Transaction Processing Concepts

Transaction in DBMS

Transaction in Database Management Systems (DBMS) can be defined as a set of logically related operations.

- It is the result of a request made by the user to access the contents of the database and perform operations on it.
- It consists of various operations and has various states in its completion journey.
- It also has some specific properties that must be followed to keep the database consistent.

Operations of Transaction

A user can make different types of requests to access and modify the contents of a database. So, we have different types of operations relating to a transaction. They are discussed as follows:

i) Read(X)

- A **read operation** is used to read the value of **X** from the database and store it in a buffer in the main memory for further actions such as displaying that value.
- Such an operation is performed when a user wishes just to see any content of the database and not make any changes to it. For example, when a user wants to check **his/her** account's balance, a read operation would be performed on user's account balance from the database.

ii) Write(X)

- A write operation is used to write the value to the database from the buffer in the main memory. For a write operation to be performed, first a read operation is performed to bring its value in buffer, and then some changes are made to it,
- **For example**, when a user requests to withdraw some money from his account, his account balance is fetched from the database using a read operation, then the amount to be deducted from the account is subtracted from this value, and then the obtained value is stored back in the database using a write operation.

iii) Commit

- This operation in transactions is used to maintain integrity in the database. Due to some failure of power, hardware, or software, etc., a transaction might get interrupted before all its operations are completed. This may cause ambiguity in the database, i.e. it might get inconsistent before and after the transaction.
- To ensure that further operations of any other transaction are performed only after work of the current transaction is done, a commit operation is performed to the changes made by a transaction permanently to the database.

iv) Rollback

- This operation is performed to bring the database to the last saved state when any transaction is interrupted in between due to any power, hardware, or software failure.
- In simple words, it can be said that a rollback operation does undo the operations of transactions that were performed before its interruption to achieve a safe state of the database and avoid any kind of ambiguity or inconsistency.

Transaction Schedules

When multiple transaction requests are made at the same time, we need to decide their order of execution. Thus, a transaction schedule can be defined as a chronological order of execution of multiple transactions. There are broadly two types of transaction schedules discussed as follows:

i) Serial Schedule

- In this kind of schedule, when multiple transactions are to be executed, they are executed serially, i.e. at one time only one transaction is executed while others wait for the execution of the current transaction to be completed. This ensures consistency in the database as transactions do not execute simultaneously.
- But, it increases the waiting time of the transactions in the queue, which in turn lowers the throughput of the system, i.e. number of transactions executed per time.

- To improve the throughput of the system, another kind of schedule are used which has some more strict rules which help the database to remain consistent even when transactions execute simultaneously.

ii) Non-Serial Schedule

- To reduce the waiting time of transactions in the waiting queue and improve the system efficiency, we use nonserial schedules which allow multiple transactions to start before a transaction is completely executed. This may sometimes result in inconsistency and errors in database operation.
- So, these errors are handled with specific algorithms to maintain the consistency of the database and improve CPU throughput as well.
- Non-serial schedules are also sometimes referred to as parallel schedules, as transactions execute in parallel in these kinds of schedules.

Serializable

- Serializability in DBMS is the property of a nonserial schedule that determines whether it would maintain the database consistency or not.
- The nonserial schedule which ensures that the database would be consistent after the transactions are executed in the order determined by that schedule is said to be Serializable Schedules.
- The serial schedules always maintain database consistency as a transaction starts only when the execution of the other transaction has been completed under it.
- Thus, serial schedules are always serializable.
- A transaction is a series of operations, so various states occur in its completion journey. They are discussed as follows:

i) Active

- It is the first stage of any transaction when it has begun to execute. The execution of the transaction takes place in this state.
- Operations such as insertion, deletion, or updation are performed during this state.
- During this state, the data records are under manipulation and they are not saved to the database, rather they remain somewhere in a buffer in the main memory.

ii) Partially Committed

- This state of transaction is achieved when it has completed most of the operations and is executing its final operation.
- It can be a signal to the commit operation, as after the final operation of the transaction completes its execution, the data has to be saved to the database through the commit operation.
- If some kind of error occurs during this state, the transaction goes into a failed state, else it goes into the Committed state.

iii) Commited

This state of transaction is achieved when all the transaction-related operations have been executed successfully along with the Commit operation, i.e. data is saved into the database after the required manipulations in this state. This marks the successful completion of a transaction.

iv) Failed

- If any of the transaction-related operations cause an error during the active or partially committed state, further execution of the transaction is stopped and it is brought into a failed state. Here, the database recovery system makes sure that the database is in a consistent state.

v) Aborted

If the error is not resolved in the failed state, then the transaction is aborted and a rollback operation is performed to bring database to the the last saved consistent state. When the transaction is aborted, the database recovery module either restarts the transaction or kills it.

The illustration below shows the various states that a transaction may encounter in its completion journey.

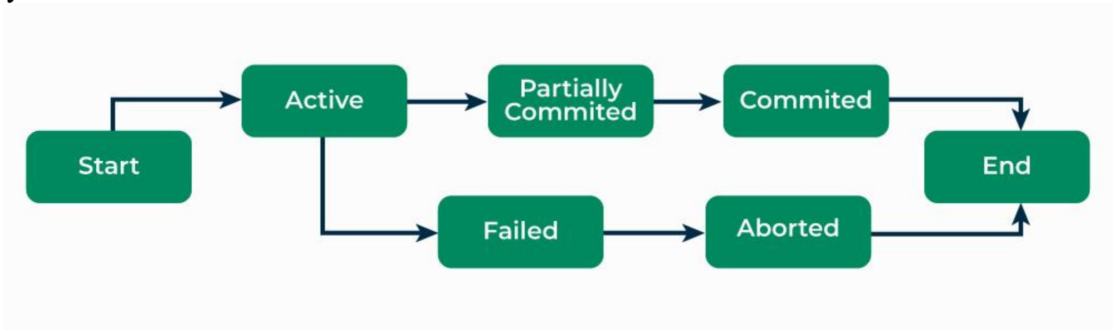
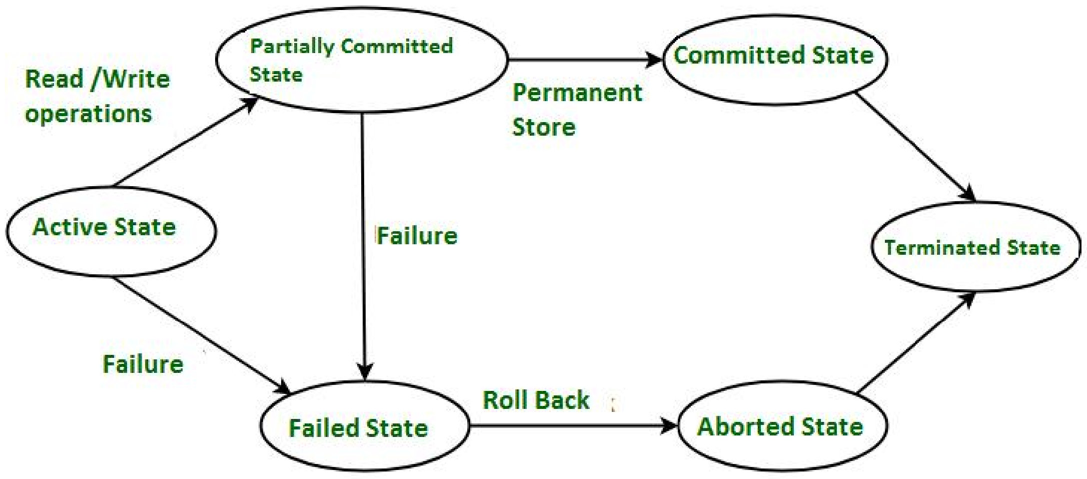


Fig: Transaction in DBMS

Transaction States in DBMS

A Transaction log is a file maintained by the recovery management component to record all the activities of the transaction. After the commit is done transaction log file is removed.



Transaction States in DBMS

In DBMS, a transaction passes through various states such as active, partially committed, failed, and aborted.

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

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

3. Failed State – When any instruction of the transaction fails, it goes to the "failed state" or if failure occurs in making a permanent change of data on Database.

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

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

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

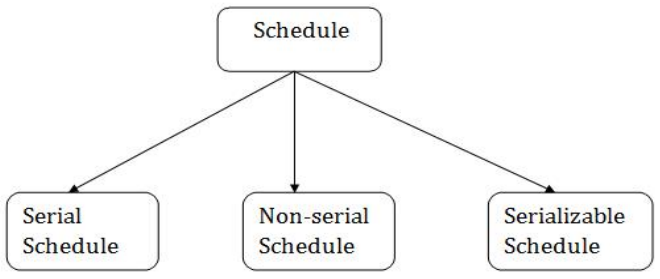
ACID Properties of Transaction:
ACID stands for Atomicity, Consistency, Isolation and Durability

1. **Atomicity:** All the operations in a transaction are considered to be atomic and as one unit. If system fails or any read/write conflicts occur during transaction the system needs to revert back to its previous state. Atomicity is maintained by the Transaction Management Component.
2. **Consistency:** Every transaction should lead to database connection from one valid state to other valid state. If system fails because of invalid data while doing an operation revert back the system to its previous state. Consistency is maintained by the Application manager.
3. **Isolation:** If multiple transactions are executing on single database, each transaction should be isolated from other transaction. If multiple transactions are performed on single database, operation from any transaction should not interfere with operation in other transaction. Isolation is maintained by the concurrency control manager.
4. **Durability:** Durability means the changes made during the transactions should exist after completion of transaction. Changes must be permanent and must not be lost due to any database failure. It is maintained by the recovery manager.

Example:
A has an account with an amount of Rs 150. B has an account with an amount of Rs 50. A is transferring amount Rs 100 to B's account.

1. **Atomicity:** Operations required for transfer are: Deduct amount Rs100 from A's account. Add amount Rs 100 to B's account. All operations should be done. If system fails to add amount in B's account after deducting from A's account, revert the operation on A's account.
2. **Consistency:** The sum amount in A's account and B's account should be same before and after the transaction completes. In the example the sum of both account before and after transaction is Rs 200, which preserves the consistency.
3. **Isolation:** If there is any other transaction (let between A and C) is going on, it should not affect the transaction between A and B i.e., both the transactions should be isolated.
4. **Durability:** It may happen system gets crashed after the completion of all operations then, after restarting it should preserve all the changes. The amount in A's and B's account should be same before and after the system restart.

Schedule
A series of operation from one transaction to another transaction is known as schedule. It is used to preserve the order of the operation in each of the individual transaction.



1. Serial Schedule
The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction. In the serial schedule, when the first transaction completes its cycle, then the next transaction is executed.

For example: Suppose there are two transactions T1 and T2 which have some operations. If it has no interleaving of operations, then there are the following two possible outcomes:

1. Execute all the operations of T1 which was followed by all the operations of T2.
2. Execute all the operations of T1 which was followed by all the operations of T2.

- In the given (a) figure, Schedule A shows the serial schedule where T1 followed by T2.
- In the given (b) figure, Schedule B shows the serial schedule where T2 followed by T1.

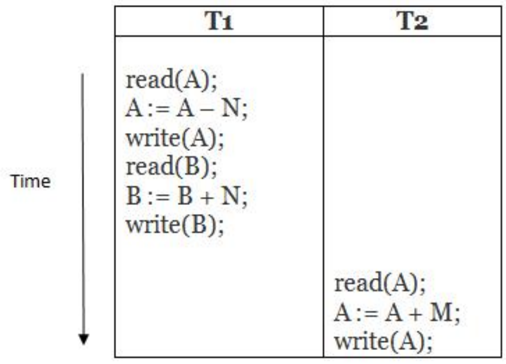
2. Non-serial Schedule

- If interleaving of operations is allowed, then there will be non-serial schedule.
- It contains many possible orders in which the system can execute the individual operations of the transactions.
- In the given figure (c) and (d), Schedule C and Schedule D are the non-serial schedules. It has interleaving of operations.

3. Serializable schedule

- The serializability of schedules is used to find non-serial schedules that allow the transaction to execute concurrently without interfering with one another.
- It identifies which schedules are correct when executions of the transaction have interleaving of their operations.
- A non-serial schedule will be serializable if its result is equal to the result of its transactions executed serially.

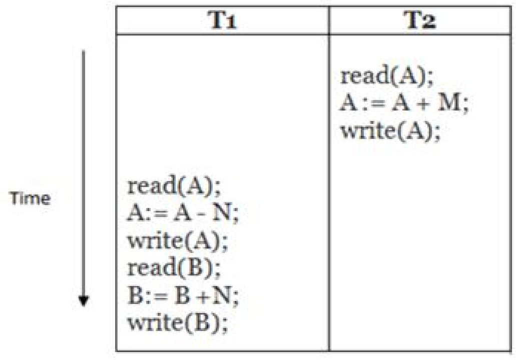
(a)



T1	T2
read(A);	
A := A – N;	
write(A);	
read(B);	
B := B + N;	
write(B);	
	read(A);
	A := A + M;
	write(A);

Schedule A

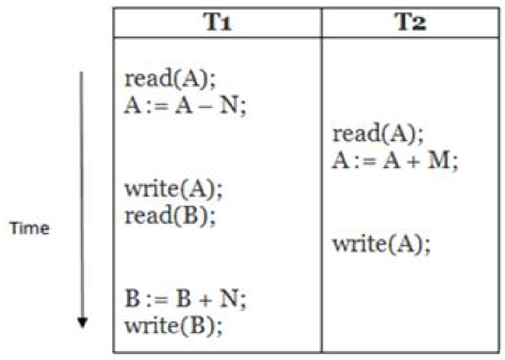
(b)



T1	T2
	read(A);
	A := A + M;
	write(A);
read(A);	
A:= A - N;	
write(A);	
read(B);	
B:= B +N;	
write(B);	

Schedule B

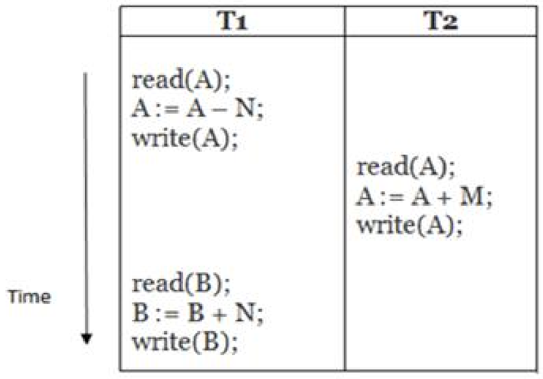
(c)



T1	T2
read(A); A := A − N;	
	read(A); A := A + M;
write(A); read(B);	
	write(A);
B := B + N; write(B);	

Schedule C

(d)



T1	T2
read(A); A := A − N; write(A);	
	read(A); A := A + M; write(A);
read(B); B := B + N; write(B);	

Schedule D

Here,

Schedule A and Schedule B are serial schedule.

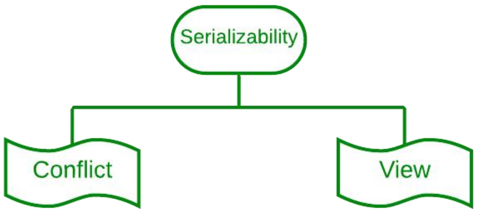
Schedule C and Schedule D are Non-serial schedule.

Serializability in DBMS

If a non-serial schedule can be transformed into its corresponding serial schedule, it is said to be serializable. Simply said, a non-serial schedule is referred to as a serializable schedule if it yields the same results as a serial timetable.

Types of Serializability

There are two ways to check whether any non-serial schedule is serializable.



1. Conflict serializability

Conflict serializability refers to a subset of serializability that focuses on maintaining the consistency of a database while ensuring that identical data items are executed in an order. In a

DBMS each transaction has a value and all the transactions, in the database rely on this uniqueness. This uniqueness ensures that no two operations with the conflict value can occur simultaneously.
For example lets consider an order table and a customer table as two instances. Each order is associated with one customer even though a single client may place orders. However there are restrictions for achieving conflict serializability in the database. Here are a few of them.

1. Different transactions should be used for the two procedures.
2. The identical data item should be present in both transactions.
3. Between the two operations, there should be at least one write operation.

Example
Three transactions—t1, t2, and t3—are active on a schedule "S" at once. Let's create a graph of precedence.

Transaction – 1 (t1)	**Transaction – 2 (t2)**	**Transaction – 3 (t3)**
R(a)		
	R(b)	
		R(b)
	W(b)	
W(a)		
		W(a)
	R(a)	
	W(a)	

It is a conflict serializable schedule as well as a serial schedule because the graph (a DAG) has no loops. We can also determine the order of transactions because it is a serial schedule.

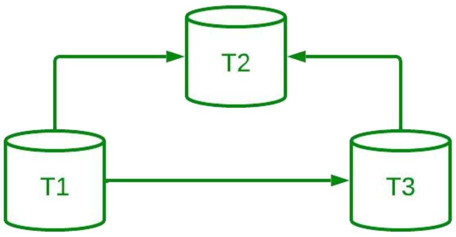


Fig: DAG of transactions
As there is no incoming edge on Transaction 1, Transaction 1 will be executed first. T3 will run second because it only depends on T1. Due to its dependence on both T1 and T3, t2 will finally be executed.
Therefore, the serial schedule's equivalent order is: t1 –> t3 –> t2
Note: A schedule is unquestionably consistent if it is conflicting serializable. A non-conflicting serializable schedule, on the other hand, might or might not be serial. We employ the idea of View Serializability to further examine its serial behavior.

2. View Serializability
View serializability is a kind of operation in a serializable in which each transaction should provide some results, and these outcomes are the output of properly sequentially executing the data item. The view serializability, in contrast to conflict serialized, is concerned with avoiding database inconsistency. The view serializability feature of DBMS enables users to see databases in contradictory ways.
To further understand view serializability in DBMS, we need to understand the schedules S1 and S2. The two transactions T1 and T2 should be used to establish these two schedules. Each schedule must follow the three transactions in order to retain the equivalent of the transaction. These three circumstances are listed below.

1. The first prerequisite is that the same kind of transaction appears on every schedule. This requirement means that the same kind of group of transactions cannot appear on both schedules S1 and S2. The schedules are not equal to one another if one schedule commits a transaction but it does not match the transaction of the other schedule.
2. The second requirement is that different read or write operations should not be used in either schedule. On the other hand, we say that two schedules are not similar if schedule S1 has two write operations whereas schedule S2 only has one. The number of the write operation must be the same in both schedules, however there is no issue if the number of the read operation is different.
3. The second to last requirement is that there should not be a conflict between either timetable. execution order for a single data item. Assume, for instance, that schedule S1's transaction is T1, and schedule S2's transaction is T2. The data item A is written by both the transaction T1 and the transaction T2. The schedules are not equal in this instance. However, we referred to the schedule as equivalent to one another if it had the same number of all write operations in the data item.

Serializability testing
We can utilize the Serialization Graph or Precedence Graph to examine a schedule's serializability. A schedule's full transactions are organized into a Directed Graph, what a serialization graph is.

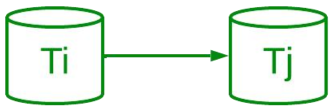


Fig: Precedence Graph
It can be described as a Graph G(V, E) with vertices V = "V1, V2, V3,…, Vn" and directed edges E = "E1, E2, E3,…, En". One of the two operations—READ or WRITE—performed by a certain transaction is contained in the collection of edges. Where Ti -> Tj, means Transaction-Ti is either performing read or write before the transaction-Tj.

Recoverability in DBMS
Recoverability is a property of database systems that ensures that, in the event of a failure or error, the system can recover the database to a consistent state. Recoverability guarantees that all committed transactions are durable and that their effects are permanently stored in the database, while the effects of uncommitted transactions are undone to maintain data consistency.
The recoverability property is enforced through the use of transaction logs, which record all changes made to the database during transaction processing. When a failure occurs, the system uses the log to recover the database to a consistent state, which involves either undoing the effects of uncommitted transactions or redoing the effects of committed transactions.

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

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

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

Database recovery techniques ensure data integrity in case of system failures.

Rollback/Undo Recovery Technique

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

Commit/Redo Recovery Technique

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

Checkpoint Recovery Technique

Checkpoint Recovery is a technique used to improve data integrity and system stability, especially in databases and distributed systems. It entails preserving the system's state at regular intervals, known as checkpoints, at which all ongoing transactions are either completed or not initiated. This saved state, which includes memory and CPU registers, is kept in stable, non-volatile storage so that it can withstand system crashes. In the event of a breakdown, the system can be restored to the most recent checkpoint, which reduces data loss and downtime. The frequency of checkpoint formation is carefully regulated to decrease system overhead while ensuring that recent data may be restored quickly.

Overall, recovery techniques are essential to ensure data consistency and availability in Database Management System, and each technique has its own advantages and limitations that must be considered in the design of a recovery system.

Log based Recovery in DBMS

The atomicity property of DBMS states that either all the operations of transactions must be performed or none. The modifications done by an aborted transaction should not be visible to the database and the modifications done by the committed transaction should be visible. To achieve our goal of atomicity, the user must first output stable storage information describing the modifications, without modifying the database itself. This information can help us ensure that all modifications performed by committed transactions are reflected in the database. This information can also help us ensure that no modifications made by an aborted transaction persist in the database.

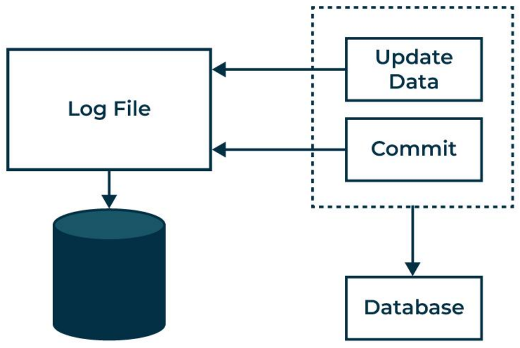


Fig: Log based Recovery in DBMS

Log and log records

The log is a sequence of log records, recording all the updated activities in the database. In stable storage, logs for each transaction are maintained. Any operation which is performed on the database is recorded on the log. Prior to performing any modification to the database, an updated log record is created to reflect that modification. An update log record represented as: <Ti, Xj, V1, V2> has these fields:

1. **Transaction identifier:** Unique Identifier of the transaction that performed the write operation.
2. **Data item:** Unique identifier of the data item written.
3. **Old value:** Value of data item prior to write.
4. **New value:** Value of data item after write operation.

Other types of log records are:

1. **<Ti start>** : It contains information about when a transaction Ti starts.
2. **<Ti commit>** : It contains information about when a transaction Ti commits.
3. **<Ti abort>** : It contains information about when a transaction Ti aborts.

Undo and Redo Operations

Because all database modifications must be preceded by the creation of a log record, the system has available both the old value prior to the modification of the data item and new value that is to be written for data item. This allows system to perform redo and undo operations as appropriate:

1. **Undo:** using a log record sets the data item specified in log record to old value.
2. **Redo:** using a log record sets the data item specified in log record to new value.

The database can be modified using two approaches –

1. **Deferred Modification Technique:** If the transaction does not modify the database until it has partially committed, it is said to use deferred modification technique.
2. **Immediate Modification Technique:** If database modification occur while the transaction is still active, it is said to use immediate modification technique.

Recovery using Log records

After a system crash has occurred, the system consults the log to determine which transactions need to be redone and which need to be undone.

1. Transaction Ti needs to be undone if the log contains the record <Ti start> but does not contain either the record <Ti commit> or the record <Ti abort>.
2. Transaction Ti needs to be redone if log contains record <Ti start> and either the record <Ti commit> or the record <Ti abort>.

Use of Checkpoints – When a system crash occurs, user must consult the log. In principle, that need to search the entire log to determine this information. There are two major difficulties with this approach:

1. The search process is time-consuming.
2. Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will cause recovery to take longer.

To reduce these types of overhead, user introduce checkpoints. A log record of the form <checkpoint L> is used to represent a checkpoint in log where L is a list of transactions active at the time of the checkpoint. When a checkpoint log record is added to log all the transactions that have committed before this checkpoint have <Ti commit> log record before the checkpoint record. Any database modifications made by Ti is written to the database either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on Ti. After a system crash has occurred, the system examines the log to find the last <checkpoint L> record. The redo or undo operations need to be applied only to transactions in L, and to all transactions that started execution after the record was written to the log. Let us denote this set of transactions as T. Same rules of undo and redo are applicable on T as mentioned in Recovery using Log records part. Note that user need to only examine the part of the log starting with the last checkpoint log record to find the set of transactions T, and to find out whether a commit or abort record occurs in the log for each transaction in T. For example, consider the set of transactions {T0, T1, . . ., T100}. Suppose that the most recent checkpoint took place during the execution of transaction T67 and T69, while T68 and all transactions with subscripts lower than 67 completed before the checkpoint. Thus, only transactions T67, T69, . . ., T100 need to be considered during the recovery scheme. Each of them needs to be redone if it has completed (that is, either committed or aborted); otherwise, it was incomplete, and needs to be undone.

Log-based recovery is a technique used in database management systems (DBMS) to recover a database to a consistent state in the event of a failure or crash. It involves the use of transaction logs, which are records of all the transactions performed on the database.

In log-based recovery, the DBMS uses the transaction log to reconstruct the database to a consistent state. The transaction log contains records of all the changes made to the database, including updates, inserts, and deletes. It also records information about each transaction, such as its start and end times.

When a failure occurs, the DBMS uses the transaction log to determine which transactions were incomplete at the time of the failure. It then performs a series of operations to undo the incomplete transactions and redo the completed ones. This process is called the redo/undo recovery algorithm.

The redo operation involves reapplying the changes made by completed transactions that were not yet saved to the database at the time of the failure. This ensures that all changes are applied to the database.

The undo operation involves undoing the changes made by incomplete transactions that were saved to the database at the time of the failure. This restores the database to a consistent state by reversing the effects of the incomplete transactions.

Once the redo and undo operations are completed, the DBMS can bring the database back online and resume normal operations.

Log-based recovery is an essential feature of modern DBMSs and provides a reliable mechanism for recovering from failures and ensuring the consistency of the database.

Advantages of Log based Recovery

- **Durability:** In the event of a breakdown, the log file offers a dependable and long-lasting method of recovering data. It guarantees that in the event of a system crash, no committed transaction is lost.
- **Faster Recovery:** Since log-based recovery recovers databases by replaying committed transactions from the log file, it is typically faster than alternative recovery methods.
- **Incremental Backup:** Backups can be made in increments using log-based recovery. Just the changes made since the last backup are kept in the log file, rather than creating a complete backup of the database each time.
- **Lowers the Risk of Data Corruption:** By making sure that all transactions are correctly committed or canceled before they are written to the database , log-based recovery lowers the risk of data corruption.

Disadvantages of Log based Recovery

- **Additional overhead:** Maintaining the log file incurs an additional overhead on the database system, which can reduce the performance of the system.
- **Complexity:** Log-based recovery is a complex process that requires careful management and administration. If not managed properly, it can lead to data inconsistencies or loss.
- **Storage space:** The log file can consume a significant amount of storage space, especially in a database with a large number of transactions.
- **Time-Consuming:** The process of replaying the transactions from the log file can be time-consuming, especially if there are a large number of transactions to recover.

Deadlock in DBMS

In database management systems (DBMS) a deadlock occurs when two or more transactions are unable to the proceed because each transaction is waiting for the other to the release locks on resources. This situation creates a cycle of the dependencies where no transaction can continue leading to the standstill in the system. The Deadlocks can severely impact the performance and reliability of a DBMS making it crucial to the understand and manage them effectively.

What is Deadlock?

The Deadlock is a condition in a multi-user database environment where transactions are unable to the complete because they are each waiting for the resources held by other transactions. This results in a cycle of the dependencies where no transaction can proceed.

Basically, **Deadlocks occur when two or more transactions wait indefinitely for resources held by each other.** Also, mastering how to detect and resolve deadlocks is vital for database efficiency.

Characteristics of Deadlock

- Mutual Exclusion: Only one transaction can hold a particular resource at a time.
- Hold and Wait: The Transactions holding resources may request additional resources held by others.
- No Preemption: The Resources cannot be forcibly taken from the transaction holding them.
- Circular Wait: A cycle of transactions exists where each transaction is waiting for the resource held by the next transaction in the cycle.

In a database management system (DBMS), a deadlock occurs when two or more transactions are waiting for each other to release resources, such as locks on database objects, that they need to complete their operations. As a result, none of the transactions can proceed, leading to a situation where they are stuck or “deadlocked.”

Deadlocks can happen in multi-user environments when two or more transactions are running concurrently and try to access the same data in a different order. When this happens, one transaction may hold a lock on a resource that another transaction needs, while the second transaction may hold a lock on a resource that the first transaction needs. Both transactions are then blocked, waiting for the other to release the resource they need.

DBMSs often use various techniques to detect and resolve deadlocks automatically. These techniques include timeout mechanisms, where a transaction is forced to release its locks after a certain period of time, and deadlock detection algorithms, which periodically scan the transaction log for deadlock cycles and then choose a transaction to abort to resolve the deadlock.

It is also possible to prevent deadlocks by careful design of transactions, such as always acquiring locks in the same order or releasing locks as soon as possible. Proper design of the database schema and application can also help to minimize the likelihood of deadlocks.

In a database, a deadlock is an unwanted situation in which two or more transactions are waiting indefinitely for one another to give up locks. Deadlock is said to be one of the most feared complications in DBMS as it brings the whole system to a Halt.

Example – let us understand the concept of deadlock 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.

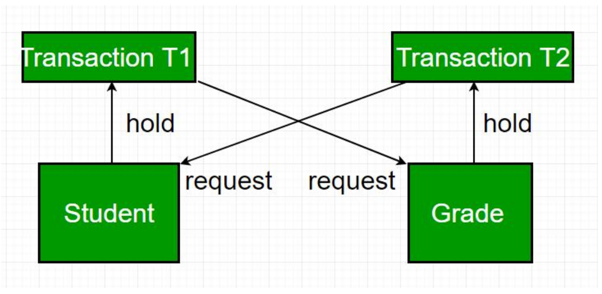


Fig: Deadlock in DBMS

What is Deadlock Avoidance?

When a database is stuck in a deadlock, It is always better to avoid the deadlock rather than restarting or aborting the database. The deadlock avoidance method is suitable for smaller databases whereas the deadlock prevention method is suitable for larger databases.

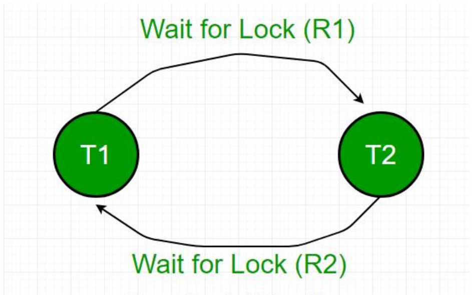
One method of avoiding deadlock is using application-consistent logic. In the above-given example, Transactions that access Students and Grades should always access the tables in the same order. In this way, in the scenario described above, Transaction T1 simply waits for transaction T2 to release the lock on Grades before it begins. When transaction T2 releases the lock, Transaction T1 can proceed freely.

Another method for avoiding deadlock is to apply both the row-level locking mechanism and the READ COMMITTED isolation level. However, It does not guarantee to remove deadlocks completely.

What is Deadlock Detection?

When a transaction waits indefinitely to obtain a lock, The database management system should detect whether the transaction is involved in a deadlock or not.

Wait-for-graph is one of the methods for detecting the deadlock situation. This method is suitable for smaller databases. In this method, a graph is drawn based on the transaction and its lock on the resource. If the graph created has a closed loop or a cycle, then there is a deadlock. For the above-mentioned scenario, the Wait-For graph is drawn below:



Deadlock Situation

What is Deadlock Prevention?

For a large database, the deadlock prevention method is suitable. A deadlock can be prevented if the resources are allocated in such a way that a deadlock never occurs. The DBMS analyzes the operations whether they can create a deadlock situation or not, If they do, that transaction is never allowed to be executed.

Deadlock prevention mechanism proposes two schemes:

- **Wait-Die Scheme:** In this scheme, If a transaction requests a resource that is locked by another transaction, then the DBMS simply checks the timestamp of both transactions and allows the older transaction to wait until the resource is available for execution.
  Suppose, there are two transactions T1 and T2, and Let the timestamp of any transaction T be TS (T). Now, If there is a lock on T2 by some other transaction and T1 is requesting resources held by T2, then DBMS performs the following actions:

Checks if TS (T1) < TS (T2) – if T1 is the older transaction and T2 has held some resource, then it allows T1 to wait until resource is available for execution. That means if a younger transaction has locked some resource and an older transaction is waiting for it, then an older transaction is allowed to wait for it till it is available. If T1 is an older transaction and has held some resource with it and if T2 is waiting for it, then T2 is killed and restarted later with random delay but with the same timestamp. i.e. if the older transaction has held some resource and the younger transaction waits for the resource, then the younger transaction is killed and restarted with a very minute delay with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

- **Wound Wait Scheme:** In this scheme, if an older transaction requests for a resource held by a younger transaction, then an older transaction forces a younger transaction to kill the transaction and release the resource. The younger transaction is restarted with a minute delay but with the same timestamp. If the younger transaction is requesting a resource that is held by an older one, then the younger transaction is asked to wait till the older one releases it.

The following table lists the differences between Wait – Die and Wound -Wait scheme prevention schemes:

Wait – Die	**Wound -Wait**
It is based on a non-preemptive technique.	It is based on a preemptive technique.
In this, older transactions must wait for the younger one to release its data items.	In this, older transactions never wait for younger transactions.
The number of aborts and rollbacks is higher in these techniques.	In this, the number of aborts and rollback is lesser.

Applications

1. **Delayed Transactions:** Deadlocks can cause transactions to be delayed, as the resources they need are being held by other transactions. This can lead to slower response times and longer wait times for users.
2. **Lost Transactions:** In some cases, deadlocks can cause transactions to be lost or aborted, which can result in data inconsistencies or other issues.
3. **Reduced Concurrency:** Deadlocks can reduce the level of concurrency in the system, as transactions are blocked waiting for resources to become available. This can lead to slower transaction processing and reduced overall throughput.
4. **Increased Resource Usage:** Deadlocks can result in increased resource usage, as transactions that are blocked waiting for resources to become available continue to consume system resources. This can lead to performance degradation and increased resource contention.
5. **Reduced User Satisfaction:** Deadlocks can lead to a perception of poor system performance and can reduce user satisfaction with the application. This can have a negative impact on user adoption and retention.

Features of Deadlock in a DBMS

1. **Mutual Exclusion:** Each resource can be held by only one transaction at a time, and other transactions must wait for it to be released.
2. **Hold and Wait:** Transactions can request resources while holding on to resources already allocated to them.
3. **No Preemption:** Resources cannot be taken away from a transaction forcibly, and the transaction must release them voluntarily.
4. **Circular Wait:** Transactions are waiting for resources in a circular chain, where each transaction is waiting for a resource held by the next transaction in the chain.
5. **Indefinite Blocking:** Transactions are blocked indefinitely, waiting for resources to become available, and no transaction can proceed.

6. **System Stagnation:** Deadlock leads to system stagnation, where no transaction can proceed, and the system is unable to make any progress.
7. **Inconsistent Data:** Deadlock can lead to inconsistent data if transactions are unable to complete and leave the database in an intermediate state.
8. **Difficult to Detect and Resolve:** Deadlock can be difficult to detect and resolve, as it may involve multiple transactions, resources, and dependencies.

Disadvantages

1. **System downtime:** Deadlock can cause system downtime, which can result in loss of productivity and revenue for businesses that rely on the DBMS.
2. **Resource waste:** When transactions are waiting for resources, these resources are not being used, leading to wasted resources and decreased system efficiency.
3. **Reduced concurrency:** Deadlock can lead to a decrease in system concurrency, which can result in slower transaction processing and reduced throughput.
4. **Complex resolution:** Resolving deadlock can be a complex and time-consuming process, requiring system administrators to intervene and manually resolve the deadlock.
5. **Increased system overhead:** The mechanisms used to detect and resolve deadlock, such as timeouts and rollbacks, can increase system overhead, leading to decreased performance.