UNIT 4

Transaction Concepts – ACID Properties – Schedules – Serializability – Concurrency Control – Need for Concurrency – Locking Protocols – Two Phase Locking – Deadlock – Transaction Recovery – Save Points – Isolation Levels – SQL Facilities for Concurrency and Recovery.

A **transaction** is a set of logically related operations. For example, you are transferring money from your bank account to your friend's account, the set of operations would be like this:

Simple Transaction Example

1. Read your account balance
2. Deduct the amount from your balance
3. Write the remaining balance to your account
4. Read your friend's account balance
5. Add the amount to his account balance
6. Write the new updated balance to his account

This whole set of operations can be called a transaction. Although I have shown you read, write and update operations in the above example but the transaction can have operations like read, write, insert, update, delete.

In DBMS, we write the above 6 steps transaction like this:

Lets say your account is A and your friend's account is B, you are transferring 10000 from Ato B, the steps of the transaction are:

```
1. R(A);
2. A = A - 10000;
3. W(A);
4. R(B);
5. B = B + 10000;
6. W(B);
```

In the above transaction **R** refers to the **Read operation** and **W** refers to the **write operation**.

Transaction failure in between the operations

Now that we understand what is transaction, we should understand what are the problems associated with it.

The main problem that can happen during a transaction is that the transaction can fail before finishing the all the operations in the set. This can happen due to power failure, system crash etc. This is a serious problem that can leave database in an inconsistent state. Assume that transaction fail after third operation (see the

example above) then the amount would be deducted from your account but your friend will not receive it.

To solve this problem, we have the following two operations

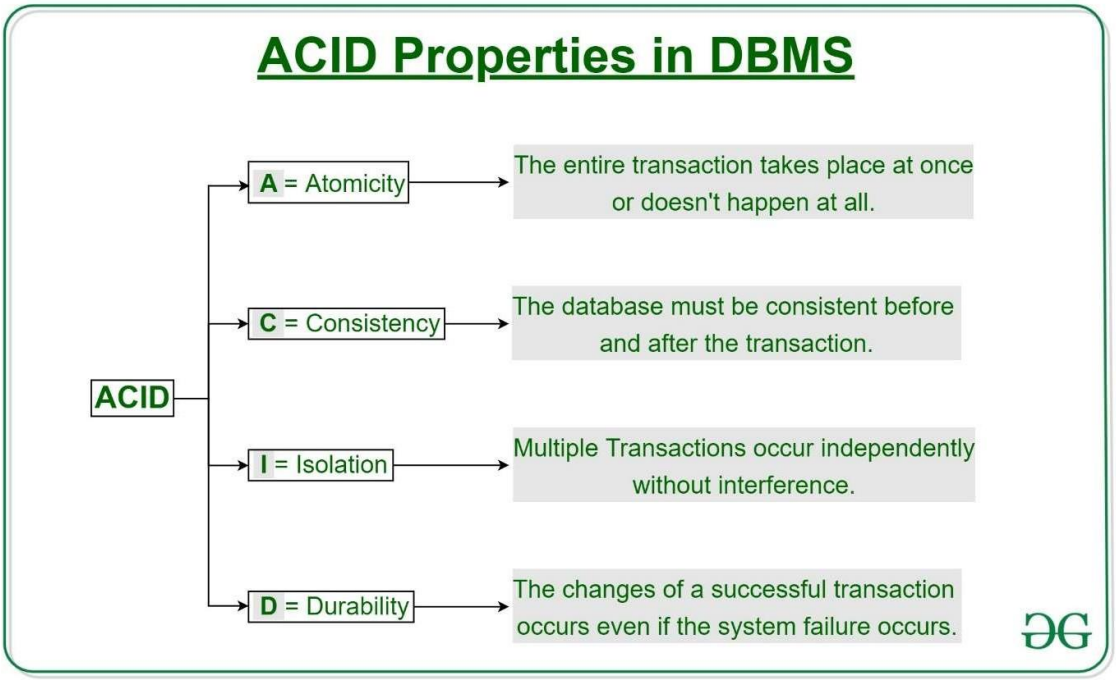
Commit: If all the operations in a transaction are completed successfully then commit those changes to the database permanently.

Rollback: If any of the operation fails then rollback all the changes done by previous operations.

Even though these operations can help us avoiding several issues that may arise during transaction but they are not sufficient when two transactions are running concurrently. To handle those problems we need to understand database ACID properties.

ACID PROPERTIES IN DBMS

A **transaction** is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.



Atomicity

By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

—**Abort**: If a transaction aborts, changes made to database are not visible.

—**Commit**: If a transaction commits, changes made are visible. Atomicity is also known as the 'All or nothing rule'.

Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X − 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of **T1** but before completion of **T2**.(say, after **write(X)** but before **write(Y)**), then amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

Consistency

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above, The total amount before and after the transaction must be maintained.

Total **before T** occurs = **500 + 200 = 700**.Total **after T occurs** = **400 + 300 = 700**.

Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. Asa result T is incomplete.

Isolation

This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order. Let **X**= 500, **Y** = 500.

Consider two transactions **T** and **T".**

T	T"
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y − 50 Write	Read (X) Read (Y) Z: = X + Y Write (Z)

Suppose **T** has been executed till **Read (Y)** and then **T''** starts. As a result , interleaving of operations takes place due to which **T''** reads correct value of **X** but incorrect value of **Y** and sum computed by

T'': (X+Y = 50, 000+500=50, 500)
is thus not consistent with the sum at end of transaction:
T: (X+Y = 50, 000 + 450 = 50, 450).

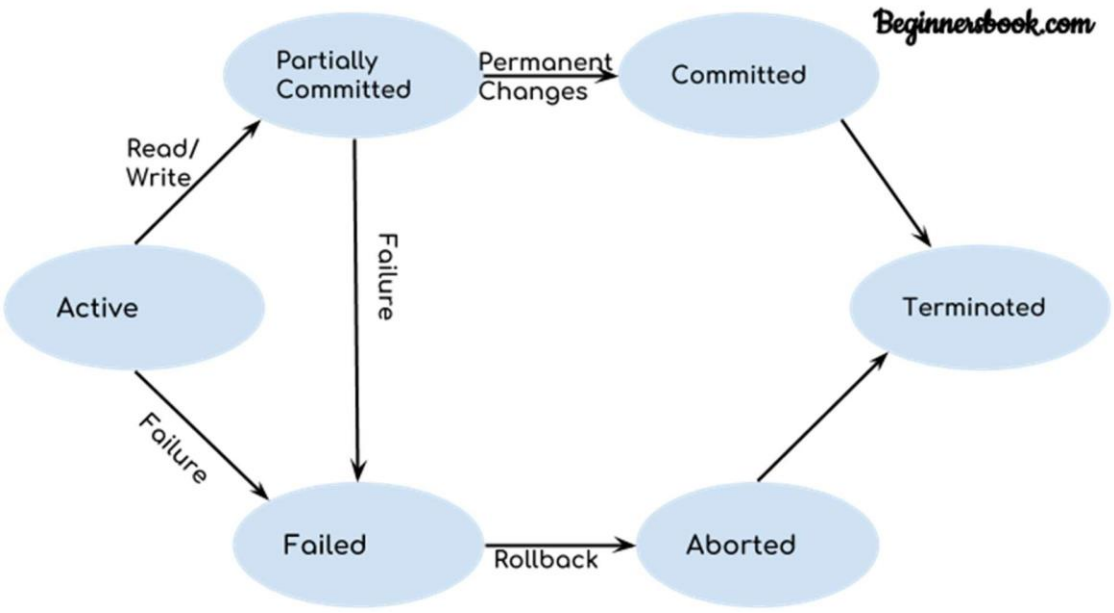
This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

Durability:
This property ensures that once the transaction has completed execution, the updates andmodifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.
The **ACID** properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored.

DBMS TRANSACTION STATES

In this guide, we will discuss the **states of a transaction in DBMS**. A transaction in DBMScan be in one of the following states.

DBMS Transaction States Diagram



Lets discuss these states one by one.
Active State

As we have discussed in the DBMS transaction introduction that a transaction is a sequence ofoperations. If a transaction is in execution then it is said to be in active state. It doesn't matterwhich step is in execution, until unless the transaction is executing, it remains in active state. **Failed State**

If a transaction is executing and a failure occurs, either a hardware failure or a software failurethen the transaction goes into failed state from the active state.

Partially Committed State

As we can see in the above diagram that a transaction goes into "partially committed" statefrom the active state when there are read and write operations present in the transaction.

A transaction contains number of read and write operations. Once the whole transaction is successfully executed, the transaction goes into partially committed state where we have all the read and write operations performed on the main memory (local memory) instead of the actual database.

The reason why we have this state is because a transaction can fail during execution so if we are making the changes in the actual database instead of local memory, database may be left in

an inconsistent state in case of any failure. **This state helps us to rollback the changes madeto the database in case of a failure during execution.**

Committed State

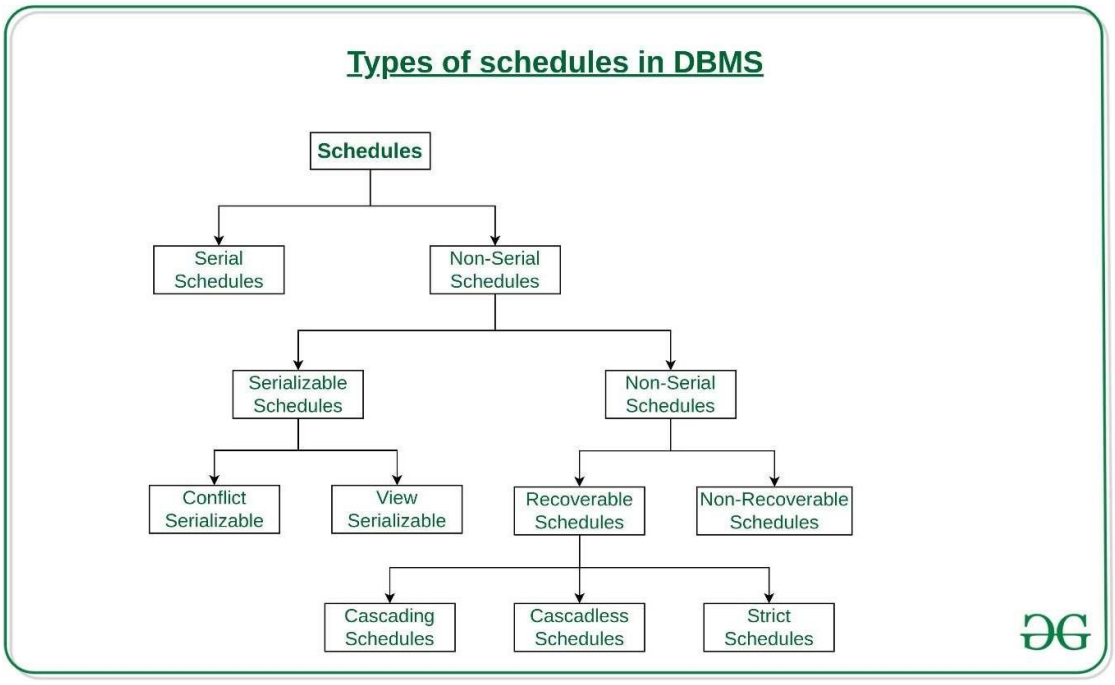
If a transaction completes the execution successfully then all the changes made in the local memory during **partially committed** state are permanently stored in the database. You can also see in the above diagram that a transaction goes from partially committed state to committed state when everything is successful.

Aborted State

As we have seen above, if a transaction fails during execution then the transaction goes into a failed state. The changes made into the local memory (or buffer) are rolled back to the previous consistent state and the transaction goes into aborted state from the failed state. Refer the diagram to see the interaction between failed and aborted state.

Types of Schedules in DBMS

Schedule, as the name suggests, is a process of lining the transactions and executing them one by one. When there are multiple transactions that are running in a concurrent manner and the order of operation is needed to be set so that the operations do not overlap each other, Scheduling is brought into play and the transactions are timed accordingly.



1. **Serial Schedules:**

   Schedules in which the transactions are executed non-interleaved, i.e., a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules. i.e., In Serial schedule, a transaction is executed completely before starting the execution of another transaction. In

other words, you can say that in serial schedule, a transaction does not start execution until the currently running transaction finished execution. This type of execution of transaction is also known as non-interleaved execution. The example we have seen above is the serial schedule.

Example: Consider the following schedule involving two transactions T1 and T2.

T_1	T_2
R(A)	
W(A)	
R(B)	
	W(B)
	R(A)
	R(B)

where R(A) denotes that a read operation is performed on some data item 'A' This is a serial schedule since the transactions perform serially in the order T1 — > T2

2. Non-Serial Schedule:

This is a type of Scheduling where the operations of multiple transactions are interleaved. This might lead to a rise in the concurrency problem. The transactions are executed in a non-serial manner, keeping the end result correct and same as the serial schedule. Unlike the serial schedule where one transaction must wait for another to complete all its operation, in the non-serial schedule, the other transaction proceeds without waiting for the previous transaction to complete. This sort of schedule does not provide any benefit of the concurrent transaction. It can be of two types namely, Serializable and Non-Serializable Schedule.

Serializability in DBMS

In the field of computer science, serializability is a term that is a property of the system that describes how the different process operates the shared data. If the result given by the system is similar to the operation performed by the system, then in this situation, we call that system serializable. Here the cooperation of the system means there is no overlapping in the execution of the data. In DBMS, when the data is being written or read then, the DBMS can stop all the other processes from accessing the data.

In the MongoDB developer certificate, the DBMS uses various locking systems to allow the other processes while maintaining the integrity of the data. In MongoDB, the most restricted level for serializability is the employee can be restricted by two-phase locking or 2PL. In the first phase of the locking level, the data objects are locked before the

execution of the operation. When the transaction has been accomplished, then the lock for the data object is released. This process guarantees that there is no conflict in operation and that all the transaction views the database as a conflict database.

The two-phase locking or 2PL system provides a strong guarantee for the conflict of the database.

Types of Serializability

In DBMS, all the transaction should be arranged in a particular order, even if all the transaction is concurrent. If all the transaction is not serializable, then it produces the incorrect result.

In DBMS, there are different types of serializable. Each type of serializable has some advantages and disadvantages. The two most common types of serializable are view serializability and conflict serializability.

1. Conflict Serializability

Conflict serializability is a type of conflict operation in serializability that operates the same data item that should be executed in a particular order and maintains the consistency of the database. In DBMS, each transaction has some unique value, and every transaction of the database is based on that unique value of the database.

This unique value ensures that no two operations having the same conflict value are executed concurrently. For example, let's consider two examples, i.e., the order table and the customer table. One customer can have multiple orders, but each order only belongs to one customer. There is some condition for the conflict serializability of the database. These are as below.

- Both operations should have different transactions.
- Both transactions should have the same data item.
- There should be at least one write operation between the two operations.

If there are two transactions that are executed concurrently, one operation has to add the transaction of the first customer, and another operation has added by the second operation. This process ensures that there would be no inconsistency in the database.

2. View Serializability

View serializability is a type of operation in the serializable in which each transaction should produce some result and these results are the output of proper sequential execution of the data item. Unlike conflict serialized, the view serializability focuses on preventing inconsistency in the database. In DBMS, the view serializability provides the user to view the database in a conflicting way.

In DBMS, we should understand schedules S1 and S2 to understand view serializability better. These two schedules should be created with the help of two transactions T1 and T2. To maintain the equivalent of the transaction each schedule has to obey the three transactions. These three conditions are as follows.

- The first condition is each schedule has the same type of transaction. The meaning of this condition is that both schedules S1 and S2 must not have the same type of set of transactions. If one schedule has committed the transaction but does not match the transaction of another schedule, then the schedule is not equivalent to each other.
- The second condition is that both schedules should not have the same type of read or write operation. On the other hand, if schedule S1 has two write operations while schedule S2 has one write operation, we say that both schedules are not equivalent to each other. We may also say that there is no problem if the number of the read operation is different, but there must be the same number of the write operation in both schedules.
- The final and last condition is that both schedules should not have the same conflict. Order of execution of the same data item. For example, suppose the transaction of schedule S1 is T1, and the transaction of schedule S2 is T2. The transaction T1 writes the data item A, and the transaction T2 also writes the data item A. in this case, the schedule is not equivalent to each other. But if the schedule has the same number of each write operation in the data item then we called the schedule equivalent to each other.

Testing of Serializability in DBMS with Examples

Serializability is a type of property of DBMS in which each transaction is executed independently and automatically, even though these transactions are executed concurrently. In other words, we can say that if there are several transactions executed concurrently, then the main work of the serializability function is to arrange these several transactions in a sequential manner.

For better understanding, let's explain these with an example. Suppose there are two users Sona and Archita. Each executes two transactions. Let's transactions T1 and T2 are executed by Sona, and T3 and T4 are executed by Archita. Suppose transaction T1 reads and writes the data item A, transaction T2 reads the data item B, transaction T3 reads and writes the data item C and transaction T4 reads the data item D. Lets the schedule the above transaction as below.

- T1: Read A → Write A→ Read B → Write B`

- `T2: Read B → Write B`

- T3: Read C → Write C→ Read D → Write D`
- T4: Read D → Write D

Let's first discuss why these transactions are not serializable.

In order for a schedule to be considered serializable, it must first satisfy the conflict serializability property. In our example schedule above, notice that Transaction 1 (T1) and Transaction 2 (T2) read data item B before either writing it. This causes a conflict between T1 and T2 because they are both trying to read and write the same data item concurrently. Therefore, the given schedule does not conflict with serializability.

However, there is another type of serializability called view serializability which our example does satisfy. View serializability requires that if two transactions cannot see each other's updates (i.e., one transaction cannot see the effects of another concurrent transaction), the schedule is considered to view serializable. In our example, Transaction 2 (T2) cannot see any updates made by Transaction 4 (T4) because they do not share common data items. Therefore, the schedule is viewed as serializable.

It's important to note that conflict serializability is a stronger property than view serializability because it requires that all potential conflicts be resolved before any updates are made (i.e., each transaction must either read or write each data item before any other transaction can write it). View serializability only requires that if two transactions cannot see each other's updates, then the schedule is view serializable & it doesn't matter whether or not there are potential conflicts between them.

All in all, both properties are necessary for ensuring correctness in concurrent transactions in a database management system.

Benefits of Serializability in DBMS

Below are the benefits of using the serializable in the database.

1. **Predictable execution:** In serializable, all the threads of the DBMS are executed at one time. There are no such surprises in the DBMS. In DBMS, all the variables are updated as expected, and there is no data loss or corruption.
2. **Easier to Reason about & Debug:** In DBMS all the threads are executed alone, so it is very easier to know about each thread of the database. This can make the debugging process very easy. So we don't have to worry about the concurrent process.
3. **Reduced Costs:** With the help of serializable property, we can reduce the cost of the hardware that is being used for the smooth operation of the database. It can also reduce the development cost of the software.

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.

LOCK-BASED PROTOCOL

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

1. Shared lock:

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

2. Exclusive lock:

- In the exclusive lock, the data item can be both reads as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

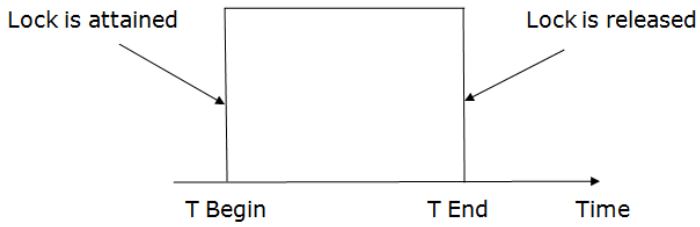
There are four types of lock protocols available:

1. Simplistic lock protocol

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

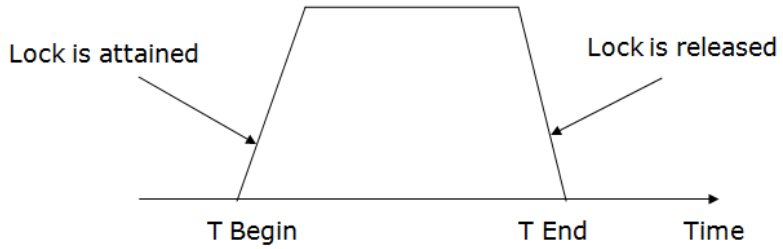
2. Pre-claiming Lock Protocol

- Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
- Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.
- If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
- If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.



3. Two-phase locking (2PL)

- The two-phase locking protocol divides the execution phase of the transaction into three parts.
- In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
- In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
- In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.



There are two phases of 2PL:

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

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

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

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

Example:

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

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

Transaction T1:

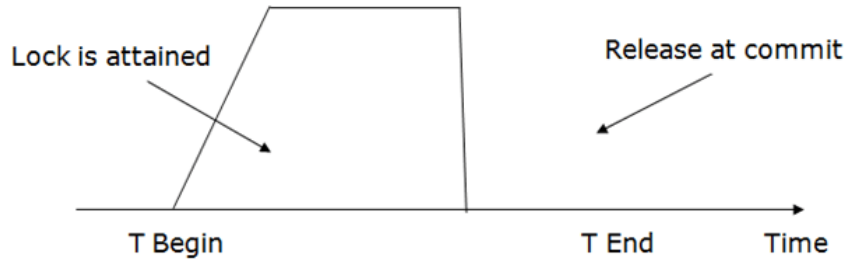
- **Growing phase:** from step 1-3
- **Shrinking phase:** from step 5-7
- **Lock point:** at 3

Transaction T2:

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

4. Strict Two-phase locking (Strict-2PL)
   - The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
   - The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
   - Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.

- Strict-2PL protocol does not have shrinking phase of lock release.

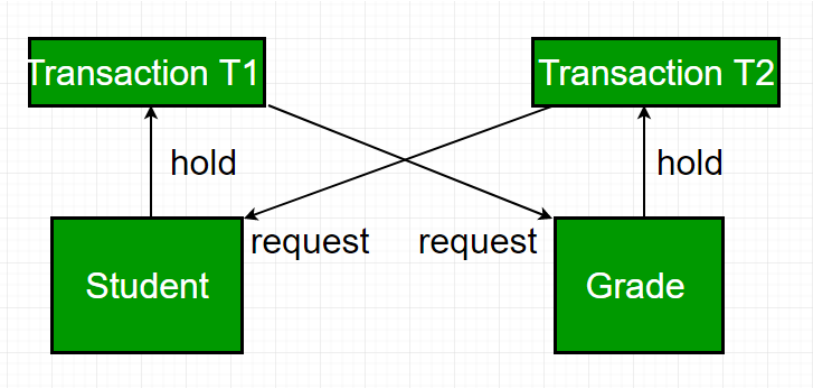


It does not have cascading abort as 2PL does.

Deadlock

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."

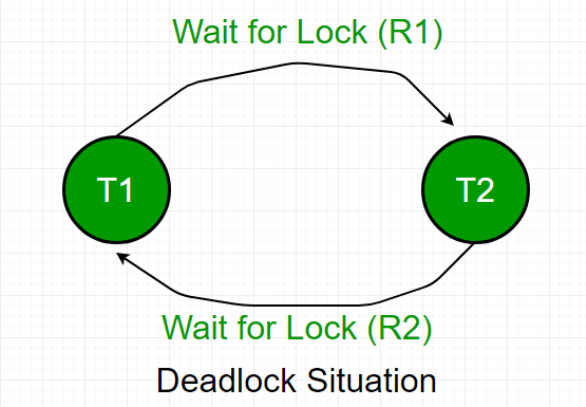
In a database, a deadlock is an unwanted situation in which two or more transactions are waiting indefinitely for one another to give up locks. Deadlock is said to be one of the most feared complications in DBMS as it brings the whole system to a Halt. **Example** – let us understand the concept of Deadlock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

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.

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 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:

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.
Lost Transactions: In some cases, deadlocks can cause transactions to be lost or aborted, which can result in data inconsistencies or other issues.
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.
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.

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:

Mutual Exclusion: Each resource can be held by only one transaction at a time, and other transactions must wait for it to be released.
Hold and Wait: Transactions can request resources while holding on to resources already allocated to them.
No Preemption: Resources cannot be taken away from a transaction forcibly, and the transaction must release them voluntarily.
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.
Indefinite Blocking: Transactions are blocked indefinitely, waiting for resources to become available, and no transaction can proceed.
System Stagnation: Deadlock leads to system stagnation, where no transaction can proceed, and the system is unable to make any progress.
Inconsistent Data: Deadlock can lead to inconsistent data if transactions are unable to complete and leave the database in an intermediate state.
Difficult to Detect and Resolve: Deadlock can be difficult to detect and resolve, as it may involve multiple transactions, resources, and dependencies.

Disadvantages:

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

RECOVERY WITH CONCURRENT TRANSACTION

- Whenever more than one transaction is being executed, then the interleaved of logs occur. During recovery, it would become difficult for the recovery system to backtrack all logs and then start recovering.
- To ease this situation, 'checkpoint' concept is used by most DBMS.

Isolation levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system. A transaction isolation level is defined by the following phenomena:

- **Dirty Read –** A Dirty read is a situation when a transaction reads data that has not yet been committed. For example, Let's say transaction 1 updates a row and leaves it uncommitted, meanwhile, Transaction 2 reads the updated row. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.
- **Non Repeatable read –** Non Repeatable read occurs when a transaction reads the same row twice and gets a different value each time. For example, suppose transaction T1 reads data. Due to concurrency, another transaction T2 updates the same data and commit, Now if transaction T1 rereads the same data, it will retrieve a different value.
- **Phantom Read –** Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different. For example, suppose transaction T1 retrieves a set of rows that satisfy some search criteria. Now, Transaction T2 generates some new rows that match the search criteria for transaction T1. If transaction T1 re-executes the statement that reads the rows, it gets a different set of rows this time.

Based on these phenomena, The SQL standard defines four isolation levels:

1. **Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transactions, thereby allowing dirty reads. At this level, transactions are not isolated from each other.
2. **Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allow dirty read. The transaction holds a read or write lock on the current row, and thus prevents other transactions from reading, updating, or deleting it.
3. **Repeatable Read –** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on referenced rows for update and delete actions. Since other transactions cannot read, update or delete these rows, consequently it avoids non-repeatable read.
4. **Serializable –** This is the highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

The Table given below clearly depicts the relationship between isolation levels, read phenomena, and locks:

Isolation Level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	May occur	May occur	May occur
Read Committed	Don't occur	May occur	May occur
Repeatable Read	Don't occur	Don't occur	May occur
Serializable	Don't occur	Don't occur	Don't occur

Anomaly Serializable is not the same as Serializable. That is, it is necessary, but not sufficient that a Serializable schedule should be free of all three phenomena types.

Transaction isolation levels are used in database management systems (DBMS) to control the level of interaction between concurrent transactions.

The four standard isolation levels are:

Read Uncommitted: This is the lowest level of isolation where a transaction can see uncommitted changes made by other transactions. This can result in dirty reads, non-repeatable reads, and phantom reads.

Read Committed: In this isolation level, a transaction can only see changes made by other committed transactions. This eliminates dirty reads but can still result in non-repeatable reads and phantom reads.

Repeatable Read: This isolation level guarantees that a transaction will see the same data throughout its duration, even if other transactions commit changes to the data. However, phantom reads are still possible.

Serializable: This is the highest isolation level where a transaction is executed as if it were the only transaction in the system. All transactions must be executed sequentially, which ensures that there are no dirty reads, non-repeatable reads, or phantom reads.