Unit - 4 Transaction Management

Transaction Concept, Transaction State, Implementation of Atomicity and Durability, Concurrent
Executions, Serializability, Recoverability, Implementation of Isolation, Testing for serializability, Lock Based Protocols, Timestamp Based Protocols, Validation- Based Protocols, Multiple Granularity, Recovery and Atomicity, Log–Based Recovery, Recovery with Concurrent Transactions.

****

Transaction Concept:

Transactions are a set of operations used to perform a logical set of work. It is the bundle of all the instructions of a logical operation. A transaction usually means that the data in the database has changed. One of the major uses of DBMS is to protect the user's data from system failures. It is done by ensuring that all the data is restored to a consistent state when the computer is restarted after a crash. The transaction is any one execution of the user program in a DBMS. One of the important properties of the transaction is that it contains a finite number of steps. Executing the same program multiple times will generate multiple transactions.

Example: Consider the following example of transaction operations to be performed to withdraw cash from an ATM vestibule.

Steps for ATM Transaction

1. Transaction Start.
2. Insert your ATM card.
3. Select a language for your transaction.
4. Select the Savings Account option.
5. Enter the amount you want to withdraw.
6. Enter your secret pin.
7. Wait for some time for processing.
8. Collect your Cash.
9. Transaction Completed.

A transaction can include the following basic database access operation.

- **Read/Access data (R):** Accessing the database item from disk (where the database stored data) to memory variable.
- **Write/Change data (W): Write** the data item from the memory variable to the disk.
- **Commit:** Commit is a transaction control language that is used to permanently save the changes done in a transaction

Example: Transfer of 50₹ from Account A to Account B. Initially A= 500₹, B= 800₹. This data is brought to RAM from the Hard Disk.

```
R(A) -- 500      // Accessed from RAM.
A = A-50         // Deducting 50₹ from A.
W(A)--450        // Updated in RAM.
R(B) -- 800      // Accessed from RAM.
B=B+50           // 50₹ is added to B's Account.
W(B) --850       // Updated in RAM.
commit           // The data in RAM is taken back to the Hard Disk.
```

Note: The updated value of Account A = 450₹ and Account B = 850₹.

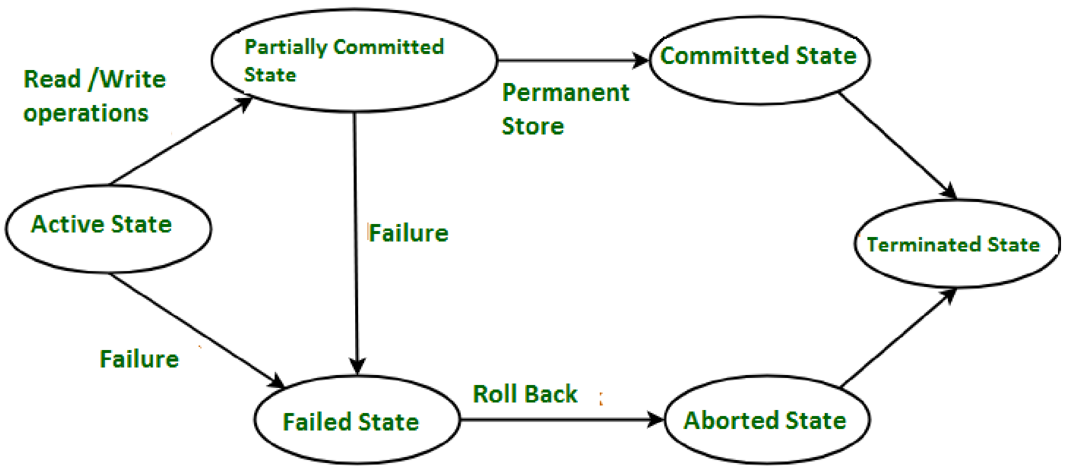
All instructions before committing come under a partially committed state and are stored in RAM. When the commit is read the data is fully accepted and is stored on the Hard Disk.

If the transaction is failed anywhere before committing we have to go back and start from the beginning. We can’t continue from the same state. This is known as Roll Back.

****

Transaction States

Transactions can be implemented using SQL queries and Servers. In the below-given diagram, you can see how transaction states work.



Transaction States in DBMS

Following are the different types of transaction States :

Active State:

When the operations of a transaction are running then the transaction is said to be in an active state. If all the *read and write* operations are performed without any error then it progresses to the *partially committed state*, if somehow any operation fails, then it goes to a state known as *failed state*.

Partially Committed:

After all the read and write operations are completed, the changes which were previously made in the main memory are now made permanent in the database, after which the state will progress to *committed state* but in case of a failure it will go to the *failed state*.

Failed State:

If any operation during the transaction fails due to some software or hardware issues, then it goes to the *failed state* . The occurrence of a failure during a transaction makes a permanent change to data in the database. The changes made into the local memory data are rolled back to the previous consistent state.

Aborted State:

If the transaction fails during its execution, it goes from *failed state* to *aborted state* and because in the previous states all the changes were only made in the main memory, these uncommitted changes are either deleted or rolled back. The transaction at this point can restart and start afresh from the active state.

Committed State:

If the transaction completes all sets of operations successfully, all the changes made during the partially committed state are permanently stored and the transaction is stated to be completed, thus the transaction can progress to finally get terminated in the *terminated state*.

Terminated State:

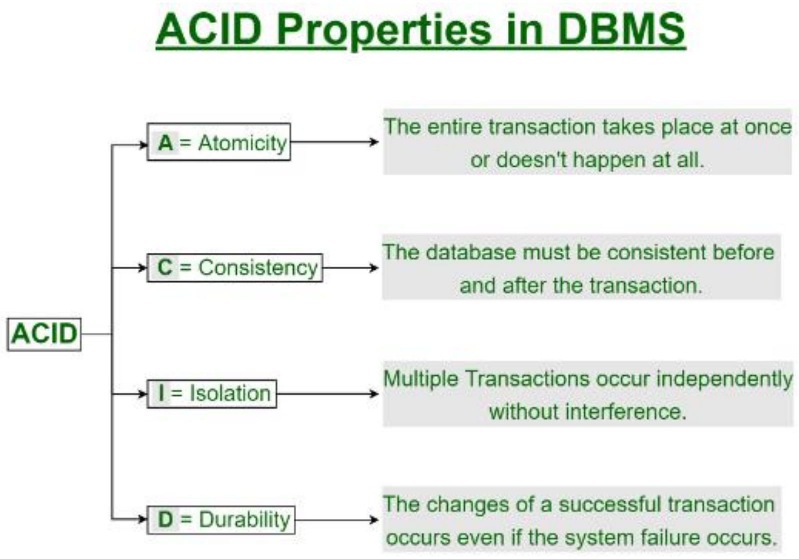
If the transaction gets aborted after roll-back or the transaction comes from the *committed state*, then the database comes to a consistent state and is ready for further new transactions since the previous transaction is now terminated.

****

ACID Properties /Transaction Properties in DBMS

A **transaction** is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.

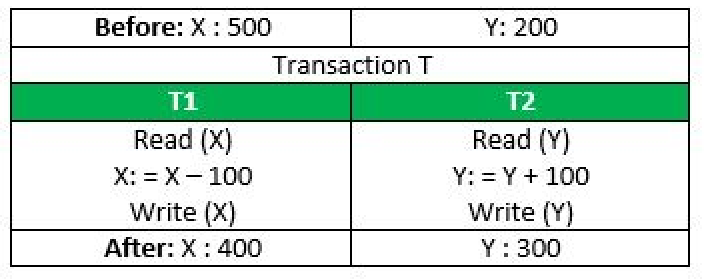
In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.


ACID Properties in DBMS
ACID
A = Atomicity
The entire transaction takes place at once or doesn't happen at all.
C = Consistency
The database must be consistent before and after the transaction.
I = Isolation
Multiple Transactions occur independently without interference.
D = Durability
The changes of a successful transaction occurs even if the system failure occurs.

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 the 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

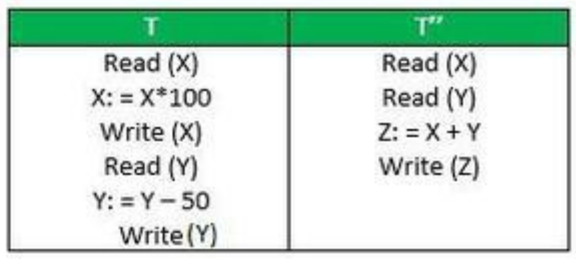
If the transaction fails after completion of **T1** but before completion of **T2**.(say, after **write(X)** but before **write(Y)**), then the amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in its entirety in order to ensure the correctness of the 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, the database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result, T is incomplete.

Isolation:

This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the 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 if these were executed serially in some order.
Let **X**= 500, **Y** = 500.
Consider two transactions **T** and **T".**



T	T''
Read (X)	Read (X)
X: = X*100	Read (Y)
Write (X)	Z: = X + Y
Read (Y)	Write (Z)
Y: = Y – 50	
Write (Y)	

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 the correct value of **X** but the 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 the 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 and modifications 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.

Some important points:

Property	**Responsibility for maintaining properties**
Atomicity	Transaction Manager
Consistency	Application programmer
Isolation	Concurrency Control Manager
Durability	Recovery Manager

The **ACID** properties, in totality, provide a mechanism to ensure the correctness and consistency of a database in a way such that each transaction is a group of operations that acts as a single unit, produces consistent results, acts in isolation from other operations, and updates that it makes are durably stored.

Advantages of ACID Properties in DBMS:

1. **Data Consistency:** ACID properties ensure that the data remains consistent and accurate after any transaction execution.
2. **Data Integrity:** ACID properties maintain the integrity of the data by ensuring that any changes to the database are permanent and cannot be lost.
3. **Concurrency Control**: ACID properties help to manage multiple transactions occurring concurrently by preventing interference between them.
4. **Recovery:** ACID properties ensure that in case of any failure or crash, the system can recover the data up to the point of failure or crash.

Disadvantages of ACID Properties in DBMS:

1. **Performance:** The ACID properties can cause a performance overhead in the system, as they require additional processing to ensure data consistency and integrity.
2. **Scalability:** The ACID properties may cause scalability issues in large distributed **systems where multiple transactions occur concurrently.**

3. **Complexity:** Implementing the ACID properties can increase the complexity of the system and require significant expertise and resources.
   Overall, the advantages of ACID properties in DBMS outweigh the disadvantages. They provide a reliable and consistent approach to data
4. management, ensuring data integrity, accuracy, and reliability. However, in some cases, the overhead of implementing ACID properties can cause performance and scalability issues. Therefore, it’s important to balance the benefits of ACID properties against the specific needs and requirements of the system.

****

Implementation of Atomicity and Durability in DBMS

Atomicity and durability are two important concepts in database management systems (DBMS) that ensure the consistency and reliability of data.

Atomicity:

One of the key characteristics of transactions in database management systems (DBMS) is atomicity, which guarantees that every operation within a transaction is handled as a single, indivisible unit of work.

Importance:

A key characteristic of transactions in database management systems is atomicity (DBMS). It makes sure that every action taken as part of a transaction is handled as a single, indivisible item of labor that can either be completed in full or not at all.

Even in the case of mistakes, failures, or crashes, atomicity ensures that the database maintains consistency. The following are some of the reasons why atomicity is essential in DBMS:

- **Consistency:** Atomicity ensures that the database remains in a consistent state at all times. All changes made by a transaction are rolled back if it is interrupted or fails for any other reason, returning the database to its initial state. By doing this, the database's consistency and data integrity are maintained.

- **Recovery:** Atomicity guarantees that, in the event of a system failure or crash, the database can be restored to a consistent state. All changes made by a transaction are undone if it is interrupted or fails, and the database is then reset to its initial state using the undo log. This guarantees that, even in the event of failure, the database may be restored to a consistent condition.
- **Concurrency:** Atomicity makes assurance that transactions can run simultaneously without affecting one another. Each transaction is carried out independently of the others, and its modifications are kept separate. This guarantees that numerous users can access the database concurrently without resulting in conflicts or inconsistent data.
- **Reliability:** Even in the face of mistakes or failures, atomicity makes the guarantee that the database is trustworthy. By ensuring that transactions are atomic, the database remains consistent and reliable, even in the event of system failures, crashes, or errors.

Implementation of Atomicity:

A number of strategies are used to establish atomicity in DBMS to guarantee that either all operations inside a transaction are correctly done or none of them are executed at all.

Techniques to Implement Atomicity in DBMS:

Here are some common techniques used to implement atomicity in DBMS:

- **Undo Log:** An undo log is a mechanism used to keep track of the changes made by a transaction before it is committed to the database. If a transaction fails, the undo log is used to undo the changes made by the transaction, effectively rolling back the transaction. By doing this, the database is guaranteed to remain in a consistent condition.
- **Redo Log:** A redo log is a mechanism used to keep track of the changes made by a transaction after it is committed to the database. If a system failure occurs after a transaction is committed but before its changes are written to disk, the redo log can be used to redo the changes and ensure that the database is consistent.

- **Two-Phase Commit:** Two-phase commit is a protocol used to ensure that all nodes in a distributed system commit or abort a transaction together. This ensures that the transaction is executed atomically across all nodes and that the database remains consistent across the entire system.
- **Locking:** Locking is a mechanism used to prevent multiple transactions from accessing the same data concurrently. By ensuring that only one transaction can edit a specific piece of data at once, locking helps to avoid conflicts and maintain the consistency of the database.

Durability:

One of the key characteristics of transactions in database management systems (DBMS) is durability, which guarantees that changes made by a transaction once it has been committed are permanently kept in the database and will not be lost even in the case of a system failure or catastrophe.

Importance:

Durability is a critical property of transactions in database management systems (DBMS) that ensures that once a transaction is committed, its changes are permanently stored in the database and will not be lost, even in the event of a system failure or crash. The following are some of the reasons why durability is essential in DBMS:

❖ **Data Integrity:** Durability ensures that the data in the database remains consistent and accurate, even in the event of a system failure or crash. It guarantees that committed transactions are durable and will be recovered without data loss or corruption.

❖ **Reliability:** Durability guarantees that the database will continue to be dependable despite faults or failures. In the event of system problems, crashes, or failures, the database is kept consistent and trustworthy by making sure that committed transactions are durable.

❖ **Recovery:** Durability guarantees that, in the event of a system failure or crash, the database can be restored to a consistent state. The database can be restored to a consistent state if a committed transaction is lost due to a system failure or crash since it can be recovered from the redo log or other backup storage.

- **Availability:** Durability ensures that the data in the database is always available for access by users, even in the event of a system failure or crash. It ensures that committed transactions are always retained in the database and are not lost in the event of a system crash.

Implementation of Durability in DBMS:

The implementation of durability in DBMS involves several techniques to ensure that committed changes are durable and can be recovered in the event of failure.

Techniques to Implement Durability:

Here are some common techniques used to implement durability in DBMS:

- **Write-Ahead Logging:** Write-ahead logging is a mechanism used to ensure that changes made by a transaction are recorded in the redo log before they are written to the database. This makes sure that the changes are permanent and that they can be restored from the redo log in the event of a system failure.
- **Checkpointing:** Checkpointing is a technique used to periodically write the database state to disk to ensure that changes made by committed transactions are permanently stored. Checkpointing aids in minimizing the amount of work required for database recovery.
- **Redundant storage:** Redundant storage is a technique used to store multiple copies of the database or its parts, such as the redo log, on separate disks or systems. This ensures that even in the event of a disk or system failure, the data can be recovered from the redundant storage.
- **RAID:** In order to increase performance and reliability, a technology called RAID (Redundant Array of Inexpensive Disks) is used to integrate several drives into a single logical unit. RAID can be used to implement redundancy and ensure that data is durable even in the event of a disk failure.

Here are Some Common Techniques used by DBMS to Implement Atomicity and Durability:

- **Transactions:** Transactions are used to group related operations that need to be executed atomically. They are either committed, in which case all their changes become permanent, or rolled back, in which case none of their changes are made permanent.
- **Logging:** Logging is a technique that involves recording all changes made to the database in a separate file called a log. The log is used to recover the database in case of a failure. Write-ahead logging is a common technique that guarantees that data is written to the log before it is written to the database.
- **Shadow copy:**

  Shadow copy is a technique that involves making a copy of the database before any changes are made. The copy is used to provide a consistent view of the database in case of failure. The modifications are made to the original database after a transaction has been committed.

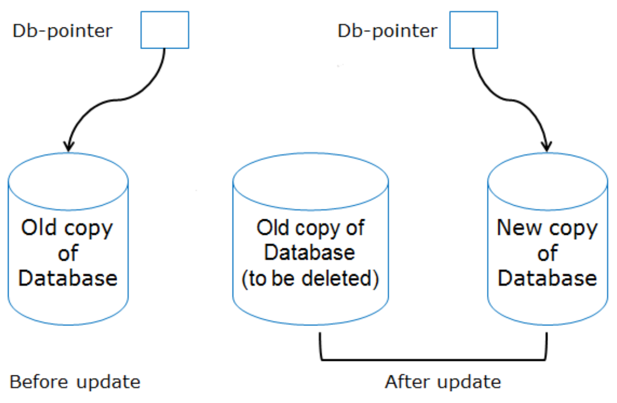
- In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.
- This scheme is based on making copies of the database, called shadow copies, and assumes that only one transaction is active at a time. The scheme also assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk; it points to the current copy of the database.

If the transaction completes, it is committed as follows:

- First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk. (Unix systems use the flush command for this purpose.)

- ❖ After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database. The old copy of the database is then deleted.

Figure below depicts the scheme, showing the database state before and after the update.



Shadow-copy technique for atomicity and durability

The transaction is said to have been committed at the point where the updated db pointer is written to disk.

How the technique handles transaction failures:

- ❖ If the transaction fails at any time before db-pointer is updated, the old contents of the database are not affected.
- ❖ We can abort the transaction by just deleting the new copy of the database.
- ❖ Once the transaction has been committed, all the updates that it performed are in the database pointed to by db pointer.
- ❖ Thus, either all updates of the transaction are reflected, or none of the effects are reflected, regardless of transaction failure.

How the technique handles system failures:

- ❖ Suppose that the system fails at any time before the updated db-pointer is written to disk. Then, when the system restarts, it will read db-pointer and will thus see the original contents of the database, and none of the effects of the transaction will be visible on the database.
- ❖ Next, suppose that the system fails after db-pointer has been updated on disk. Before the pointer is updated, all updated pages of the new copy of the database were written to disk.
- ❖ Again, we assume that, once a file is written to disk, its contents will not be damaged even if there is a system failure. Therefore, when the system restarts, it will read db-pointer and will thus see the contents of the database after all the updates performed by the transaction.

- **Backup and Recovery:** In order to guarantee that the database can be recovered to a consistent state in the event of a failure, backup and recovery procedures are used. This involves making regular backups of the database and keeping track of changes made to the database since the last backup.

****

Serializability in DBMS

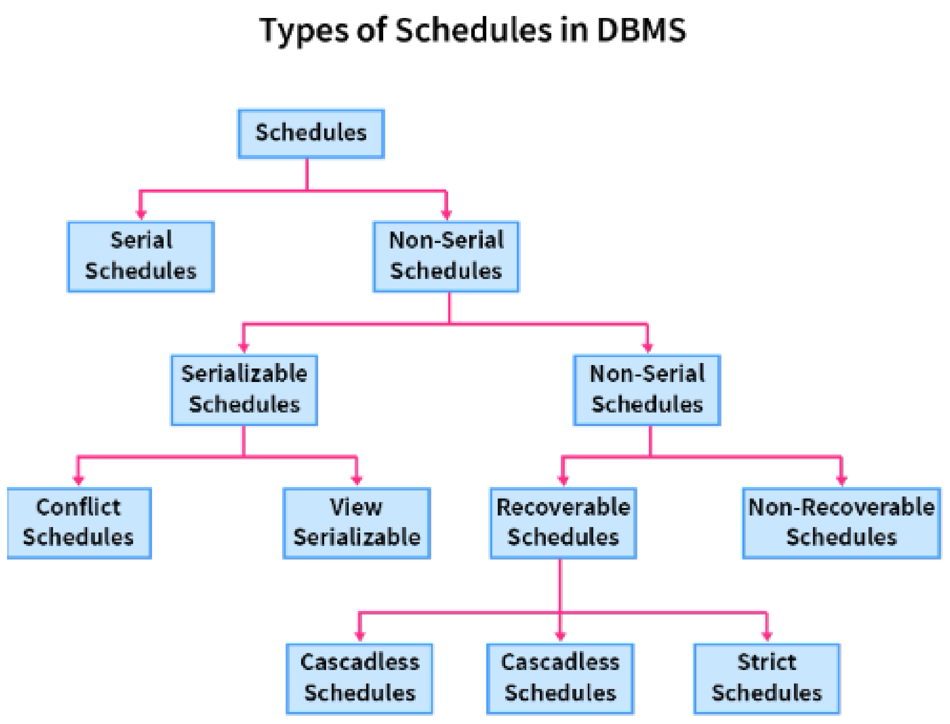
What is Serializability?

Serializability of schedules ensures that a non-serial schedule is equivalent to a serial schedule. It helps in maintaining the transactions to execute simultaneously without interleaving one another. In simple words, serializability is a way to check if the execution of two or more transactions are maintaining the database consistency or not.

Schedules and Serializable Schedules in DBMS

Schedules in DBMS are a series of operations performing one transaction to the other.

R(X) means Reading the value: X; and W(X) means Writing the value: X.



Schedules in DBMS are of two types:

1. **Serial Schedule -** A schedule in which only one transaction is executed at a time, i.e., one transaction is executed completely before starting another transaction.

Example:

Transaction-1	**Transaction-2**
R(a)	
W(a)	
R(b)	
W(b)	

	R(b)
	W(b)
	R(a)
	W(a)

Here, we can see that Transaction-2 starts its execution after the completion of Transaction-1.

NOTE:

Serial schedules are always serializable because the transactions only work one after the other. Also, for a transaction, there are n! serial schedules possible (where n is the number of transactions).

2. **Non-serial Schedule -** A schedule in which the transactions are interleaving or interchanging. There are several transactions executing simultaneously as they are being used in performing real-world database operations. These transactions may be working on the same piece of data. Hence, the serializability of non-serial schedules is a major concern so that our database is consistent before and after the execution of the transactions.

Example:

Transaction-1	**Transaction-2**
R(a)	
W(a)	
	R(b)
	W(b)
R(b)	
	R(a)
W(b)	

	W(a)

We can see that Transaction-2 starts its execution before the completion of Transaction-1, and they are interchangeably working on the same data, i.e., "a" and "b".

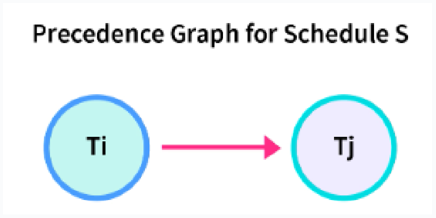
What is a serializable schedule?

A non-serial schedule is called a serializable schedule if it can be converted to its equivalent serial schedule. In simple words, if a non-serial schedule and a serial schedule result in the same then the non-serial schedule is called a serializable schedule.

Testing of Serializability

To test the serializability of a schedule, we can use Serialization Graph or Precedence Graph. A serialization Graph is nothing but a Directed Graph of the entire transactions of a schedule.

It can be defined as a Graph G(V, E) consisting of a set of directed-edges E = {E1, E2, E3, ..., En} and a set of vertices V = {V1, V2, V3, ...,Vn}. The set of edges contains one of the two operations - READ, WRITE performed by a certain transaction.



Ti -> Tj, means Transaction-Ti is either performing read or write before the transaction-Tj.

NOTE:

If there is a cycle present in the serialized graph then the schedule is non-serializable because the cycle resembles that one transaction is dependent on the other transaction and vice versa. It also means that there are one or more conflicting pairs of operations in the transactions. On the other hand, no-cycle means that the non-serial schedule is serializable.

What is a conflicting pair in transactions?

Two operations inside a schedule are called conflicting if they meet these three conditions:

1. They belong to two different transactions.
2. They are working on the same data piece.
3. One of them is performing the WRITE operation.

To conclude, let’s take two operations on data: "a". The conflicting pairs are:

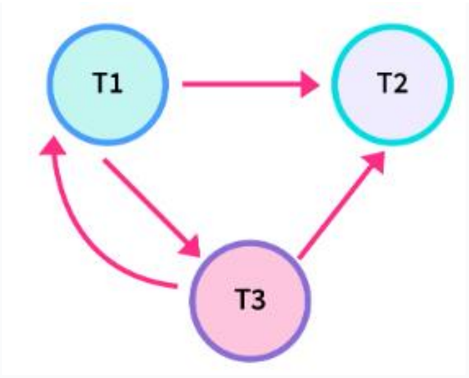
1. READ(a) - WRITE(a)
2. WRITE(a) - WRITE(a)
3. WRITE(a) - READ(a)

Note:

There can never be a read-read conflict as there is no change in the data.

Let's take an example of schedule "S" having three transactions t1, t2, and t3 working simultaneously

t1	**t2**	**t3**
R(x)		
		R(z)
		W(z)
	R(y)	
R(y)		
	W(y)	
		W(x)
	W(z)	
W(x)		



Non-serializable schedule. R(x) of T1 conflicts with W(x) of T3, so there is a directed edge from T1 to T3. R(y) of T1 conflicts with W(y) of T2, so there is a directed edge from T1 to T2. W(y\x) of T3 conflicts with W(x) of T1, so there is a directed edge from T3 to T. Similarly, we will make edges for every conflicting pair. Now, as the cycle is formed, the transactions cannot be serializable.

Types of Serializability

Serializability of any non-serial schedule can be verified using two types mainly:

1. Conflict Serializability
2. View Serializability.

What is Conflict Serializability in DBMS?

Conflict Serializability checks if a non-serial schedule is conflict serializable or not. A non-serial schedule is conflict serializable if it can convert into a serial schedule by swapping its non-conflicting operations.

Conflicting Operations

A pair of operations are said to be conflicting operations if they follow the set of conditions given below:

- Each operation is a part of different transactions.
- Both operations get performed on the same data item.

- One of the performed must be a write operation.

Let's consider two different transactions, *Ti* and *Tj* .Then considering the above conditions, the following table follows:

Transaction i	**Transaction j**	**IsConflicting**
Readi (X)	Readj (X)	Non-conflicting
Readi (X)	Writej (X)	Conflicting
Writei (X)	Readj (X)	Conflicting
Writei (X)	Writej (X)	Conflicting

Example

Let's see an example based on the following schedule.

Transaction 1	**Transaction 2**
R1(A)	W2(B)
W1(A)	
	R2(A)
R1(B)	W2(A)

In the above schedule, we can notice that:

- W1(A) and R2(A) are part of different transactions.
- Both apply to the same data item, i.e., A.
- W1(A) is a write operation.

W1(A) and R2(A) are conflicting operations as they satisfy all the above conditions.

Similarly, W1(A) and W2(A) are conflicting operations as they are part of different transactions working on the same data item, and one of them is the write operation.

W1(A) and W2(B) are non-conflicting operations as they work on different data items and thus do not satisfy all the given conditions.

R1(A) and R2(A) are non-conflicting operations as none of them is a write operation and thus does not satisfy the third condition.

W1(A) and R1(A) are non-conflicting as they belong to the same transactions and thus do not satisfy the first condition.

Conflict Equivalent

If a schedule gets converted into another schedule by swapping the non-conflicting operations, they are said to be conflict equivalent schedules.

Checking Whether a Schedule is Conflict Serializable Or Not

A non-serial schedule gets checked for conflict serializability using the following steps:

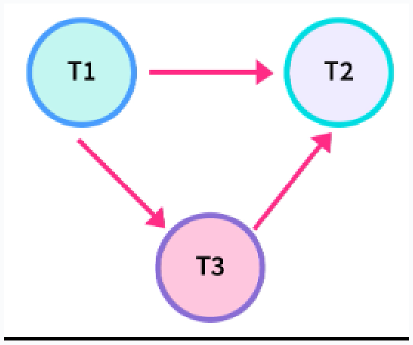
1. Figure out all the conflicting operations and enlist them.
2. Create a precedence graph. For every transaction in the schedule, draw a node in the precedence graph.
3. If Xi(A) and Yj(A) represent a conflict pair, then draw an edge from Ti to Tj for each conflict pair. The precedence graph ensures that Ti gets executed before Tj.
4. The next step involves checking for any cycle formed in the graph. A schedule is a conflict serializable if there is no cycle present.

Example:

We have a schedule "S" having three transactions t1, t2, and t3 working simultaneously. Let's form a precedence graph.

t1	**t2**	**t3**
R(x)		

	R(y)	
		R(y)
	W(y)	
W(x)		
		W(x)
	R(x)	
	W(x)	



As there is no loop in the graph, it is a conflict serializable schedule as well as a serial schedule. Since it is a serial schedule, we can detect the order of transactions as well.

The order of the Transactions: t1 will execute first as there is no incoming edge on T1. t3 will execute second as it depends on T1 only. t2 will execute at last as it depends on both T1 and T3.

So, order of its equivalent serial schedule is: t1 --> t3 --> t2

NOTE:

If a schedule is conflicting serializable, then it is surely a consistent schedule. On the other hand, a non-conflicting serializable schedule may or may not be serial. To further check its serial behavior, we use the concept of View Serializability.

View Serializability in DBMS

What is View Serializability?

A system handles multiple transactions running concurrently, there is a possibility that a few of them leaves the system databases in an inconsistent state and thus, the system fails. To be aware of such schedules and save the system from failures, we check if schedules are serializable or not. If the given schedule is serializable implies it will never leave the database in an inconsistent state.

Serial schedules are the schedules where no two transactions are executed concurrently. Thus, serial schedules never leave the database in an inconsistent state. If the given schedule is view serializable, then we are sure that it is a consistent schedule or serial schedule.

View Serializability is a process used to check whether the given schedule is view serializable or not. To do so we check if the given schedule is View Equivalent to its serial schedule.

What is view equivalency?

The two conditions needed by schedules (S1 and S2) to be view equivalent are:

1. Initial read must be on the same piece of data.

   Example: If transaction t1 is reading "A" from database in schedule S1, then in schedule S2, t1 must read A.

2. Final write must be on the same piece of data.

   Example: If a transaction t1 updated A at last in S1, then in S2, t1 should perform final write as well.

3. The mid sequence should also be in the same order.

   Example: If t1 is reading A which is updated by t2 in S1, then in S2, t1 should read A which should be updated by t2.

This process of checking view equivalency of a schedule is called View Serializability.

Example

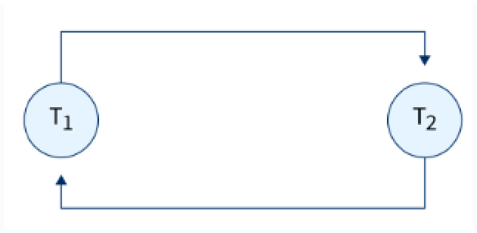
Let us understand View-Serializability with an example of a schedule S~1~.

T1	**T2**
R(X)	
X=X+10	
W(A)	
	R(X)
	X=X+10
	W(X)
R(Y)	
Y=Y+20	
W(Y)	
	R(Y)
	Y=Y*1.1
	W(Y)

Checking conflict serializability Precedence Graph of S1:

- R1(X), W2(X) (T1 → T2)
- R2(X), W1(X) (T2 → T1)
- W1(X), W2(X) (T1 → T2)
- R1(Y), W2(Y) (T1 → T2)
- R2(Y), W1(Y) (T2 → T1)
- W1(Y), W2(Y) (T1 → T2)

The conflict precedence graph is as follows:

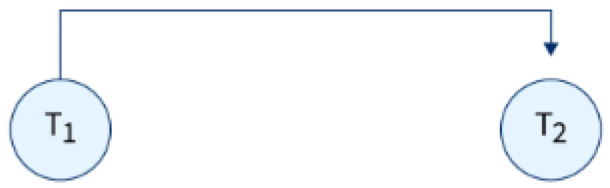


The above graph contains cycle/loop thus, it is not conflict serializable but by only considering the precedence graph we can't conclude if the given schedule is consistent or not.

In the above example if we swap a few transaction operations, and format the table as follows:

T1	T2
R(X)	
X=X+10	
W(A)	
R(Y)	
Y=Y+20	
W(Y)	
	R(X)
	X=X+10
	W(X)
	R(Y)
	Y=Y*1.1
	W(Y)

Now the precedence graph is as follows:



The precedence graph doesn't contain any cycle or loop which implies the given schedule is conflict serializable and the result will be the same as that of the first table.

Note: If a schedule is conflict serializable, then it will also be view-serializable, consistent as well as equivalent to a serial schedule. The view serializable schedule may or may not be conflict serializable but it is consistent as well.

But if the given schedule does not conflict serializable, then we need to check for View Serializability. If it is view serializable then it is consistent else it is inconsistent. Thus, to address the limitations of conflict serializability, the concept of view-serializability was introduced.

Methods to Check View Serializability

There are two methods to check View Serializability.

Method 1: View Equivalent

If the given schedule is view-equivalent to the serial schedule itself then we can say that the schedule is view-serializable.

To generate the serial schedule of the given schedule, we follow the following steps:

- Take the transaction say 'Ti' that performs any operation first in the schedule. Write all the operations performed by that transaction one after the other in the given schedule.
- Similarly, take another transaction that is performing any operation right after Ti in the given schedule and write all the operations performed by it below transaction Ti on any data item.

- Follow the above steps for all the transactions. The order is very important.

Let us take the example of the table for checking**View Serializability**.

Let us call this Schedule S1

T1	**T2**
R(X)	
X=X+10	
W(A)	
	R(X)
	X=X+10
	W(X)
R(Y)	
Y=Y+20	
W(Y)	
	R(Y)
	Y=Y*1.1
	W(Y)

Also, for the above table the serial schedule is as follows:

Let us call this Schedule S2

T1	**T2**
R(X)	
X=X+10	
W(A)	

R(Y)	
Y=Y+20	
W(Y)	
	R(X)
	X=X+10
	W(X)
	R(Y)
	Y=Y*1.1
	W(Y)

As transaction T1 performed the first operation we wrote all of its operations on data item X as well as Y first and then all the operations of T2 were written next.

Two schedules are said to view equivalent if they satisfy the following three conditions:

Condition 1: Initial Read

Initially if transaction T1 of Schedule S1 reads data item X, then transaction T1 of Schedule S2 must read data item X initially as well.

In the example above, the Initial read is made by the T1 transaction for both the data items X and Y in Schedule S1 and in S2 as well initial read is made by transaction T1 for both. Thus, the initial read condition is satisfied in S1 and S2 for data items X and Y.

Condition 2: Update Read

If transaction Ti is reading updated data item X by transaction Tj in S1, then in Schedule S2 Ti should read X which is updated by Tj.

Value of X and Y is updated by transaction T1 in schedule S1, which is read by transaction T2. In Schedule S2 the X and Y data items are updated by T1 and are read by T2 as well. Thus, in the above two schedules, the update read condition is also satisfied.

Condition 3: Final Write

If transaction T1 updates data item Y at last in S1, then in S2 also, T1 should perform the final write operation on Y.

The final write for data items X and Y are made by Transaction T2 in Schedule S1. Also, the final write is made by T2 in Schedule S2 in Schedule S1 an well as serial schedule S2. Thus, the condition for the final write is also satisfied.

Result

As all the three conditions are satisfied for our schedule S1 and its Serial Schedule S2, they are view equivalent. Thus, we can say that the given schedule S1 is the view-serializable schedule and is consistent.

Method 2: Check the View-Serializability of a Schedule

There's another method to check the view-serializability of a schedule. The first step of this method is the same as the previous method i.e. checking the conflict serializability of the schedule by creating the precedence graph.

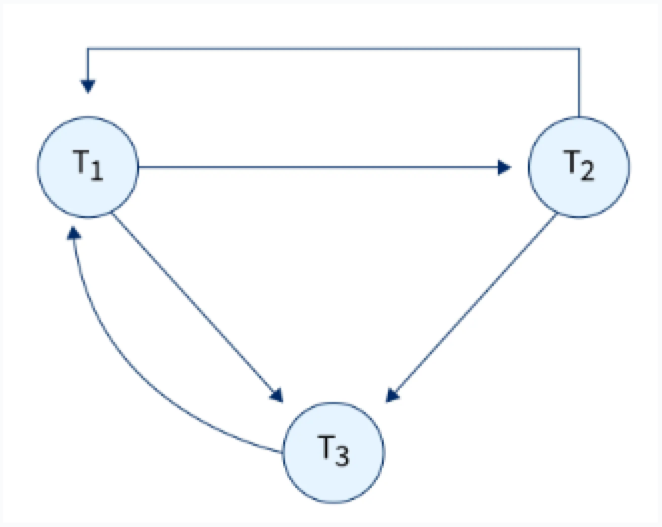
Let us take an example of the schedule as follows to check the View Serializability

T1	**T2**	**T3**
R(X)		
	W(X)	
		R(X)
W(X)		
		W(X)

Writing all the conflicting operations:

- R1(X), W2(X) (T1 → T2)
- R1(X), W3(X) (T1 → T3)
- W2(X), R3(X) (T2 → T3)
- W2(X), W1(X) (T2 → T1)
- W2(X), W3(X) (T2 → T3)
- R3(X), W1(X) (T3 → T1)
- W1(X), W3(X) (T1 → T3)

The precedence graph for the above schedule is as follows:



If the precedence graph doesn't contain a loop/cycle, then it is conflict serializable, and thus concludes that the given schedule is consistent. We are not required to perform the View Serializability test as a conflict serializable schedule is view-serializable as well.

As the above graph contains a loop/cycle, it does not conflict with serializable. Thus we need to perform the following steps -

Next, we will check for blind writes. If the blind writes don't exist, then the schedule is non-view Serializable. We can simply conclude that the given schedule is inconsistent.

If there exist any blind writes in the schedule, then it may or may not be view serializable.

Blind writes: If a write action is performed on a data item by a Transaction (updation), without performing the reading operation then it is known as blind write.

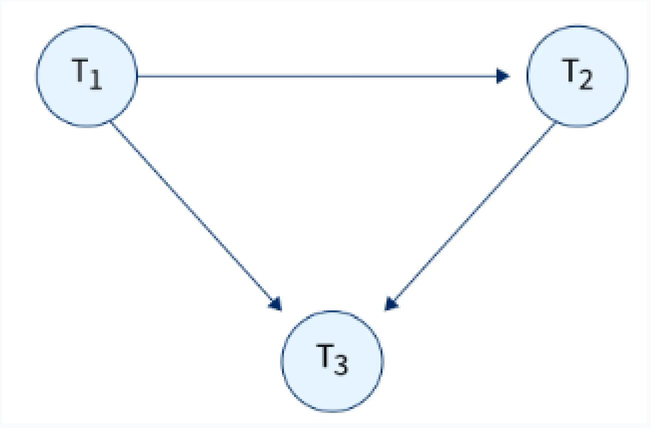
In the above example, transaction T2 contains a blind write, thus we are not sure if the given schedule is view-serializable or not.

Lastly, we will draw a dependency graph for the schedule. Note: The dependency graph is different from the precedence graph.

Steps for dependency graph:

- Firstly, T1 reads X, and T2 first updates X. So, T1 must execute before the T2 operation. (T1 → T2)
- T3 performs the final updation on X thus, T3 must execute after T1 and T2. (T1 → T3 and T2 → T3)
- T2 updates X before T3, so we get the dependency as (T2 → T3)

The dependency graph is formed as follows:



As no cycles exist in the dependency graph for the example we can say that the schedule is view-serializable.

If any cycle/ loop exists, then it is not view-serializable and the schedule is inconsistent. If cycle//loop doesn't exist, then it is view-serializable.

****

Recoverability in DBMS

Recoverability refers to the ability of a system to restore its state to a point where the integrity of its data is not compromised, especially after a failure or an error.

When multiple transactions are executing concurrently, issues may arise that affect the system's recoverability. The interaction between transactions, if not managed correctly, can result in scenarios where a transaction's effects cannot be undone, which would violate the system's integrity.

Importance of Recoverability:

The need for recoverability arises because databases are designed to ensure data reliability and consistency.

1. Recoverable Schedules

A schedule is said to be recoverable if, for any pair of transactions

Ti and Tj, if Tj reads a data item previously written by Ti, then Ti must commit before Tj commits. If a transaction fails for any reason and needs to be rolled back, the system can recover without having to rollback other transactions that have read or used data written by the failed transaction.

Example of a Recoverable Schedule

Suppose we have two transactions T1 and T2

```
| Transaction T1       | Transaction T_2      |
|--------------------|--------------------|
| Write(A)             |                      |
|                      | Read(A)              |
| Commit               |                      |
|                      | Write(B)             |
|                      | Commit               |
```

In the above schedule, T2 reads a value written by T1, but T1 commits before T2, making the schedule recoverable.

2. Non-Recoverable Schedules

A schedule is said to be non-recoverable (or irrecoverable) if there exists a pair of transactions

Ti and Tj such that Tj reads a data item previously written by Ti , but Ti has not committed yet and Tj commits before Ti. If ‘Ti’ fails and needs to be rolled back after Tj has committed, there's no straightforward way to roll back the effects of Tj, leading to potential data inconsistency.

Example of a Non-Recoverable Schedule

Again, consider two transactions T1 and T2.

```
| Transaction T1       | Transaction T2       |
|--------------------|--------------------|
| Write(A)             |                      |
|                      | Read(A)              |
|                      | Write(B)             |
|                      | Commit               |
| Commit               |                      |
```

In this schedule, T2 reads a value written by T1 and commits before T1 does. If T1 encounters a failure and has to be rolled back after T2 has committed, we're left in a problematic situation since we cannot easily roll back T2, making the schedule non-recoverable.

3. Cascading Rollback

A cascading rollback occurs when the rollback of a single transaction causes one or more dependent transactions to be rolled back. This situation can arise when one transaction reads uncommitted changes of another transaction, and then the latter transaction fails and needs to be rolled back. Consequently, any transaction that has read the uncommitted changes of the failed transaction also needs to be rolled back, leading to a cascade effect.

Example of Cascading Rollback

Consider two transactions T1 and T2

```
| Transaction T1       | Transaction T2       |
|---------------------|---------------------|
| Write(A)             |                      |
|                      | Read(A)              |
|                      | Write(B)             |
| Abort(some failure)  |                      |
| Rollback             |                      |
|                      | Rollback (because it read uncommitted changes from T1)
```

Here, T2 reads an uncommitted value of A written by T1. When T1 fails and is rolled back, T2 also has to be rolled back, leading to a cascading rollback. This is undesirable because it wastes computational effort and can complicate recovery procedures.

4. Cascadeless Schedules

A schedule is considered cascadeless if transactions only read committed values. This means, in such a schedule, a transaction can read a value written by another transaction only after the latter has committed. Cascadeless schedules prevent cascading rollbacks.

Example of Cascadeless Schedule

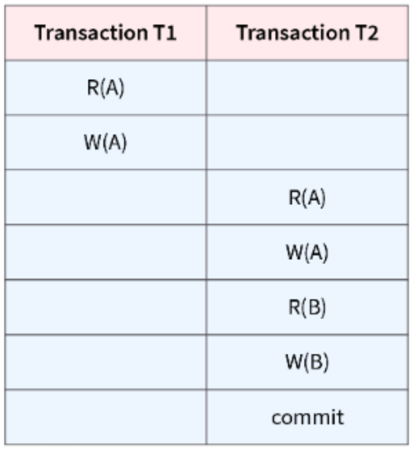
Consider two transactions T1 and T2

```
| Transaction T1       | Transaction T2       |
|----------------------|----------------------|
| Write(A)             |                      |
| Commit               |                      |
|                      | Read(A)              |
|                      | Write(B)             |
|                      | Commit               |
```

In this schedule, T2 reads the value of A only after T1has committed. Thus, even if T1 were to fail before committing (not shown in this schedule), it would not affect T2. This means there's no risk of cascading rollback in this schedule.

Non-Recoverable Schedule

If a transaction reads the value of an operation from an uncommitted transaction and commits before the transaction from where it has read the value, then such a schedule is called Non-Recoverable schedule. A non-recoverable schedule means when there is a system failure, we may not be able to recover to a consistent database state. If the commit operation of Ti doesn't occur before the commit operation of Tj, it is non-recoverable.



Transaction T1	Transaction T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	commit

Consider the following Schedule involving two transactions T1 and T2. T2 read the value of A written by T1, and committed. T1 might later abort/commit; therefore, the value read by T2 is wrong, but since T2 committed, this Schedule is non-recoverable.

****

Implementation of Isolation in DBMS

Isolation is one of the core ACID properties of a database transaction, ensuring that the operations of one transaction remain hidden from other transactions until completion. It means that no two transactions should interfere with each other and affect the other's intermediate state.

Isolation Levels

Isolation levels defines the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system. There are four levels of transaction isolation defined by SQL -

1. Serializable

- The highest isolation level.
- Guarantees full serializability and ensures complete isolation of transaction operations.

2. Repeatable Read

- This is the most restrictive isolation level.
- The transaction holds read locks on all rows it references.
- It holds write locks on all rows it inserts, updates, or deletes.
- Since other transaction cannot read, update or delete these rows, it avoids non repeatable read.

3. Read Committed

- This isolation level allows only committed data to be read.
- Thus it does not allow dirty read (i.e. one transaction reading of data immediately after written by another transaction).
- The transaction holds a read or write lock on the current row, and thus prevent other rows from reading, updating or deleting it.

4. Read Uncommitted

- It is the lowest isolation level.
- In this level, one transaction may read not yet committed changes made by another transaction.
- This level allows dirty reads.

The proper isolation level or concurrency control mechanism to use depends on the specific requirements of a system and its workload. Some systems may prioritize high throughput and can tolerate lower isolation levels, while others might require strict consistency and higher isolation.

Isolation level	**Dirty Read**	**Unrepetable Read**
Serializable	NO	NO
Repeatable Read	NO	NO
Read Committed	NO	Maybe
Read Uncommitted	Maybe	Maybe

****

Concurrent Execution in DBMS

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

Concurrency Control Problems

Several problems that arise when numerous transactions execute simultaneously in a random manner are referred to as Concurrency Control Problems.

Dirty Read Problem

The dirty read problem in DBMS occurs when *a transaction reads the data that has been updated by another transaction that is still uncommitted.* It arises due to multiple uncommitted transactions executing simultaneously.

Example: Consider two transactions A and B performing read/write operations on a data DT in the database DB. The current value of DT is 1000: The following table shows the read/write operations in A and B transactions.

Time	**A**	**B**
T1	READ(DT)	------

T2	DT=DT+500	------
T3	WRITE(DT)	------
T4	------	READ(DT)
T5	------	COMMIT
T6	ROLLBACK	------

Transaction A reads the value of data DT as 1000 and modifies it to 1500 which gets stored in the temporary buffer. The transaction B reads the data DT as 1500 and commits it and the value of DT permanently gets changed to 1500 in the database DB. Then some server errors occur in transaction A and it wants to get rollback to its initial value, i.e., 1000 and then the dirty read problem occurs.

Unrepeatable Read Problem

The unrepeatable read problem occurs when two or more different values of the same data are read during the read operations in the same transaction.

Example: Consider two transactions A and B performing read/write operations on a data DT in the database DB. The current value of DT is 1000: The following table shows the read/write operations in A and B transactions.

Time	**A**	**B**
T1	READ(DT)	------
T2	------	READ(DT)
T3	DT=DT+500	------
T4	WRITE(DT)	------
T5	------	READ(DT)

Transaction A and B initially read the value of DT as 1000. Transaction A modifies the value of DT from 1000 to 1500 and then again transaction B reads the value and finds it to be 1500. Transaction B finds two different values of DT in its two different read operations.

Phantom Read Problem

In the phantom read problem, data is read through two different read operations in the same transaction. In the first read operation, a value of the data is obtained but in the second operation, an error is obtained saying the data does not exist.

Example: Consider two transactions A and B performing read/write operations on a data DT in the database DB. The current value of DT is 1000: The following table shows the read/write operations in A and B transactions.

Time	**A**	**B**
T1	READ(DT)	------
T2	------	READ(DT)
T3	DELETE(DT)	------
T4	------	READ(DT)

Transaction B initially reads the value of DT as 1000. Transaction A deletes the data DT from the database DB and then again transaction B reads the value and finds an error saying the data DT does not exist in the database DB.

Lost Update Problem

The Lost Update problem arises when an update in the data is done over another update but by two different transactions.

Example: Consider two transactions A and B performing read/write operations on a data DT in the database DB. The current value of DT is 1000: The following table shows the read/write operations in A and B transactions.

Time	**A**	**B**
T1	READ(DT)	------

T2	DT=DT+500	------
T3	WRITE(DT)	------
T4	------	DT=DT+300
T5	------	WRITE(DT)
T6	READ(DT)	------

Transaction A initially reads the value of DT as 1000. Transaction A modifies the value of DT from 1000 to 1500 and then again transaction B modifies the value to 1800. Transaction A again reads DT and finds 1800 in DT and therefore the update done by transaction A has been lost.

Incorrect Summary Problem

The Incorrect summary problem occurs when there is an incorrect sum of the two data. This happens when a transaction tries to sum two data using an aggregate function and the value of any one of the data get changed by another transaction.

Example: Consider two transactions A and B performing read/write operations on two data DT1 and DT2 in the database DB. The current value of DT1 is 1000 and DT2 is 2000: The following table shows the read/write operations in A and B transactions.

Time	**A**	**B**
T1	READ(DT1)	------
T2	add=0	------
T3	add=add+DT1	------
T4	------	READ(DT2)
T5	------	DT2=DT2+500
T6	READ(DT2)	------
T7	add=add+DT2	------

Transaction A reads the value of DT1 as 1000. It uses an aggregate function SUM which calculates the sum of two data DT1 and DT2 in variable add but in between the value of DT2 get changed from 2000 to 2500 by transaction B. Variable add uses the modified value of DT2 and gives the resultant sum as 3500 instead of 3000.

****

Concurrency Control Protocols

To avoid concurrency control problems and to maintain consistency and serializability during the execution of concurrent transactions some rules are made. These rules are known as Concurrency Control Protocols.

There are 3 types of Concurrency Control Protocols

1. Lock-Based Protocols
2. Time-based Protocols
3. Validation Based Protocol

Lock-Based Protocol

Why Do We Need Locks?

Locks are essential in a database system to ensure:

1. **Consistency**: Without locks, multiple transactions could modify the same data item simultaneously, resulting in an inconsistent state.
2. **Isolation**: Locks ensure that the operations of one transaction are isolated from other transactions, i.e., they are invisible to other transactions until the transaction is committed.
3. **Concurrency**: While ensuring consistency and isolation, locks also allow multiple transactions to be processed simultaneously by the system, optimizing system throughput and overall performance.
4. **Avoiding Conflicts**: Locks help in avoiding data conflicts that might arise due to simultaneous read and write operations by different transactions on the same data item.

5. **Preventing Dirty Reads:** With the help of locks, a transaction is prevented from reading data that hasn't yet been committed by another transaction.

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:

Shared lock:

- It is also known as a Read-only lock. In a shared lock, the data item can only be 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.

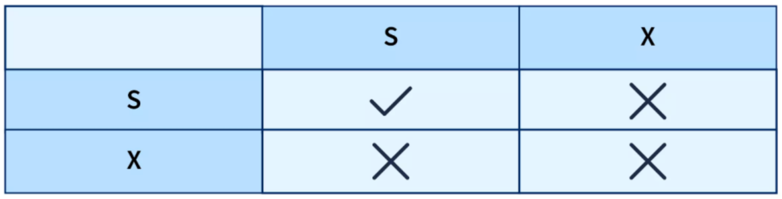
Exclusive lock:

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

Lock Compatibility Matrix

A vital point to remember when using Lock-based protocols in Database Management System is that a Shared Lock can be held by any amount of transactions. On the other hand, an Exclusive Lock can only be held by one transaction in DBMS, this is because a shared lock only reads data but does not perform any other activities, whereas an exclusive lock performs read as well as writing activities.

The figure given below demonstrates that when two transactions are involved, and both of these transactions seek to read a specific data item, the transaction is authorized, and no conflict occurs; but, in a situation when one transaction intends to write the data item and another transaction attempts to read or write simultaneously, the interaction is rejected.



	S	X
S	✓	✗
X	✗	✗

The two methods outlined below can be used to convert between the locks:

1. Conversion from a read lock to a write lock is an upgrade.
2. Conversion from a write lock to a read lock is a downgrade.

There are four types of lock protocols available:

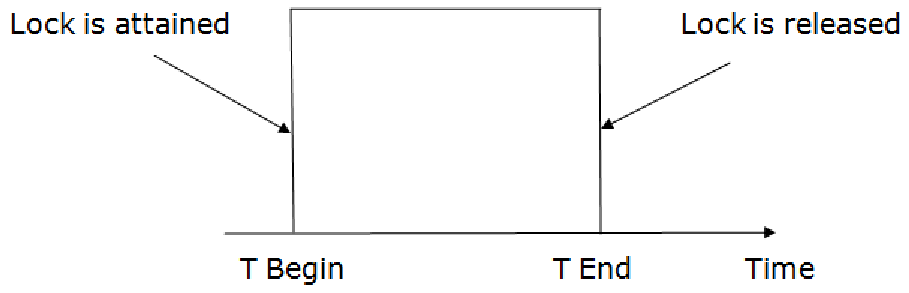
1. Simplistic lock protocol

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 locks 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 locks.

- If all the locks are not granted then this protocol allows the transaction to roll back and waits until all the locks are granted.



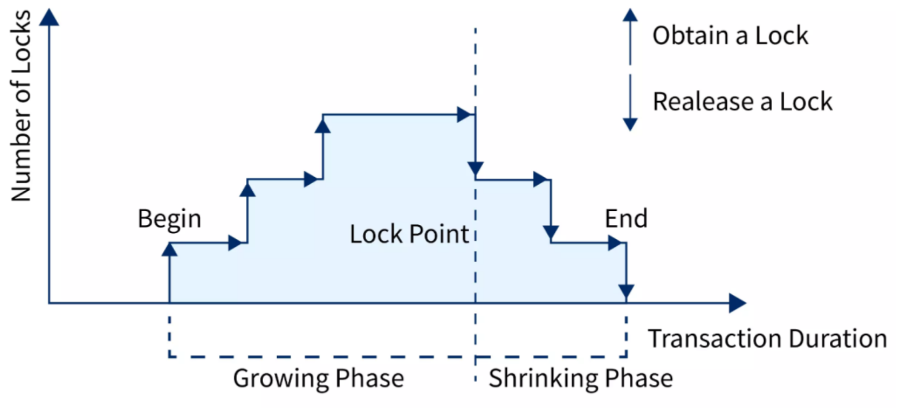
3. Two-phase locking (2PL)

- 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 locks 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 the growing phase.
2. Downgrading of lock (from X(a) to S(a)) must be done in a shrinking phase.

Two-phase locking helps to reduce the amount of concurrency in a schedule

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

Advantages of Two-Phase Locking (2PL)

- **Ensures Serializability:** 2PL guarantees conflict-serializability, ensuring the consistency of the database.
- **Concurrency:** By allowing multiple transactions to acquire locks and release them, 2PL increases the concurrency level, leading to better system throughput and overall performance.
- **Avoids Cascading Rollbacks:** Since a transaction cannot read a value modified by another uncommitted transaction, cascading rollbacks are avoided, making recovery simpler.

Disadvantages of Two-Phase Locking (2PL)

- **Deadlocks:** The main disadvantage of 2PL is that it can lead to deadlocks, where two or more transactions wait indefinitely for a resource locked by the other.
- **Reduced Concurrency (in certain cases):** Locking can block transactions, which can reduce concurrency. For example, if one transaction holds a lock for a long time, other transactions needing that lock will be blocked.

- **Overhead:** Maintaining locks, especially in systems with a large number of items and transactions, requires overhead. There's a time cost associated with acquiring and releasing locks, and memory overhead for maintaining the lock table.
- **Starvation:** It's possible for some transactions to get repeatedly delayed if other transactions are continually requesting and acquiring locks.

Starvation

Starvation is the situation when a transaction needs to wait for an indefinite period to acquire a lock.

Following are the reasons for Starvation:

- When waiting scheme for locked items is not properly managed
- In the case of resource leak
- The same transaction is selected as a victim repeatedly

Deadlock

Deadlock refers to a specific situation where two or more processes are waiting for each other to release a resource or more than two processes are waiting for the resource in a circular chain.

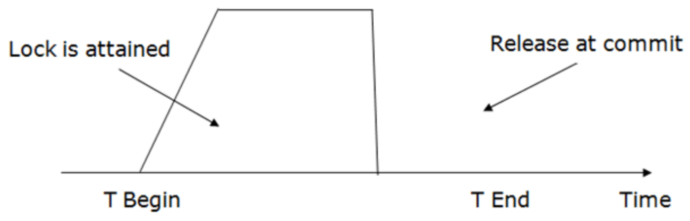
Categories of Two-Phase Locking in DBMS

1. Strict Two-Phase Locking
2. Rigorous Two-Phase Locking
3. Conservative (or Static) Two-Phase Locking:

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 a shrinking phase of lock release.



It does not have cascading abort as 2PL does.

Rigorous Two-Phase Locking

- A transaction can release a lock after using it, but it cannot commit until all locks have been acquired.
- Like strict 2PL, rigorous 2PL is deadlock-free and ensures serializability.

Conservative or Static Two-Phase Locking

- A transaction must request all the locks it will ever need before it begins execution. If any of the requested locks are unavailable, the transaction is delayed until they are all available.
- This approach can avoid deadlocks since transactions only start when all their required locks are available.

****

2.Timestamp-based Protocols

Timestamp based Protocol in DBMS is an algorithm which uses the System Time or Logical Counter as a timestamp to serialize the execution of concurrent transactions.

The Timestamp-based protocol ensures that every conflicting read and write operations are executed in a timestamp order.

The older transaction is always given priority in this method. It uses system time to determine the time stamp of the transaction. This is the most commonly used concurrency protocol.

Lock-based protocols help us to manage the order between the conflicting transactions when they will execute. Timestamp-based protocols manage conflicts as soon as an operation is created.

Example:

Suppose there are three transactions T1, T2, and T3.

T1 has entered the system at time 0010

T2 has entered the system at 0020

T3 has entered the system at 0030

Priority will be given to transaction T1, then transaction T2 and lastly Transaction T3.

Basic timestamp ordering protocol can be checked by the following conditions:

1. When a transaction T performs a Write _item(X) operation check the following conditions :

- If Read timestamp of data item X is greater than the Timestamp of the Transaction T i.e R_TS(X) > TS(T) or if Write Timestamp of the data item X is greater than the timestamp of the transaction i.e W_TS(X) > TS(T), then abort and rollback T and reject the operation. else,
- Execute the Write_ item(X) operation of T and set W_TS(X) to TS(T).

2. When a transaction T performs a Read_ item(X) operation check the following conditions :

- If Write Timestamp of data item X is greater than the timestamp of the transaction T i.e W_TS(X) > TS(T), then abort and reject T and reject the operation, else
- If Write Timestamp of data item X is less than or equal to the timestamp of the transaction T i.e W_TS(X) <= TS(T), then execute the R_item(X) operation of T and set R_TS(X) to the larger of TS(T) and current R_TS(X).

Whenever there is a conflicting operation that violates the timestamp ordering protocol then the later operation is rejected and the transaction is aborted. Schedules created by the Basic timestamp ordering protocol are conflict serializable and deadlock-free.

One of the drawbacks of the Basic timestamp ordering protocol is that cascading Rollback is possible in the timestamp ordering protocol. For Example, if transactions T1 and T2 use a value written by T1. If T1 aborts and rolls back before committing the transaction then T2 must also be aborted and rolled back. So cascading problems may occur in the Basic timestamp ordering protocol.

Strict Timestamp Ordering

strict timestamp ordering is a variation of basic timestamp ordering. Strict timestamp ordering ensures that the transaction is both strict and conflicts serializable. In Strict timestamp ordering a transaction T that issues a Read_ item(X) or Write_ item(X) such that TS(T) > W_TS(X) has its read or write operation delayed until the Transaction T‘that wrote the values of X has committed or aborted.

Thomas write Rule

Thomas Write Rule provides the guarantee of serializability order for the protocol. It improves the Basic Timestamp Ordering Algorithm.

The basic Thomas write rules are as follows:

- If TS(T) < R_TS(X) then transaction T is aborted and rolled back, and operation is rejected.
- If TS(T) < W_TS(X) then don't execute the W_ item(X) operation of the transaction and continue processing.
- If neither condition 1 nor condition 2 occurs, then allowed to execute the WRITE operation by transaction Ti and set W_TS(X) to TS(T).

Advantages

- Timestamp-based protocols ensure serializability as the transaction is ordered on their creation timestamp.
- No deadlock occurs when timestamp ordering protocol is used as no transaction waits.
- No older transaction waits for a longer period of time so the protocol is free from deadlock.
- Timestamp based protocol ensures that there are no conflicting items in the transaction execution.

Disadvantages

- Timestamp-based protocols may not be cascade free or recoverable.
- In timestamp based protocol there is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction.

****

3.Validation Based Protocol

Validation Based Protocol is also called Optimistic Concurrency Control Technique. This protocol is used in DBMS (Database Management System) for avoiding concurrency in transactions. It is called optimistic because of the assumption it makes, i.e. very less interference occurs, therefore, there is no need for checking while the transaction is executed.

In this technique, no checking is done while the transaction is being executed. Until the transaction end is reached updates in the transaction are not applied directly to the database. All updates are applied to local copies of data items kept for the transaction. At the end of transaction execution, while execution of the transaction, a **validation phase** checks whether any of the transaction updates violate serializability. If there is no violation of serializability the transaction is committed and the database is updated; or else, the transaction is updated and then restarted.

Optimistic Concurrency Control is a three-phase protocol. The three phases for validation based protocol:

1. **Read phase:** In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables. It can perform all the write operations on temporary variables without an update to the actual database.
2. **Validation phase:** In this phase, the temporary variable value will be validated against the actual data to see if it violates the serializability.
3. **Write phase:** If the validation of the transaction is validated, then the temporary results are written to the database or system otherwise the transaction is rolled back.

Here each phase has the following different timestamps:

Start(Ti): It contains the time when Ti started its execution.

Validation (T_i): It contains the time when Ti finishes its read phase and starts its validation phase.

Finish(Ti): It contains the time when Ti finishes its write phase.

- This protocol is used to determine the time stamp for the transaction for serialization using the timestamp of the validation phase, as it is the actual phase which determines if the transaction will commit or rollback.
- Hence TS(T) = validation(T).

- The serializability is determined during the validation process. It can't be decided in advance.
- While executing the transaction, it ensures a greater degree of concurrency and also less number of conflicts.
- Thus it contains transactions which have less number of rollbacks.

****

Multiple Granularity

Let's start by understanding the meaning of granularity.

Granularity: It is the size of data item allowed to lock.

Multiple Granularity:

- It can be defined as hierarchically breaking up the database into blocks which can be locked.
- The Multiple Granularity protocol enhances concurrency and reduces lock overhead.
- It maintains the track of what to lock and how to lock.
- It makes easy to decide either to lock a data item or to unlock a data item. This type of hierarchy can be graphically represented as a tree.

For example: Consider a tree which has four levels of nodes.

- The first level or higher level shows the entire database.
- The second level represents a node of type area. The higher level database consists of exactly these areas.
- The area consists of children nodes which are known as files. No file can be present in more than one area.
- Finally, each file contains child nodes known as records. The file has exactly those records that are its child nodes. No records represent in more than one file.

- Hence, the levels of the tree starting from the top level are as follows:
    - Database
    - Area
    - File
    - Record

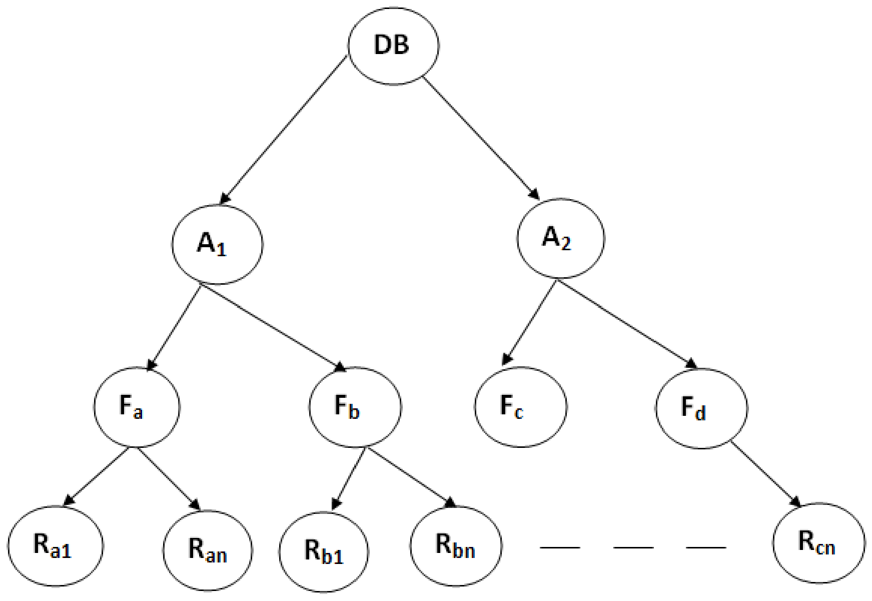


Figure: Multi Granularity tree Hierarchy

In this example, the highest level shows the entire database. The levels below are file, record, and fields.

There are three additional lock modes with multiple granularity:

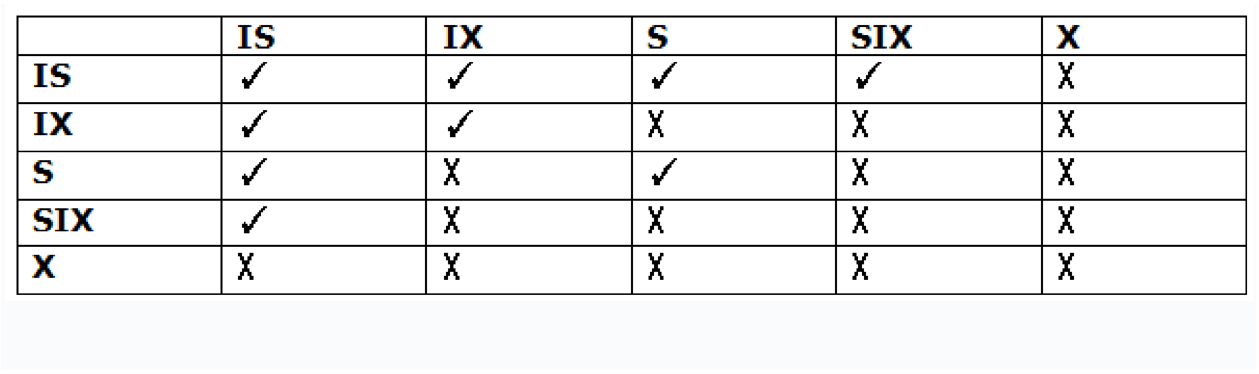
Intention Mode Lock

Intention-shared (IS): It contains explicit locking at a lower level of the tree but only with shared locks.

Intention-Exclusive (IX): It contains explicit locking at a lower level with exclusive or shared locks.

Shared & Intention-Exclusive (SIX): In this lock, the node is locked in shared mode, and some node is locked in exclusive mode by the same transaction.

Compatibility Matrix with Intention Lock Modes: The below table describes the compatibility matrix for these lock modes:



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	X
IX	✓	✓	X	X	X
S	✓	X	✓	X	X
SIX	✓	X	X	X	X
X	X	X	X	X	X

It uses the intention lock modes to ensure serializability. It requires that if a transaction attempts to lock a node, then that node must follow these protocols:

- Transaction T1 should follow the lock-compatibility matrix.
- Transaction T1 firstly locks the root of the tree. It can lock it in any mode.
- If T1 currently has the parent of the node locked in either IX or IS mode, then the transaction T1 will lock a node in S or IS mode only.
- If T1 currently has the parent of the node locked in either IX or SIX modes, then the transaction T1 will lock a node in X, SIX, or IX mode only.
- If T1 has not previously unlocked any node only, then the Transaction T1 can lock a node.
- If T1 currently has none of the children of the node-locked only, then Transaction T1 will unlock a node.

Observe that in multiple-granularity, the locks are acquired in top-down order, and locks must be released in bottom-up order.

- If transaction T1 reads record R_{a9} in file F_a, then transaction T1 needs to lock the database, area A_1 and file F_a in IX mode. Finally, it needs to lock R_{a2} in S mode.
- If transaction T2 modifies record R_{a9} in file F_a, then it can do so after locking the database, area A_1 and file F_a in IX mode. Finally, it needs to lock the R_{a9} in X mode.
- If transaction T3 reads all the records in file F_a, then transaction T3 needs to lock the database, and area A in IS mode. At last, it needs to lock F_a in S mode.
- If transaction T4 reads the entire database, then T4 needs to lock the database in S mode.

****

Recovery and Atomicity in DBMS

Introduction

Data may be monitored, stored, and changed rapidly and effectively using a DBMS (Database Management System).A database possesses atomicity, consistency, isolation, and durability qualities. The ability of a system to preserve data and changes made to data defines its durability. A database could fail for any of the following reasons:

- System breakdowns occur as a result of hardware or software issues in the system.
- Transaction failures arise when a certain process dealing with data updates cannot be completed.
- Disk crashes may occur as a result of the system's failure to read the disc.
- Physical damages include issues such as power outages or natural disasters.
- The data in the database must be recoverable to the state they were in prior to the system failure, even if the database system fails. In such situations, database recovery procedures in DBMS are employed to retrieve the data.

The recovery procedures in DBMS ensure the database's atomicity and durability. If a system crashes in the middle of a transaction and all of its data is lost, it is not regarded as durable. If just a portion of the data is updated during the transaction, it is not considered atomic. Data recovery procedures in DBMS make sure that the data is always recoverable to protect the durability property and that its state is retained to protect the atomic property. The procedures listed below are used to recover data from a DBMS,

- Recovery based on logs.
- Recovery through Deferred Update
- Immediate Recovery via Immediate Update

The atomicity attribute of DBMS safeguards the data state. If a data modification is performed, the operation must be completed entirely, or the data's state must be maintained as if the manipulation never occurred. This characteristic may be impacted by DBMS failure brought on by transactions, but DBMS recovery methods will protect it.

****

Log-Based Recovery

- The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.
- If any operation is performed on the database, then it will be recorded in the log.
- But the process of storing the logs should be done before the actual transaction is applied in the database.

There are two approaches to modify the database:

1. Deferred database modification:

- The deferred modification technique occurs if the transaction does not modify the database until it has committed.
- In this method, all the logs are created and stored in the stable storage, and the database is updated when a transaction commits.

2. Immediate database modification:

- The Immediate modification technique occurs if database modification occurs while the transaction is still active.

- In this technique, the database is modified immediately after every operation. It follows an actual database modification.

Recovery using Log records

When the system is crashed, then the system consults the log to find which transactions need to be undone and which need to be redone.

1. If the log contains the record <Ti, Start> and <Ti, Commit> or <Ti, Commit>, then the Transaction Ti needs to be redone.
2. If log contains record<T_n, Start> but does not contain the record either <Ti, commit> or <Ti, abort>, then the Transaction Ti needs to be undone.

****

Recovery with Concurrent Transactions

Concurrency control means that multiple transactions can be executed at the same time and then the interleaved logs occur. But there may be changes in transaction results so maintain the order of execution of those transactions.

During recovery, it would be very difficult for the recovery system to backtrack all the logs and then start recovering.

Recovery with concurrent transactions can be done in the following four ways.

1. Interaction with concurrency control
2. Transaction rollback
3. Checkpoints
4. Restart recovery

Interaction with concurrency control :

In this scheme, the recovery scheme depends greatly on the concurrency control scheme that is used. So, to rollback a failed transaction, we must undo the updates performed by the transaction.

Transaction rollback :

- In this scheme, we rollback a failed transaction by using the log.
- The system scans the log backward for a failed transaction, for every log record found in the log the system restores the data item.

Checkpoints :

- Checkpoints is a process of saving a snapshot of the applications state so that it can restart from that point in case of failure.
- Checkpoint is a point of time at which a record is written onto the database form the buffers.
- Checkpoint shortens the recovery process.
- When it reaches 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 the next checkpoint and so on.
- The checkpoint is used to declare the point before which the DBMS was in the consistent state, and all the transactions were committed.

To ease this situation, 'Checkpoints 'Concept is used by most DBMS.

- In this scheme, we used checkpoints to reduce the number of log records that the system must scan when it recovers from a crash.
- In a concurrent transaction processing system, we require that the checkpoint log record be of the form <checkpoint L>, where 'L' is a list of transactions active at the time of the checkpoint.
- A fuzzy checkpoint is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out.

Restart recovery :

- When the system recovers from a crash, it constructs two lists.
- The undo-list consists of transactions to be undone, and the redo-list consists of transactions to be redone.

- The system constructs the two lists as follows: Initially, they are both empty. The system scans the log backward, examining each record, until it finds the first <checkpoint> record.

****