single file, which results in limiting the size of a table and size of a row in that table.

For a complete list of restrictions on Derby databases and database objects, see the Java DB Reference Manual.

- **Single-level directory –**
  The single-level directory is the simplest directory structure. In it, all files are contained in the same directory which makes it easy to support and understand.
  A single level directory has a significant limitation, however, when the number of files increases or when the system has more than one user. Since all the files are in the same directory, they must have a unique name. if two users call their dataset test, then the unique name rule violated.



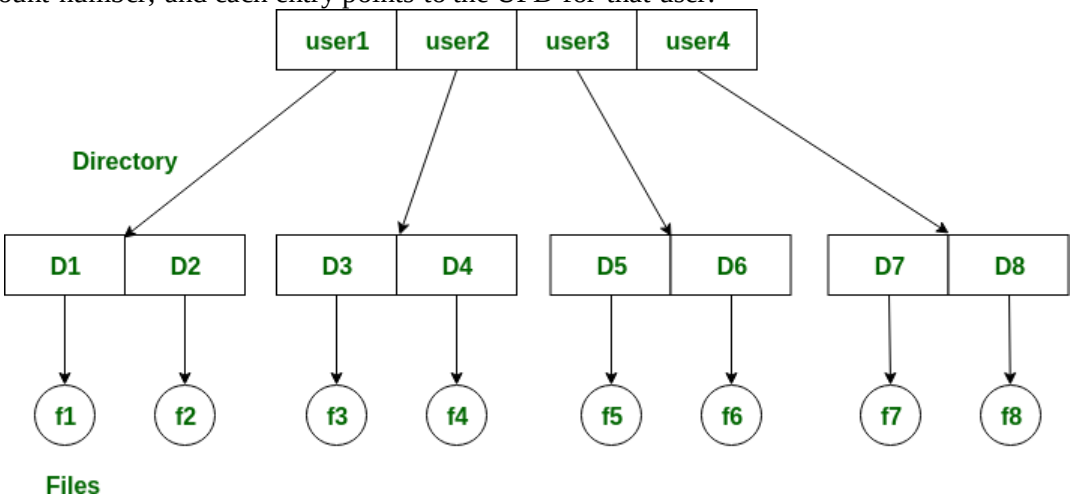
Advantages:

- Since it is a single directory, so its implementation is very easy.
- If the files are smaller in size, searching will become faster.
- The operations like file creation, searching, deletion, updating are very easy in such a directory structure.

Disadvantages:

- There may change of name collision because two files can have the same name.
- Searching will become time taking if the directory is large.
- This cannot group the same type of files together.

Two-level directory –
As we have seen, a single level directory often leads to confusion of files names among different users. The solution to this problem is to create a separate directory for each user.
In the two-level directory structure, each user has their own user files directory (UFD). The UFDs have similar structures, but each lists only the files of a single user. System's master file directory (MFD) is searches whenever a new user id=s logged in. The MFD is indexed by username or account number, and each entry points to the UFD for that user.



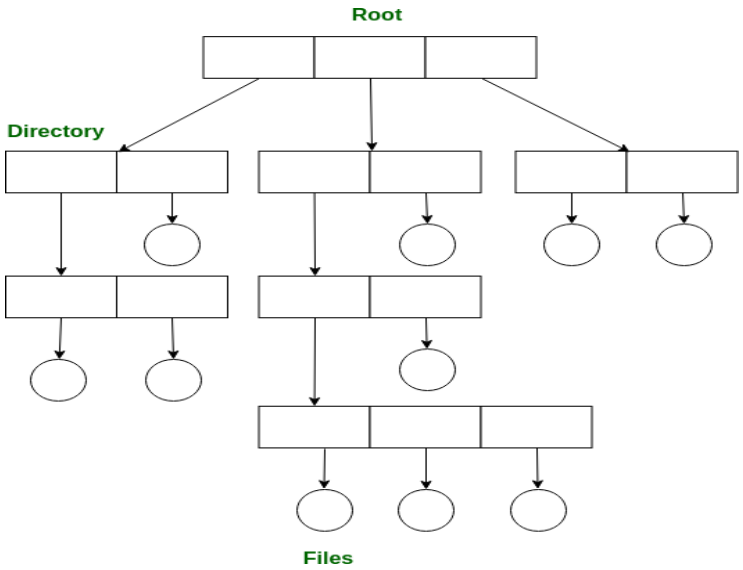
Advantages:

- We can give full path like /User-name/directory-name/.
- Different users can have the same directory as well as the file name.
- Searching of files becomes easier due to pathname and user-grouping.

Disadvantages:

- A user is not allowed to share files with other users.
- Still, it not very scalable, two files of the same type cannot be grouped together in the same user.

- Tree-structured directory –
  Once we have seen a two-level directory as a tree of height 2, the natural generalization is to extend the directory structure to a tree of arbitrary height.
  This generalization allows the user to create their own subdirectories and to organize their files accordingly.



A tree structure is the most common directory structure. The tree has a root directory, and every file in the system has a unique path.

Advantages:

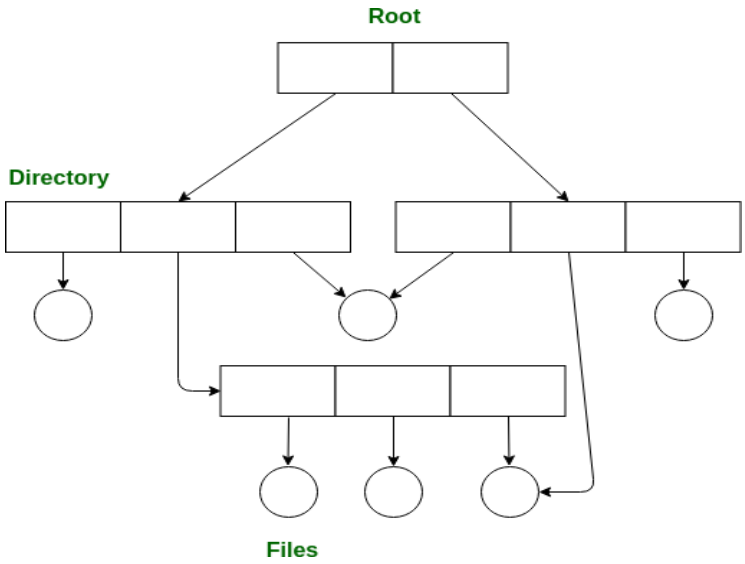
- Very general, since full pathname can be given.
- Very scalable, the probability of name collision is less.
- Searching becomes very easy; we can use both absolute paths as well as relative.

Disadvantages:

- Every file does not fit into the hierarchical model; files may be saved into multiple directories.
- We cannot share files.
- It is inefficient, because accessing a file may go under multiple directories.

- **Acyclic graph directory –**
  An acyclic graph is a graph with no cycle and allows us to share subdirectories and files. The same file or subdirectories may be in two different directories. It is a natural generalization of the tree-structured directory.
  It is used in the situation like when two programmers are working on a joint project and they need to access files. The associated files are stored in a subdirectory, separating them from other projects and files of other programmers since they are working on a joint project so they want the subdirectories to be into their own directories. The common subdirectories should be shared. So here we use acyclic directories.

  It is the point to note that the shared file is not the same as the copy file. If any programmer makes some changes in the subdirectory it will reflect in both subdirectories.



Advantages:

- We can share files.
- Searching is easy due to different-different paths.

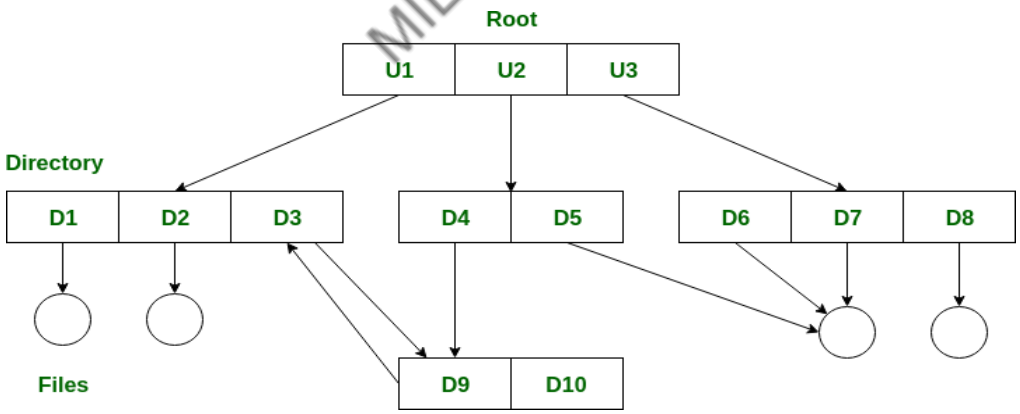
Disadvantages:

- We share the files via linking, in case deleting it may create the problem,
- If the link is a soft link then after deleting the file we left with a dangling pointer.
- In the case of a hard link, to delete a file we have to delete all the references associated with it.

General graph directory structure –

In general graph directory structure, cycles are allowed within a directory structure where multiple directories can be derived from more than one parent directory.

The main problem with this kind of directory structure is to calculate the total size or space that has been taken by the files and directories.



Advantages:

- It allows cycles.
- It is more flexible than other directories structure.

Disadvantages:

- It is more costly than others.
- It needs garbage collection.