**Unit-5**

**Database Transaction**

A Database Transaction is a logical unit of processing in a DBMS which entails one or more database access operation. In a nutshell, database transactions represent real-world events of any enterprise.

All types of database access operation which are held between the beginning and end transaction statements are considered as a single logical transaction in DBMS. During the transaction the database is inconsistent. Only once the database is committed the state is changed from one consistent state to another.

A **Database Transaction** is a logical unit of processing in a DBMS which entails one or more database access operation. In a nutshell, database transactions represent real-world events of any enterprise.

All types of database access operation which are held between the beginning and end transaction statements are considered as a single logical transaction in DBMS. During the transaction the database is inconsistent. Only once the database is committed the state is changed from one consistent state to another.

**Example of transaction:**

A simple example of a transaction will be dealing with the bank accounts of two users, let say Karlos and Ray. A simple transaction of moving an amount of 5000 from Karlos to Ray engages many low-level jobs. As the amount of Rs. 5000 gets transferred from the Karlos's account to Ray's account, a series of tasks gets performed in the background of the screen.

This straightforward and small transaction includes several steps: decrease Karlos's bank account from 5000:

Open\_Acc (Karlos)

OldBal = Karlos.bal

NewBal = OldBal - 5000

Ram.bal = NewBal

CloseAccount(Karlos)

You can say, the transaction involves many tasks, such as opening the account of Karlos, reading the old balance, decreasing the specific amount of 5000 from that account, saving new balance to an account of Karlos, and finally closing the transaction session.

For adding amount 5000 in Ray's account, the same sort of tasks needs to be done:

OpenAccount(Ray)

Old\_Bal = Ray.bal

NewBal = OldBal + 1000

Ahmed.bal = NewBal

CloseAccount(B)

## **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 A to 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**

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](https://beginnersbook.com/2015/04/acid-properties-in-dbms/).

**Facts about Database Transactions**

* A transaction is a program unit whose execution may or may not change the contents of a database.
* The transaction concept in DBMS is executed as a single unit.
* If the database operations do not update the database but only retrieve data, this type of transaction is called a read-only transaction.
* A successful transaction can change the database from one CONSISTENT STATE to another
* DBMS transactions must be atomic, consistent, isolated and durable
* If the database were in an inconsistent state before a transaction, it would remain in the inconsistent state after the transaction.

**Why do you need concurrency in Transactions?**

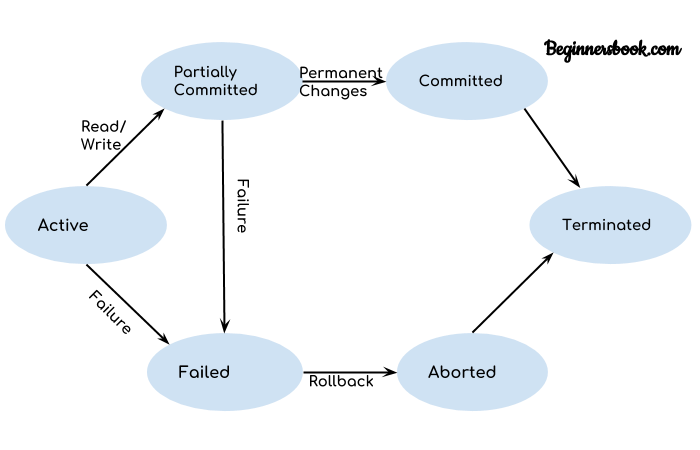
A database is a shared resource accessed. It is used by many users and processes concurrently. For example, the banking system, railway, and air reservations systems, stock market monitoring, supermarket inventory, and checkouts, etc.   
Not managing concurrent access may create issues like:

* Hardware failure and system crashes
* Concurrent execution of the same transaction, deadlock, or slow performance

## **States of Transactions**

The various states of a transaction concept in DBMS are listed below:

|  |  |
| --- | --- |
| **State** | **Transaction types** |
| Active State | A transaction enters into an active state when the execution process begins. During this state read or write operations can be performed. |
| Partially Committed | A transaction goes into the partially committed state after the end of a transaction. |
| Committed State | When the transaction is committed to state, it has already completed its execution successfully. Moreover, all of its changes are recorded to the database permanently. |
| Failed State | A transaction considers failed when any one of the checks fails or if the transaction is aborted while it is in the active state. |
| Terminated State | State of transaction reaches terminated state when certain transactions which are leaving the system can't be restarted. |



Let's study a [state transition diagram](https://www.guru99.com/state-transition-testing.html) that highlights how a transaction moves between these various states.

1. Once a transaction states execution, it becomes active. It can issue READ or WRITE operation.
2. Once the READ and WRITE operations complete, the transactions becomes partially committed state.
3. Next, some recovery protocols need to ensure that a system failure will not result in an inability to record changes in the transaction permanently. If this check is a success, the transaction commits and enters into the committed state.
4. If the check is a fail, the transaction goes to the Failed state.
5. If the transaction is aborted while it's in the active state, it goes to the failed state. The transaction should be rolled back to undo the effect of its write operations on the database.
6. The terminated state refers to the transaction leaving the system.

**Properties of Transaction:**

There are properties that all transactions should follow and possess. The four basic are in combination termed as ACID properties. ACID properties and its concepts of a transaction are put forwarded by Haerder and Reuter in the year 1983. The ACID has a full form and is as follows:

* **Atomicity:** The 'all or nothing' property. A transaction is an indivisible entity that is either performed in its entirety or will not get performed at all. This is the responsibility or duty of the recovery subsystem of the DBMS to ensure atomicity.
* **Consistency:** A transaction must alter the database from one steady-state to another steady state. This is the responsibility of both the DBMS and the application developers to make certain consistency. The DBMS can ensure consistency by putting into effect all the constraints that have been mainly on the database schema such as integrity and enterprise constraints.
* **Isolation:** Transactions that are executing independently of one another is the primary concept followed by isolation. In other words, the frictional effects of incomplete transactions should not be visible or come into notice to other transactions going on simultaneously. It is the responsibility of the concurrency control sub-system to ensure adapting the isolation.
* **Durability:** The effects of an accomplished transaction are permanently recorded in the database and must not get lost or vanished due to subsequent failure. So this becomes the responsibility of the recovery sub-system to ensure durability.

### **ACID Property in DBMS with example:**

Below is an example of ACID property in DBMS:

Transaction 1: Begin X=X+50, Y = Y-50 END

Transaction 2: Begin X=1.1\*X, Y=1.1\*Y END

Transaction 1 is transferring $50 from account X to account Y.

Transaction 2 is crediting each account with a 10% interest payment.

If both transactions are submitted together, there is no guarantee that the Transaction 1 will execute before Transaction 2 or vice versa. Irrespective of the order, the result must be as if the transactions take place serially one after the other.

**DBMS Transaction Property:**

DBMS Transaction Property describes a few significant features which each DBMS Transaction holds to maintain the integrity of the database server during this processing. A Transaction in DBMS is defined as a group of logically associated operations that proceeds through several states its life cycle and the process should be either fully completed or aborted, without any partial or transitional states. When there exists a successful transaction then there also occurs variations of a database from one reliable state to another, which includes all satisfied data integrity constraints in the server. In DBMS, a transaction is denoted as an action which either reads from or writes to a database holding the distinct and robust required properties such as atomicity, isolation, consistency, and durability. This properties of a DBMS transaction together is defined as the DBMS ACID properties.

**Syntax:**

There are three transaction types in DBMS is Base on Application Areas, Structure, and Action. Also, there are important transaction states as Active, Partially Committed, Failed, and lastly Terminate.

**Operations of Transaction to maintain the ACID properties:**

Subsequent are the fundamental operations of the DBMS Transaction to ensure the ACID properties:

* **Read(X)**: This read operation is functional to read the X’s value from the database server and preserves it in a buffer in the main memory.
* **Write(X)**: This write operation is functional to write the X’s value back to the database server from the buffer.

Let us view a simple syntax to show the debit transaction in DBMS from an account that encompasses underneath operational commands:

R(X);  
X = X – 2000;//Amount to be provided  
W(X);

# **DBMS Schedules**

We know that [transactions](https://beginnersbook.com/2017/09/transaction-management-in-dbms/) are set of instructions and these instructions perform operations on database. When multiple transactions are running concurrently then there needs to be a sequence in which the operations are performed because at a time only one operation can be performed on the database. This sequence of operations is known as **Schedule**.

## **DBMS Schedule example**

The following sequence of operations is a schedule. Here we have two transactions T1 & T2 which are running concurrently.This schedule determines the exact order of operations that are going to be performed on database. In this example, all the instructions of transaction T1 are executed before the instructions of transaction T2, however this is not always necessary.

T1 T2

---- ----

R(X)

W(X)

R(Y)

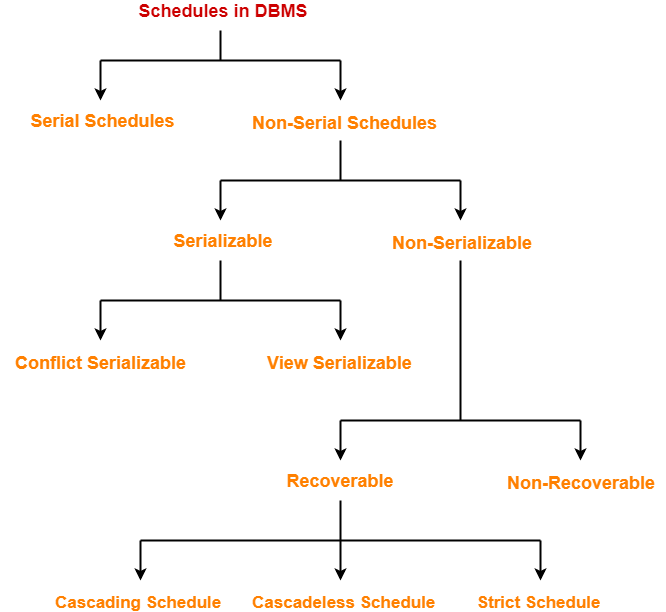
R(Y)

R(X)

W(Y)

## **Types of Schedules-**

In DBMS, schedules may be classified as-



## **Serial Schedules-**

In serial schedules,

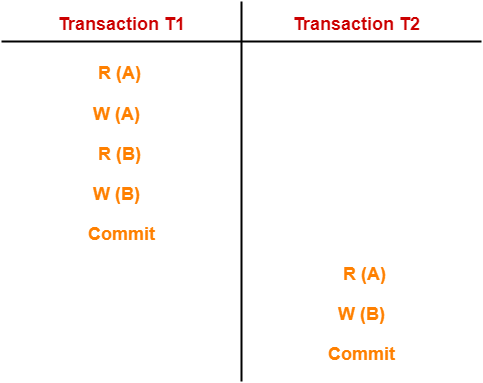
* All the transactions execute serially one after the other.
* When one transaction executes, no other transaction is allowed to execute.

## **Characteristics-**

Serial schedules are always-

* Consistent
* Recoverable
* Cascadeless
* Strict

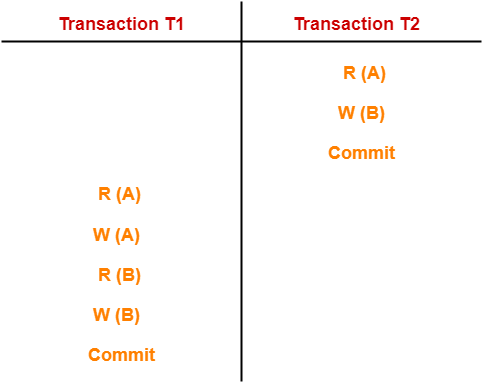
### **Example-01:**



In this schedule,

* There are two transactions T1 and T2 executing serially one after the other.
* Transaction T1 executes first.
* After T1 completes its execution, transaction T2 executes.
* So, this schedule is an example of a **Serial Schedule**.

### **Example-02:**



In this schedule,

* There are two transactions T1 and T2 executing serially one after the other.
* Transaction T2 executes first.
* After T2 completes its execution, transaction T1 executes.
* So, this schedule is an example of a **Serial Schedule**.

## **Non-Serial Schedules-**

In non-serial schedules,

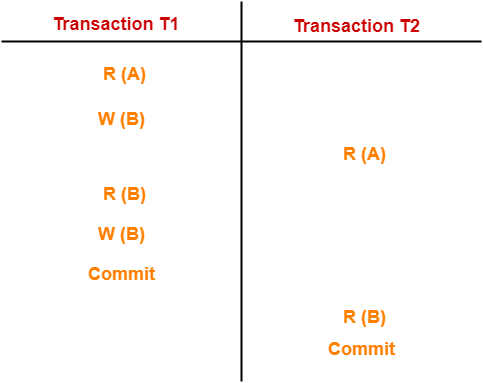
* Multiple transactions execute concurrently.
* Operations of all the transactions are inter leaved or mixed with each other.

## **Characteristics-**

Non-serial schedules are **NOT** always-

* Consistent
* Recoverable
* Cascadeless
* Strict

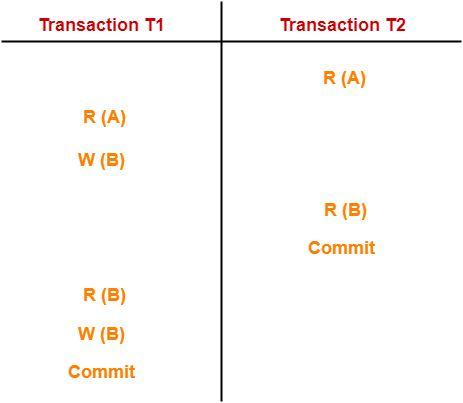
### **Example-01:**



In this schedule,

* There are two transactions T1 and T2 executing concurrently.
* The operations of T1 and T2 are interleaved.
* So, this schedule is an example of a **Non-Serial Schedule**.

### **Example-02:**



In this schedule,

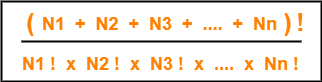
* There are two transactions T1 and T2 executing concurrently.
* The operations of T1 and T2 are interleaved.
* So, this schedule is an example of a **Non-Serial Schedule**.

## **Finding Number Of Schedules-**

Consider there are n number of transactions T1, T2, T3 …. , Tn with N1, N2, N3 …. , Nn number of operations respectively.

### **Total Number of Schedules-**

Total number of possible schedules (serial + non-serial) is given by-



### **Total Number of Serial Schedules-**

Total number of serial schedules

= Number of different ways of arranging n transactions

= n!

### **Total Number of Non-Serial Schedules-**

Total number of non-serial schedules

= Total number of schedules – Total number of serial schedules

## **PRACTICE PROBLEM BASED ON FINDING NUMBER OF SCHEDULES-**

## **Problem-**

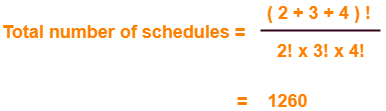
Consider there are three transactions with 2, 3, 4 operations respectively, find-

1. How many total number of schedules are possible?
2. How many total number of serial schedules are possible?
3. How many total number of non-serial schedules are possible?

## **Solution-**

### **Total Number of Schedules-**

Using the above formula, we have-



### **Total Number of Serial Schedules-**

Total number of serial schedules

= Number of different ways of arranging 3 transactions

= 3!

= 6

### **Total Number of Non-Serial Schedules-**

Total number of non-serial schedules

= Total number of schedules – Total number of serial schedules

= 1260 – 6

= 1254

## **Serializable in DBMS-**

* Some non-serial schedules may lead to inconsistency of the database.
* Serializability is a concept that helps to identify which non-serial schedules are correct and will maintain the consistency of the database.

## **Serializable Schedules-**

If a given non-serial schedule of ‘n’ transactions is equivalent to some serial schedule of ‘n’ transactions, then it is called as a **serializable schedule**.

## **Characteristics-**

Serializable schedules behave exactly same as serial schedules.

Thus, serializable schedules are always-

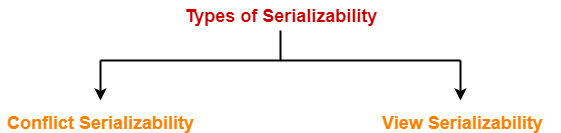
* Consistent
* [**Recoverable**](https://www.gatevidyalay.com/recoverable-schedules-irrecoverable-schedules-non-serializable-schedules/)
* [**Casacadeless**](https://www.gatevidyalay.com/cascading-schedule-cascading-rollback-cascadeless-schedule/)
* Strict

## **Serial Schedules Vs Serializable Schedules-**

|  |  |
| --- | --- |
| **Serial Schedules** | **Serializable Schedules** |
| No concurrency is allowed.  Thus, all the transactions necessarily execute serially one after the other. | Concurrency is allowed.  Thus, multiple transactions can execute concurrently. |
| Serial schedules lead to less resource utilization and CPU throughput. | Serializable schedules improve both resource utilization and CPU throughput. |
| Serial Schedules are less efficient as compared to serializable schedules.  (due to above reason) | Serializable Schedules are always better than serial schedules.  (due to above reason) |

**Types of Serializability-**

Serializability is mainly of two types-



1. Conflict Serializability
2. View Serializability

**Conflict Serializable:**

A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

**Conflicting operations:** Two operations are said to be conflicting if all conditions satisfy:

* They belong to different transactions
* They operate on the same data item
* At Least one of them is a write operation

Example: –

* **Conflicting** operations pair (R1(A), W2(A)) because they belong to two different transactions on same data item A and one of them is write operation.
* Similarly, (W1(A), W2(A)) and (W1(A), R2(A)) pairs are also **conflicting**.
* On the other hand, (R1(A), W2(B)) pair is **non-conflicting** because they operate on different data item.
* Similarly, ((W1(A), W2(B)) pair is **non-conflicting.**

### **Example of Conflict Serializability**

Lets consider this schedule:

T1 T2

----- ------

R(A)

R(B)

R(A)

R(B)

W(B)

W(A)

To convert this schedule into a serial schedule we must have to swap the R(A) operation of transaction T2 with the W(A) operation of transaction T1. However we cannot swap these two operations because they are conflicting operations, thus we can say that this given schedule is **not Conflict Serializable**.

Lets take another example:

T1 T2

----- ------

R(A)

R(A)

R(B)

W(B)

R(B)

W(A)

Lets **swap non-conflicting operations**:

After swapping R(A) of T1 and R(A) of T2 we get:

T1 T2

----- ------

R(A)

R(A)

R(B)

W(B)

R(B)

W(A)

After swapping R(A) of T1 and R(B) of T2 we get:

T1 T2

----- ------

R(A)

R(B)

R(A)

W(B)

R(B)

W(A)

After swapping R(A) of T1 and W(B) of T2 we get:

T1 T2

----- ------

R(A)

R(B)

W(B)

R(A)

R(B)

W(A)

We finally got a serial schedule after swapping all the non-conflicting operations so we can say that the given schedule is **Conflict Serializable**.

**Conflict Equivalent:**

Two schedules are said to be conflict equivalent when one can be transformed to another by swapping non-conflicting operations. In the example discussed above, S11 is conflict equivalent to S1 (S1 can be converted to S11 by swapping non-conflicting operations). Similarly, S11 is conflict equivalent to S12 and so on.

**Note 1:** Although S2 is not conflict serializable, but still it is conflict equivalent to S21 and S21 because S2 can be converted to S21 and S22 by swapping non-conflicting operations.

**Note 2:** The schedule which is conflict serializable is always conflict equivalent to one of the serial schedule. S1 schedule discussed above (which is conflict serializable) is equivalent to serial schedule (T1->T2).

**Checking Whether a Schedule is Conflict Serializable Or Not-**

### **Step-01:**Find and list all the conflicting operations.

### **Step-02:**Start creating a precedence graph by drawing one node for each transaction.

### **Step-03:**

* Draw an edge for each conflict pair such that if Xi (V) and Yj (V) forms a conflict pair then draw an edge from Ti to Tj.
* This ensures that Ti gets executed before Tj.

### **Step-04:**

* Check if there is any cycle formed in the graph.
* If there is no cycle found, then the schedule is conflict serializable otherwise not.

## **PRACTICE PROBLEMS BASED ON CONFLICT SERIALIZABILITY-**

## **Problem-01:**

Check whether the given schedule S is conflict serializable or not-

**S : R1(A) , R2(A) , R1(B) , R2(B) , R3(B) , W1(A) , W2(B)**

## **Solution-**

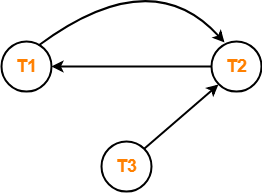
### **Step-01:**

List all the conflicting operations and determine the dependency between the transactions-

* R2(A) , W1(A) (T2 → T1)
* R1(B) , W2(B) (T1 → T2)
* R3(B) , W2(B) (T3 → T2)

### **Step-02:**

Draw the precedence graph-

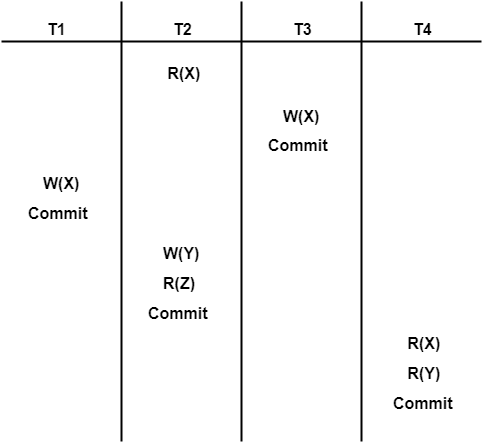


Clearly, there exists a cycle in the precedence graph.

Therefore, the given schedule S is not conflict serializable.

**Problem-02:**

Check whether the given schedule S is conflict serializable and recoverable or not-



### **Checking Whether S is Conflict Serializable Or Not-**

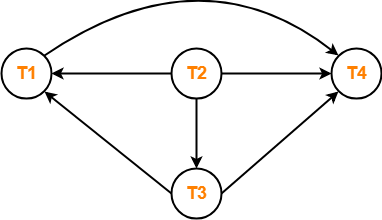
### **Step-01:**

List all the conflicting operations and determine the dependency between the transactions-

* R2(X) , W3(X) (T2 → T3)
* R2(X) , W1(X) (T2 → T1)
* W3(X) , W1(X) (T3 → T1)
* W3(X) , R4(X) (T3 → T4)
* W1(X) , R4(X) (T1 → T4)
* W2(Y) , R4(Y) (T2 → T4)

### **Step-02:**

 Draw the precedence graph-



* Clearly, there exists no cycle in the precedence graph.
* Therefore, the given schedule S is conflict serializable.

## **View Serializability-**

 If a given schedule is found to be view equivalent to some serial schedule, then it is called as a view serializable schedule.

## **View Equivalent Schedules-**

Consider two schedules S1 and S2 each consisting of two transactions T1 and T2.

Schedules S1 and S2 are called view equivalent if the following three conditions hold true for them-

## **Condition-01:**

 For each data item X, if transaction Ti reads X from the database initially in schedule S1, then in schedule S2 also, Ti must perform the initial read of X from the database.

### **Thumb Rule :**“Initial readers must be same for all the data items”.

## **Condition-02:**

If transaction Ti reads a data item that has been updated by the transaction Tj in schedule S1, then in schedule S2 also, transaction Ti must read the same data item that has been updated by the transaction Tj.

**Thumb Rule** :“Write-read sequence must be same.”.

## **Condition-03:**

For each data item X, if X has been updated at last by transaction Ti in schedule S1, then in schedule S2 also, X must be updated at last by transaction Ti.

### **Thumb Rule:**“Final writers must be same for all the data items”.

## **Checking Whether a Schedule is View Serializable Or Not-**

## **Method-01:**

Check whether the given schedule is conflict serializable or not.

* If the given schedule is conflict serializable, then it is surely view serializable. Stop and report your answer.
* If the given schedule is not conflict serializable, then it may or may not be view serializable. Go and check using other methods.

**Method-02:**

Check if there exists any blind write operation.

(Writing without reading is called as a blind write).

* If there does not exist any blind write, then the schedule is surely not view serializable. Stop and report your answer.
* If there exists any blind write, then the schedule may or may not be view serializable. Go and check using other methods.

**Method-03:**

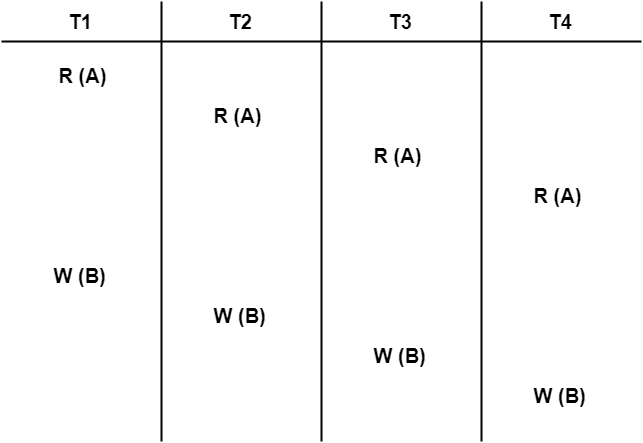
In this method, try finding a view equivalent serial schedule.

* By using the above three conditions, write all the dependencies.
* Then, draw a graph using those dependencies.
* If there exists no cycle in the graph, then the schedule is view serializable otherwise not.

## **PRACTICE PROBLEMS BASED ON VIEW SERIALIZABILITY-**

**Problem-01:**

Check whether the given schedule S is view serializable or not-



## **Solution-**

* We know, if a schedule is conflict serializable, then it is surely view serializable.
* So, let us check whether the given schedule is conflict serializable or not.

## **Checking Whether S is Conflict Serializable Or Not-**

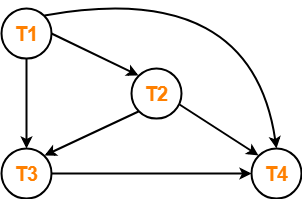
### **Step-01:**

List all the conflicting operations and determine the dependency between the transactions-

* W1(B) , W2(B) (T1 → T2)
* W1(B) , W3(B) (T1 → T3)
* W1(B) , W4(B) (T1 → T4)
* W2(B) , W3(B) (T2 → T3)
* W2(B) , W4(B) (T2 → T4)
* W3(B) , W4(B) (T3 → T4)

### **Step-02:**

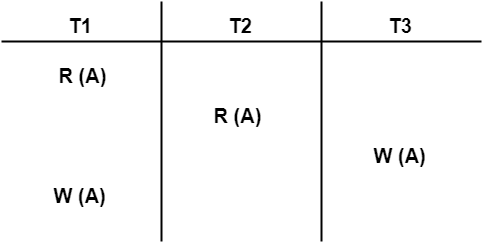
Draw the precedence graph-



* Clearly, there exists no cycle in the precedence graph.
* Therefore, the given schedule S is conflict serializable.
* Thus, we conclude that the given schedule is also view serializable.

## **Problem-02:**

Check whether the given schedule S is view serializable or not-



## **Solution-**

* We know, if a schedule is conflict serializable, then it is surely view serializable.
* So, let us check whether the given schedule is conflict serializable or not.

## **Checking Whether S is Conflict Serializable Or Not-**

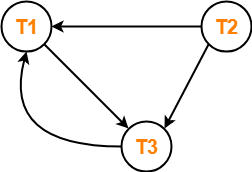
### **Step-01:**

List all the conflicting operations and determine the dependency between the transactions-

* R1(A) , W3(A) (T1 → T3)
* R2(A) , W3(A) (T2 → T3)
* R2(A) , W1(A) (T2 → T1)
* W3(A) , W1(A) (T3 → T1)

### **Step-02:**

Draw the precedence graph-



* Clearly, there exists a cycle in the precedence graph.
* Therefore, the given schedule S is not conflict serializable.

Now,

* Since, the given schedule S is not conflict serializable, so, it may or may not be view serializable.
* To check whether S is view serializable or not, let us use another method.
* Let us check for blind writes.

## **Checking for Blind Writes-**

* There exists a blind write W3 (A) in the given schedule S.
* Therefore, the given schedule S may or may not be view serializable.

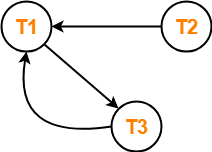
Now,

* To check whether S is view serializable or not, let us use another method.
* Let us derive the dependencies and then draw a dependency graph.

## **Drawing a Dependency Graph-**

* T1 firstly reads A and T3 firstly updates A.
* So, T1 must execute before T3.
* Thus, we get the dependency **T1 → T3**.
* Final updation on A is made by the transaction T1.
* So, T1 must execute after all other transactions.
* Thus, we get the dependency **(T2, T3) → T1**.
* There exists no write-read sequence.

Now, let us draw a dependency graph using these dependencies-



* Clearly, there exists a cycle in the dependency graph.
* Thus, we conclude that the given schedule S is not view serializable.

**Difference between Conflict and View Serializability :**

| S.No. | Conflict Serializability | View serializabilityviewView Serializability |
| --- | --- | --- |
| 1. | Two schedules are said to be conflict equivalent if all the conflicting operations in both the schedule get executed in the same order. If a schedule is a conflict equivalent to its serial schedule then it is called Conflict Serializable Schedule. | Two schedules are said to be view equivalent if the order of initial read, final write and update operations is the same in both the schedules. If a schedule is view equivalent to its serial schedule then it is called View Serializable Schedule. |
| 2. | If a schedule is view serializable then it may or may not be conflict serializable. | If a schedule is conflict serializable then it is also view serializable schedule. |
| 3. | Conflict equivalence can be easily achieved by reordering the operations of two transactions therefore, Conflict Serializability is easy to achieve. | Viewequivalence is rather difficult to achieve as both transactions should perform similar actions in a similar manner. Thus, View Serializability is difficult to achieve. |
| 4. | For a transaction T1 writing a value A that no one else reads but later some other transactions say T2 write its own value of A, W(A) cannot be placed under positions where it is never read. | If a transaction T1 writes a value A that no other transaction reads (because later some other transactions say T2 writes its own value of A) W(A) can be placed in positions of the schedule where it is never read |

## **Non-Serializable Schedules-**

* A non-serial schedule which is not serializable is called as a non-serializable schedule.
* A non-serializable schedule is not guaranteed to produce the the same effect as produced by some serial schedule on any consistent database.

## **Characteristics-**

Non-serializable schedules-

* may or may not be consistent
* may or may not be recoverable

The non-serializable schedule is divided into two types, Recoverable and Non-recoverable Schedule.

1. [Recoverable Schedule:](https://www.geeksforgeeks.org/recoverability-in-dbms/)Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction Tj is reading value updated or written by some other transaction Ti, then the commit of Tj must occur after the commit of Ti.

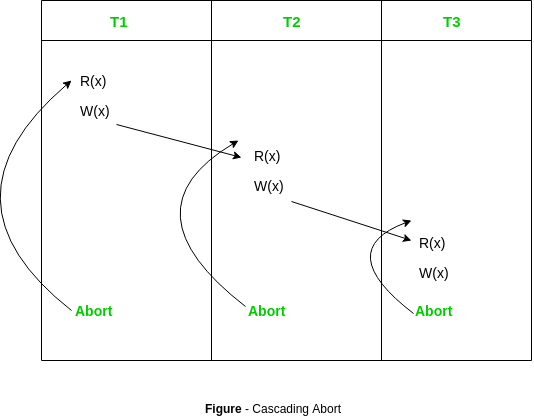
**Example –** Consider the following schedule involving two transactions T1 and T2.

| T1 | T2 |
| --- | --- |
| R(A) |  |
| W(A) |  |
|  | W(A) |
|  | R(A) |
| commit |  |
|  | commit |

This is a recoverable schedule since T1 commits before T2, that makes the value read by T2 correct.

There can be three types of recoverable schedule:

1. **Cascading Schedule:** Also called Avoids cascading aborts/rollbacks (ACA). When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or cascading abort. Example:



1. [Cascadeless Schedule:](https://www.geeksforgeeks.org/cascadeless-in-dbms/)Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

In other words, if some transaction Tj wants to read value updated or written by some other transaction Ti, then the commit of Tj must read it after the commit of Ti.

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

| T1 | T2 |
| --- | --- |
| R(A) |  |
| W(A) |  |
|  | W(A) |
| commit |  |
|  | R(A) |
|  | commit |

This schedule is cascadeless. Since the updated value of A is read by T2 only after the updating transaction i.e. T1 commits.

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

| T1 | T2 |
| --- | --- |
| R(A) |  |
| W(A) |  |
|  | R(A) |
|  | W(A) |
| abort |  |

It is a recoverable schedule but it does not avoid cascading aborts. It can be seen that if T1 aborts, T2 will have to be aborted too in order to maintain the correctness of the schedule as T2 has already read the uncommitted value written by T1.

1. **Strict Schedule:**A schedule is strict if for any two transactions Ti, Tj, if a write operation of Ti precedes a conflicting operation of Tj (either read or write), then the commit or abort event of Ti also precedes that conflicting operation of Tj.In other words, Tj can read or write updated or written value of Ti only after Ti commits/aborts.

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

| T1 | T2T2 |
| --- | --- |
| R(A) |  |
|  | R(A) |
| W(A) |  |
| commit |  |
|  | W(A) |
|  | R(A) |
|  | commit |
|  |  |
|  |  |

This is a strict schedule since T2 reads and writes A which is written by T1 only after the commit of T1.

2) Non-Recoverable Schedule:Example:

Consider the following schedule involving two transactions T1 and T2.

| T1 | T2 |
| --- | --- |
| R(A) |  |
| W(A) |  |
|  | W(A) |
|  | R(A) |
|  | commit |
| abort |  |

T2 read the value of A written by T1, and committed. T1 later aborted, therefore the value read by T2 is wrong, but since T2 committed, this schedule is non-recoverable.

Note – It can be seen that:

* Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
* Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.
* Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.

**Checking Whether a Schedule is Recoverable or Irrecoverable-**

## **Method-01:**

Check whether the given schedule is conflict serializable or not.

* If the given schedule is conflict serializable, then it is surely recoverable. Stop and report your answer.
* If the given schedule is not conflict serializable, then it may or may not be recoverable. Go and check using other methods.

## **Method-02:**

Check if there exists any dirty read operation.

(Reading from an uncommitted transaction is called as a dirty read)

* If there does not exist any dirty read operation, then the schedule is surely recoverable. Stop and report your answer.
* If there exists any dirty read operation, then the schedule may or may not be recoverable.

If there exists a dirty read operation, then follow the following cases-

### **Case-01:**

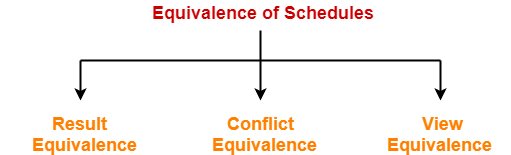
If the commit operation of the transaction performing the dirty read occurs before the commit or abort operation of the transaction which updated the value, then the schedule is irrecoverable.

### **Case-02:**

If the commit operation of the transaction performing the dirty read is delayed till the commit or abort operation of the transaction which updated the value, then the schedule is recoverable.

**Equivalence of Schedules-**

In DBMS, schedules may have the following three different kinds of equivalence relations among them-



1. Result Equivalence
2. Conflict Equivalence
3. View Equivalence

## **1. Result Equivalent Schedules-**

* If any two schedules generate the same result after their execution, then they are called as result equivalent schedules.
* This equivalence relation is considered of least significance.
* This is because some schedules might produce same results for some set of values and different results for some other set of values.

## **2. Conflict Equivalent Schedules-**

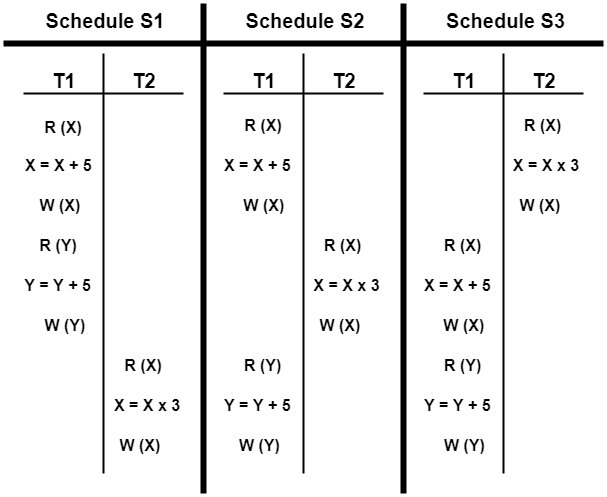
If any two schedules satisfy the following two conditions, then they are called as conflict equivalent schedules-

* The set of transactions present in both the schedules is same.
* The order of pairs of conflicting operations of both the schedules is same.

## **PRACTICE PROBLEMS BASED ON EQUIVALENCE OF SCHEDULES-**

## **Problem-01:**

Are the following three schedules result equivalent?



## **Solution-**

To check whether the given schedules are result equivalent or not,

* We will consider some arbitrary values of X and Y.
* Then, we will compare the results produced by each schedule.
* Those schedules which produce the same results will be result equivalent.

Let X = 2 and Y = 5.

On substituting these values, the results produced by each schedule are-

**Results by Schedule S1-** X = 21 and Y = 10

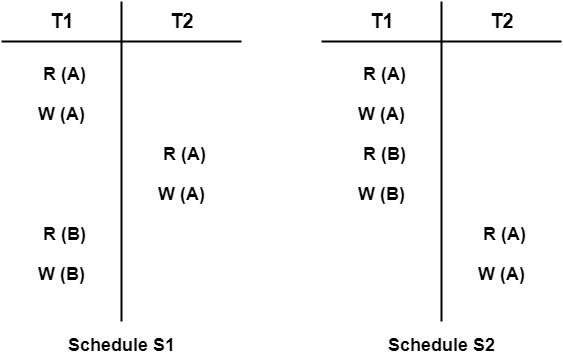
**Results by Schedule S2-** X = 21 and Y = 10

**Results by Schedule S3-** X = 11 and Y = 10

* Clearly, the results produced by schedules S1 and S2 are same.
* Thus, we conclude that S1 and S2 are result equivalent schedules.

## **Problem-02:**

Are the following two schedules conflict equivalent?



## **Solution-**

To check whether the given schedules are conflict equivalent or not,

* We will write their order of pairs of conflicting operations.
* Then, we will compare the order of both the schedules.
* If both the schedules are found to have the same order, then they will be conflict equivalent.

### **For schedule S1-**

The required order is-

* R1(A) , W2(A)
* W1(A) , R2(A)
* W1(A) , W2(A)

### **For schedule S2-**

The required order is-

* R1(A) , W2(A)
* W1(A) , R2(A)
* W1(A) , W2(A)
* Clearly, both the given schedules have the same order.
* Thus, we conclude that S1 and S2 are conflict equivalent schedules.

**Why recovery is needed in DBMS:**

Basically, whenever a transaction is submitted to a DBMS for execution, the operating system is responsible for making sure or to be confirmed that all the operation which need to be in performed in the transaction have completed successfully and their effect is either recorded in the database or the transaction doesn’t affect the database or any other transactions.

The DBMS must not permit some operation of the transaction T to be applied to the database while other operations of T is not. This basically may happen if a transaction fails after executing some of its operations but before executing all of them.

**Types of failures –**There are basically following types of failures that may occur and leads to failure of the transaction such as:

1. Transaction failure
2. System failure
3. Media failure

The different types of failures that may occur during the transaction.

1. **System crash –**A hardware, software or network error occurs comes under this category this types of failures basically occurs during the execution of the transaction. Hardware failures are basically considered as Hardware failure.
2. **System error –**Some operation that is performed during the transaction is the reason for this type of error to occur, such as integer or divide by zero. This type of failures is also known as the transaction which may also occur because of erroneous parameter values or because of a logical programming error. In addition to this user may also interrupt the execution during execution which may lead to failure in the transaction.
3. **Local error –**This basically happens when we are doing the transaction but certain conditions may occur that may lead to cancellation of the transaction. This type of error is basically coming under Local error. The simple example of this is that data for the transaction may not found. When we want to debit money from an insufficient balance account which leads to the cancellation of our request or transaction. And this exception should be programmed in the transaction itself so that it wouldn’t be considered as a failure.
4. **Concurrency control enforcement –**The concurrency control method may decide to abort the transaction, to start again because it basically violates serializability or we can say that several processes are in a deadlock.
5. **Disk failure –**This type of failure basically occur when some disk loses their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read /write operation of the transaction.
6. **Castropher –**These are also known as physical problems it basically refers to the endless list of problems that include power failure or air-conditioning failure, fire, theft sabotage overwriting disk or tapes by mistake and mounting of the wrong tape by the operator.

**Transaction Isolation Levels in DBMS:**

As we know that, in order to maintain consistency in a database, it follows ACID properties. Among these four properties (Atomicity, Consistency, Isolation and Durability) Isolation determines how transaction integrity is visible to other users and systems. It means that a transaction should take place in a system in such a way that it is the only transaction that is accessing the resources in a database system.

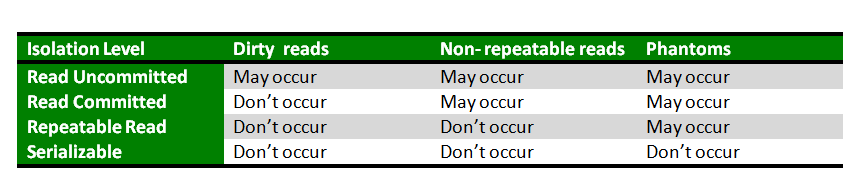
Isolation levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system. A transaction isolation level is defined by the following phenomena –

* **Dirty Read –** A Dirty read is the situation when a transaction reads a 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 same row twice, and get 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 transaction, thereby allowing dirty reads. In 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 allows dirty read. The transaction holds a read or write lock on the current row, and thus prevent 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 all rows it inserts, updates, or deletes. Since other transaction 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 is given below clearly depicts the relationship between isolation levels, read phenomena and locks :



**Database Recovery Techniques in DBMS:**

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

There are both automatic and non-automatic ways for both, backing up of data and recovery from any failure situations. The techniques used to recover the lost data due to system crash, transaction errors, viruses, catastrophic failure, incorrect commands execution etc. are database recovery techniques. So to prevent data loss recovery techniques based on deferred update and immediate update or backing up data can be used.

* 1. Log based recovery
  2. Checkpoints
  3. Immediate database modification
  4. Delayed database modification

**Log based Recovery in DBMS:**

Atomicity property of DBMS states that either all the operations of transactions must be performed or none. The modifications done by an aborted transaction should not be visible to database and the modifications done by committed transaction should be visible.

To achieve our goal of atomicity, user must first output to stable storage information describing the modifications, without modifying the database itself. This information can help us ensure that all modifications performed by committed transactions are reflected in the database. This information can also help us ensure that no modifications made by an aborted transaction persist in the database.

**Log and log records –**The log is a sequence of log records, recording all the update activities in the database. In a stable storage, logs for each transaction are maintained. Any operation which is performed on the database is recorded is on the log. Prior to performing any modification to database, an update log record is created to reflect that modification.

An update log record represented as: <Ti, Xj, V1, V2> has these fields:

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

Other type of log records are:

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

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

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

**Recovery using Log records –**

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

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

**Deferred-modification technique:**

The **deferred-modification technique** ensures transaction atomicity by recording all database modifications in the log, but deferring the execution of all write operations of a transaction until the transaction partially commits. A transaction is said to be partially committed once the final action of the transaction has been executed.

When a transaction partially commits, the information on the log associated with the transaction is used in executing the deferred writes. If the system crashes before the transaction completes its execution, or if the transaction aborts, then the information on the log is simply ignored.

The execution of transaction Ti proceeds as follows. Before Ti starts its execution, a record i start> is written to the log. A write(X) operation by Ti results in the writing of a new record to the log. Finally, when Ti partially commits, a record i commit> is written to the log.

When transaction Ti partially commits, the records associated with it in the log are used in executing the deferred writes. Since a failure may occur while this updating is taking place, we must ensure that, before the start of these updates, all the log records are written out to stable storage. Once they have been written, the actual updating takes place, and the transaction enters the committed state.

Observe that only the new value of the data item is required by the deferred modification technique.

Let T0 be a transaction that transfers $50 from account A to account B:

T0: read(A);

A := A − 50;

write(A);

read(B);

B := B 50;

write(B).

Let T1 be a transaction that withdraws $100 from account C:

T1: read(C);

C := C − 100;

write(C).

Suppose that these transactions are executed serially, in the order T0 followed by T1, and that the values of accounts A, B, and C before the execution took place were $1000, $2000, and $700, respectively.

<T0 START>

<T0,A,950>

<T0,B,1050>

<T0 COMMIT>

<T0,C,600>

<T0 COMMIT>

There are various orders in which the actual outputs can take place to both the database system and the log as a result of the execution of T0 and T1. One such order appears in the above log. Note that the value of A is changed in the database only after the record 0, A, 950> has been placed in the log.

**Immediate Database Modification:**

The immediate-modification technique allows database modifications to be output to the database while the transaction is still in the active state. Data modifications written by active transactions are called **uncommitted modifications**. In the event of a crash or a transaction failure, the system must use the old-value field of the log records to restore the modified data items to the value they had prior to the start of the transaction. The undo operation accomplishes this restoration.

<T0 START>

<T0,A,1000,950>

<T0,B,2000,2050>

<T0 COMMIT>

<T1 START>

<T1,C,700,600>

<T1 COMMIT>

Before a transaction Ti starts its execution, the system writes the record <Ti start> to the log. During its execution, any write(X) operation by Ti is preceded by the writing of the appropriate new update record to the log. When Ti partially commits, the system writes the record <Ti commit> to the log.Since the information in the log is used in reconstructing the state of the database, we cannot allow the actual update to the database to take place before the corresponding log record is written out to stable storage.We therefore require that, before execution of an output(B) operation, the log records corresponding to B be written onto stable storage.

Let us reconsider our simplified banking system, with transactions T0 and T1 executed one after the other in the order T0 followed by T1. The portion of the log containing the relevant information concerning these two transactions appears in above log which shows one possible order in which the actual outputs took place in both the database system and the log as a result of the execution of T0 and T1.

Using the log, the system can handle any failure that does not result in the loss of information in non-volatile storage. The recovery scheme uses two recovery procedures:  
• undo(Ti) restores the value of all data items updated by transaction Ti to the old values.  
• redo(Ti) sets the value of all data items updated by transaction Ti to the new values.  
  
The set of data items updated by Ti and their respective old and new values can be found in the log. The undo and redo operations must be idempotent to guarantee correct behavior even if a failure occurs during the recovery process. After a failure has occurred, the recovery scheme consults the log to determine which transactions need to be redone, and which need to be undone:

• Transaction Ti needs to be undone if the log contains the record <Ti start>, but does not contain the record <Ti commit>.  
• Transaction Ti needs to be redone if the log contains both the record <Ti start> and the record <Ti commit>.

Suppose that the system crashes before the completion of the transactions. We shall consider three cases. The state of the logs for each of these cases appears in log.

First, let us assume that the crash occurs just after the log record for the step

**write(B)**

of transaction T0 has been written to stable storage (log)When the system comes back up, it finds the record <T0 start> in the log, but no corresponding <T0 commit> record. Thus, transaction T0 must be undone, so an undo(T0) is performed. As a result, the values in accounts A and B (on the disk) are restored to $1000 and $2000, respectively.

Next, let us assume that the crash comes just after the log record for the step

write(C)

of transaction T1 has been written to stable storage (log). When the system comes back up, two recovery actions need to be taken. The operation undo(T1) must be performed, since the record <T1 start> appears in the log, but there is no record <T1 commit>. The operation redo(T0)must be performed, since the log contains both the record <T0 start> and the record <T0 commit>.

Finally, let us assume that the crash occurs just after the log record

<T1 commit>has been written to stable storage (log). When the system comes back up, both T0 and T1 need to be redone, since the records <T0 start> and <T0 commit> appear in the log, as do the records <T1 start> and <T1 commit>

**Checkpoints in DBMS:**

Why do we need Checkpoints ?

Whenever transaction logs are created in a real-time environment, it eats up lots of storage space. Also keeping track of every update and its maintenance may increase the physical space of the system. Eventually, the transaction log file may not be handled as the size keeps growing. This can be addressed with checkpoints. The methodology utilized for removing all previous transaction logs and storing them in permanent storage is called a Checkpoint.

What is a Checkpoint ?

The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed. During transaction execution, such checkpoints are traced. After execution, transaction log files will be created.

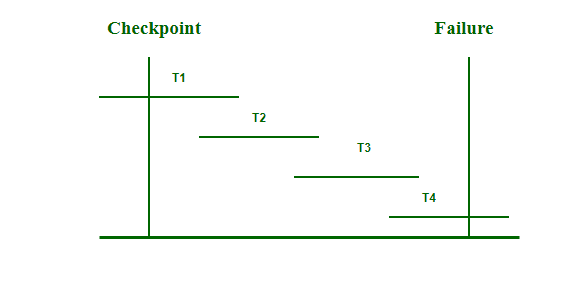
Upon reaching the savepoint/checkpoint, the log file is destroyed by saving its update to the database. Then a new log is created with upcoming execution operations of the transaction and it will be updated until the next checkpoint and the process continues.

How to use Checkpoints in database ?

Steps :

* Write begin\_checkpoint record into log.
* Collect checkpoint data in the stable storage.
* Write end\_checkpoint record into log.

The behavior when the system crashes and recovers when concurrent transactions are executed is shown below –



* The recovery system reads the logs backward from the end to the last checkpoint i.e. from T4 to T1.
* It will keep track of two lists – Undo and Redo.
* Whenever there is a log with instruction <Tn, start>and <Tn, commit> or only <Tn, commit> then it will put that transaction in Redo List. T2 and T3 contain <Tn, Start> and <Tn, Commit> whereas T1 will have only <Tn, Commit>. Here, T1, T2, and T3 are in the redo list.
* Whenever a log record with no instruction of commit or abort is found, that transaction is put to Undo List <Here, T4 has <Tn, Start> but no <Tn, commit> as it is an ongoing transaction. T4 will be put in the undo list.

All the transactions in the redo-list are deleted with their previous logs and then redone before saving their logs. All the transactions in the undo-list are undone and their logs are deleted.

Relevance of Checkpoints :

A checkpoint is a feature that adds a value of C in ACID-compliant to RDBMS. A checkpoint is used for recovery if there is an unexpected shutdown in the database. Checkpoints work on some intervals and write all dirty pages (modified pages) from logs relay to data file from i.e from a buffer to physical disk. It is also known as the hardening of dirty pages. It is a dedicated process and runs automatically by SQL Server at specific intervals. The synchronization point between the database and transaction log is served with a checkpoint.

Advantages of using Checkpoints :

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

Real-Time Applications of Checkpoints :

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

**Concurrency control:**

Concurrency Control in Database Management System is a procedure of managing simultaneous operations without conflicting with each other. It ensures that Database transactions are performed concurrently and accurately to produce correct results without violating data integrity of the respective Database.

Concurrent access is quite easy if all users are just reading data. There is no way they can interfere with one another. Though for any practical Database, it would have a mix of READ and WRITE operations and hence the concurrency is a challenge.

DBMS Concurrency Control is used to address such conflicts, which mostly occur with a multi-user system. Therefore, Concurrency Control is the most important element for proper functioning of a Database Management System where two or more database transactions are executed simultaneously, which require access to the same data.

## **Potential problems of Concurrency**

Here, are some issues which you will likely to face while using the DBMS Concurrency Control method:

* **Lost Updates** occur when multiple transactions select the same row and update the row based on the value selected
* Uncommitted dependency issues occur when the second transaction selects a row which is updated by another transaction (**dirty read**)
* **Non-Repeatable Read** occurs when a second transaction is trying to access the same row several times and reads different data each time.
* **Incorrect Summary issue** occurs when one transaction takes summary over the value of all the instances of a repeated data-item, and second transaction update few instances of that specific data-item. In that situation, the resulting summary does not reflect a correct result.

## **Why use Concurrency method?**

Reasons for using Concurrency control method is DBMS:

* To apply Isolation through mutual exclusion between conflicting transactions
* To resolve read-write and write-write conflict issues
* To preserve database consistency through constantly preserving execution obstructions
* The system needs to control the interaction among the concurrent transactions. This control is achieved using concurrent-control schemes.
* Concurrency control helps to ensure serializability

### **Example**

Assume that two people who go to electronic kiosks at the same time to buy a movie ticket for the same movie and the same show time.

However, there is only one seat left in for the movie show in that particular theatre. Without concurrency control in DBMS, it is possible that both moviegoers will end up purchasing a ticket. However, concurrency control method does not allow this to happen. Both moviegoers can still access information written in the movie seating database. But concurrency control only provides a ticket to the buyer who has completed the transaction process first.

**Concurrency problems in DBMS Transactions:**

When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems. These problems are commonly referred to as concurrency problems in database environment. The five concurrency problems that can occur in database are:

(i). Temporary Update Problem

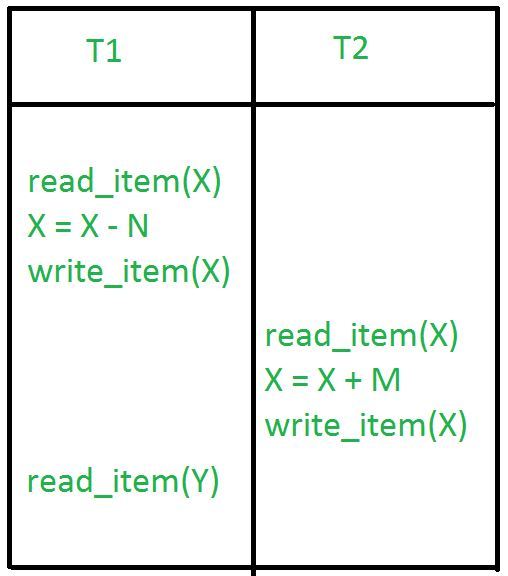
(ii). Incorrect Summary Problem

(iii). Lost Update Problem

(iv). Unrepeatable Read Problem

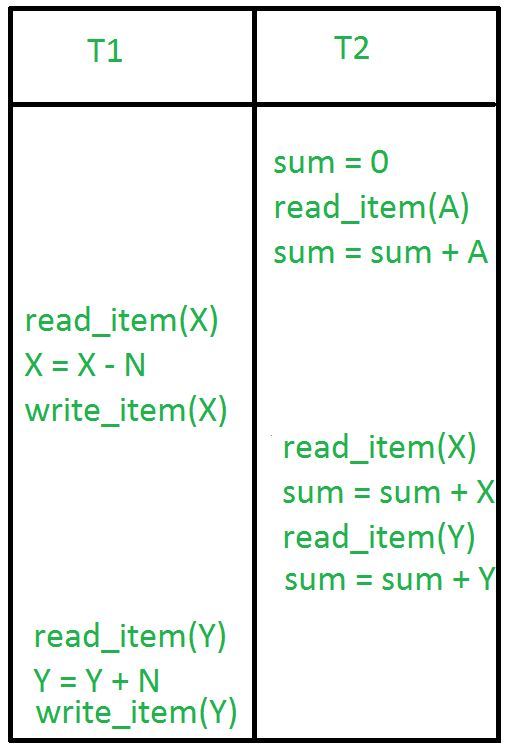
(v). Phantom Read Problem

**1)Temporary Update Problem:**Temporary update or dirty read problem occurs when one transaction updates an item and fails. But the updated item is used by another transaction before the item is changed or reverted back to its last value.

Example:  


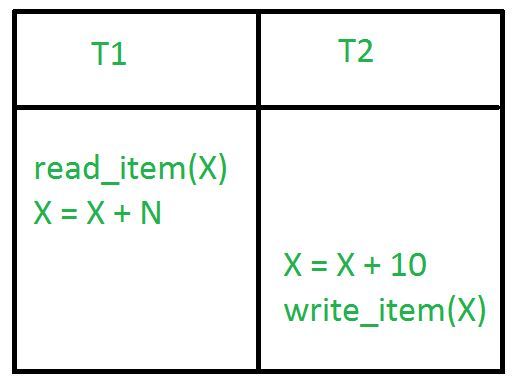
In the above example, if transaction 1 fails for some reason then X will revert back to its previous value. But transaction 2 has already read the incorrect value of X.

**2)Incorrect Summary Problem:** Consider a situation, where one transaction is applying the aggregate function on some records while another transaction is updating these records. The aggregate function may calculate some values before the values have been updated and others after they are updated.

Example:  


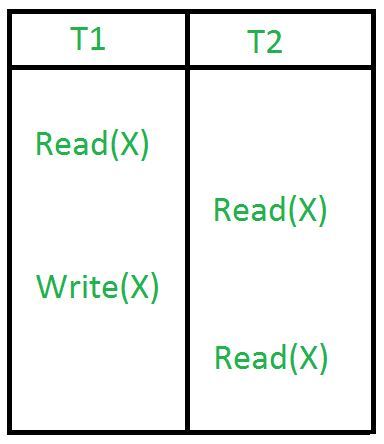
In the above example, transaction 2 is calculating the sum of some records while transaction 1 is updating them. Therefore the aggregate function may calculate some values before they have been updated and others after they have been updated.

**3)Lost Update Problem:**In the lost update problem, update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction.

**Example:**  


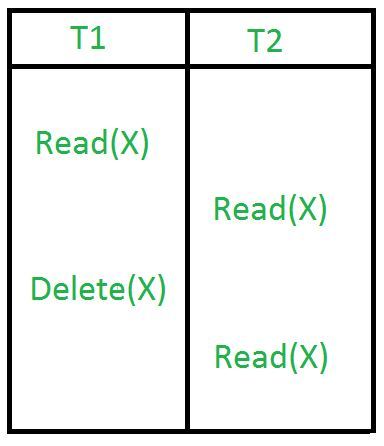
In the above example, transaction 1 changes the value of X but it gets overwritten by the update done by transaction 2 on X. Therefore, the update done by transaction 1 is lost.

**4)Unrepeatable Read Problem:**The unrepeatable problem occurs when two or more read operations of the same transaction read different values of the same variable.

**Example:**  


In the above example, once transaction 2 reads the variable X, a write operation in transaction 1 changes the value of the variable X. Thus, when another read operation is performed by transaction 2, it reads the new value of X which was updated by transaction 1.

**5)Phantom Read Problem:**The phantom read problem occurs when a transaction reads a variable once but when it tries to read that same variable again, an error occurs saying that the variable does not exist.

**Example:**  


In the above example, once transaction 2 reads the variable X, transaction 1 deletes the variable X without transaction 1’s knowledge. Thus, when transaction 2 tries to read X, it is not able to it.

**Concurrency Control Techniques:**

Concurrency control is provided in a database to:

(i) enforce isolation among transactions.

(ii) preserve database consistency through consistency preserving execution of transactions.

(iii) resolve read-write and write-read conflicts.

Various concurrency control techniques are:

**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.

* Interaction with concurrency control
* Transaction rollback
* Checkpoints
* 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 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 the 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 transaction 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.

**Concurrency Control Protocols:**

Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose. Following are the Concurrency Control techniques in DBMS:

* Lock-Based Protocols
* Two Phase Locking Protocol
* Timestamp-Based Protocols
* Validation-Based Protocols
* Multiple Granularity Locking protocol

**Lock-based Protocols:**

Lock Based Protocols in DBMS is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock. Lock based protocols help to eliminate the concurrency problem in DBMS for simultaneous transactions by locking or isolating a particular transaction to a single user.

A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks in DBMS help synchronize access to the database items by concurrent transactions.

All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

**Binary Locks:** A Binary lock on a data item can either locked or unlocked states.

Shared/exclusive: This type of locking mechanism separates the locks in DBMS based on their uses. If a lock is acquired on a data item to perform a write operation, it is called an exclusive lock.

1. Shared Lock (S): A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions. This is because you will never have permission to update data on the data item.

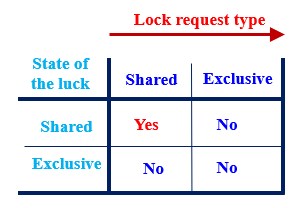
For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account's balance, shared lock prevent it until the reading process is over.

2. Exclusive Lock (X): With the Exclusive Lock, a data item can be read as well as written. This is exclusive and can't be held concurrently on the same data item. X-lock is requested using lock-x instruction. Transactions may unlock the data item after finishing the 'write' operation.

For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

**Lock Compatibility Matrix**

A keynote while applying the Lock based protocols in DBMS is that: any number of tractions can hold a Shared Lock while only one traction has a claim over Exclusive Lock since because shared lock is only reading the data and not performing any other actions whereas the exclusive lock is performing both read and write operations. This can be illustrated using a compatibility matrix like the one shown below:



The figure illustrates that when two transactions are involved, and both attempt to only read a given data item, it is permitted and no conflict arises, but when one transaction attempts to write the data item and another one tries to read or write at the same time, conflict occurs resulting in a denied interaction.

Conversion between the locks is possible by the two methods listed below:

Upgrading: conversion from a read lock to write lock

Downgrading: conversion from a write lock to read lock

**Problems associated with Simple locking:**

* Data inconsistency between multiple transactions
* Deadlock, a situation where the transactions try to access lock on already locked data items
* No guarantee of serializability (i.e. execution of a concurrent transaction equivalent to that of a transaction executed serially)

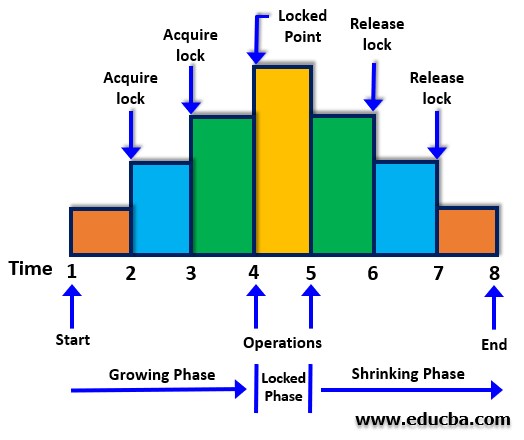
**Two Phase Locking (2PL):**

The trivial requirement for a 2PL to exist is that the locking and unlocking of a transaction take place in either of the 2 phases i.e. Growing or Shrinking phase

Growing Phase: It is the phase where new locks can be acquired on the data items.

Shrinking phase: It is the phase where the existing locks on the data items are released.

The above phases in a DBMS are determined by something called a ‘Lock Point’. Lock point is the point where a transaction has achieved its final lock. It is also the point where the growing phase ends and the shrinking phase begins.



Note:

* Two-Phase Locking (2PL) is a concurrency control method which divides the execution phase of a transaction into three parts.
* It ensures conflict serializable schedules.
* If read and write operations introduce the first unlock operation in the transaction, then it is said to be Two-Phase Locking Protocol.

**Types of 2 Phase Locking (PL)**

Here are the types of 2 phase locking mention below

1. Conservative 2PL:Conservative or Static 2PL when implied acquires all the locks before a transaction begins and releases the locks once the transaction is committed. This kind of 2 PL helps in overcoming problems related to cascading rollback and deadlocks.

Note:

* Conservative Two – Phase Locking Protocol is also called as Static Two – Phase Locking Protocol.
* This protocol is almost free from deadlocks as all required items are listed in advanced.
* It requires locking of all data items to access before the transaction starts.

2. Strict 2PL:In this type of 2PL, the exclusive (write) lock is released only after a transaction is committed, whereas a shared (read) lock can be released at regular intervals. Though Strict 2PL helps overcome the cascading rollback issues, it may also cause a bottleneck in some cases.

Note:

* Strict Two-Phase Locking Protocol avoids cascaded rollbacks.
* This protocol not only requires two-phase locking but also all exclusive-locks should be held until the transaction commits or aborts.
* It is not deadlock free.
* It ensures that if data is being modified by one transaction, then other transaction cannot read it until first transaction commits.
* Most of the database systems implement rigorous two – phase locking protocol.

3. Strong strict or Rigorous 2PL:In this type of 2PL, both shared and exclusive locks are released only when the transaction is ended, i.e. when the transaction is committed or aborted. This kind of 2PL is used in practice today, promotes serializability and is simpler to implement due to the strictness involved w.r.t the implementation of the phase endings.

Note:

* Rigorous Two – Phase Locking Protocol avoids cascading rollbacks.
* This protocol requires that all the share and exclusive locks to be held until the transaction commits.

**Timestamp Ordering Protocol**

* The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
* The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.
* The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.
* Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.
* The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

The main idea for this protocol is to order the transactions based on their Timestamps. A schedule in which the transactions participate is then serializable and the only equivalent serial schedule permitted has the transactions in the order of their Timestamp Values. Stating simply, the schedule is equivalent to the particular Serial Order corresponding to the order of the Transaction timestamps. Algorithm must ensure that, for each items accessed by Conflicting Operations in the schedule, the order in which the item is accessed does not violate the ordering. To ensure this, use two Timestamp Values relating to each database item X.

1. W­\_TS(X) is the largest timestamp of any transaction that executed write(X) successfully.
2. R\_TS(X) is the largest timestamp of any transaction that executed read(X) successfully.

Basic Timestamp ordering protocol works as follows:

Every transaction is issued a timestamp based on when it enters the system. Suppose, if an old transaction Ti has timestamp TS(Ti), a new transaction Tj is assigned timestamp TS(Tj) such that TS(Ti) < TS(Tj).The protocol manages concurrent execution such that the timestamps determine the serializability order. The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. Whenever some Transaction T tries to issue a R\_item(X) or a W\_item(X), the Basic TO algorithm compares the timestamp of T with R\_TS(X) & W\_TS(X) to ensure that the Timestamp order is not violated. This describe the Basic TO protocol in following two cases.

1. Whenever a Transaction T issues a W\_item(X) operation, check the following conditions:

* If R\_TS(X) > TS(T) or if W\_TS(X) > TS(T), then abort and rollback T and reject the operation. else,
* Execute W\_item(X) operation of T and set W\_TS(X) to TS(T).

1. Whenever a Transaction T issues a R\_item(X) operation, check the following conditions:

* If W\_TS(X) > TS(T), then abort and reject T and reject the operation, else
* If 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 the Basic TO algorithm detects twp conflicting operation that occur in incorrect order, it rejects the later of the two operation by aborting the Transaction that issued it. Schedules produced by Basic TO are guaranteed to be conflict serializable. Already discussed that using Timestamp, can ensure that our schedule will be deadlock free.

One drawback of Basic TO protocol is that it Cascading Rollback is still possible. Suppose we have a Transaction T1 and T2 has used a value written by T1. If T1 is aborted and resubmitted to the system then, T must also be aborted and rolled back. So the problem of Cascading aborts still prevails.

**Advantages and Disadvantages of TO protocol:**

TO protocol ensures serializability since the precedence graph is as follows:

* TS protocol ensures freedom from deadlock that means no transaction ever waits.
* But the schedule may not be recoverable and may not even be cascade- free.

**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 been 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 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:

**Read Phase:** Values of committed data items from the database can be read by a transaction. Updates are only applied to local data versions.

**Validation Phase:** Checking is performed to make sure that there is no violation of serializability when the transaction updates are applied to the database.

**Write Phase:** On the success of the validation phase, the transaction updates are applied to the database, otherwise, the updates are discarded and the transaction is slowed down.

The idea behind optimistic concurrency is to do all the checks at once; hence transaction execution proceeds with a minimum of overhead until the validation phase is reached. If there is not much interference among transactions most of them will have successful validation, otherwise, results will be discarded and restarted later. These circumstances are not much favourable for optimization techniques, since, the assumption of less interference is not satisfied.

Validation based protocol is useful for rare conflicts. Since only local copies of data are included in rollbacks, cascading rollbacks are avoided. This method is not favourable for longer transactions because they are more likely to have conflicts and might be repeatedly rolled back due to conflicts with short transactions.

In order to perform the Validation test, each transaction should go through the various phases as described above. Then, we must know about the following three time-stamps that we assigned to transaction Ti, to check its validity:

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

2. Validation(Ti): It is the time when Ti just ﬁnished its read phase and begin its validation phase.

3. Finish(Ti): the time when Ti end it’s all writing operations in the database under write-phase.

Two more terms that we need to know are:

1. Write\_set: of a transaction contains all the write operations that Ti performs.

2. Read\_set: of a transaction contains all the read operations that Ti performs.

In the Validation phase for transaction Ti the protocol inspect that Ti doesn’t overlap or intervene with any other transactions currently in their validation phase or in committed. The validation phase for Ti checks that for all transaction Tj one of the following below conditions must hold to being validated or pass validation phase:

1.Finish(Tj)<Starts(Ti), since Tj finishes its execution means completes its write-phase before Ti started its execution(read-phase). Then the serializability indeed maintained.

2. Ti begins its write phase after Tj completes its write phase, and the read\_set of Ti should be disjoint with write\_set of Tj.

3. Tj completes its read phase before Ti completes its read phase and both read\_set and write\_set of Ti are disjoint with the write\_set of Tj.

**Advantages:**

1. Avoid Cascading-rollbacks: This validation based scheme avoid cascading rollbacks since the final write operations to the database are performed only after the transaction passes the validation phase. If the transaction fails then no updation operation is performed in the database. So no dirty read will happen hence possibilities cascading-rollback would be null.

2. Avoid deadlock: Since a strict time-stamping based technique is used to maintain the specific order of transactions. Hence deadlock isn’t possible in this scheme.

**Disadvantages:**

1. Starvation: There might be a possibility of starvation for long-term transactions, due to a sequence of conﬂicting short-term transactions that cause the repeated sequence of restarts of the long-term transactions so on and so forth. To avoid starvation, conﬂicting transactions must be temporarily blocked for some time, to let the long-term transactions to ﬁnish.

**Multiple Granularity locking protocol:**

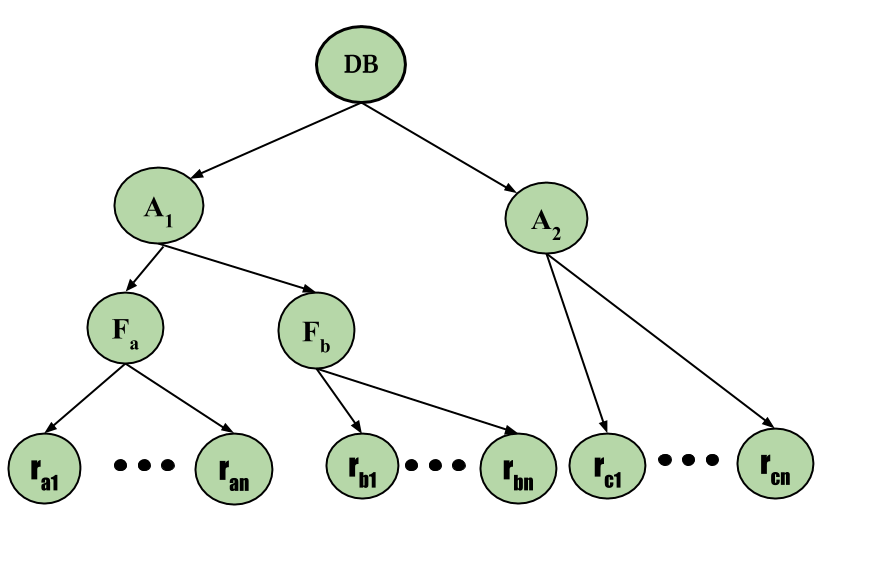
In the various Concurrency Control schemes have used different methods and every individual Data item as the unit on which synchronization is performed. A certain drawback of this technique is if a transaction Ti needs to access the entire database, and a locking protocol is used, then Ti must lock each item in the database. It is less efficient, it would be more simpler if Ti could use a single lock to lock the entire database. But, if it consider the second proposal, this should not in fact overlook the certain flaw in the proposed method. Suppose another transaction just needs to access a few data items from a database, so locking the entire database seems to be unnecessary moreover it may cost us loss of Concurrency, which was our primary goal in the first place. To bargain between Efficiency and Concurrency. Use Granularity.

Granularity – It is the size of data item allowed to lock. Now Multiple Granularity means hierarchically breaking up the database into blocks which can be locked and can be track what need to lock and in what fashion. Such a hierarchy can be represented graphically as a tree.

For example, consider the tree, which consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type area; the database consists of exactly these areas. Area has children nodes which are called files. Every area has those files that are its child nodes. No file can span more than one area.

Finally, each file has child nodes called records. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file. Hence, the levels starting from the top level are:

* database
* area
* file
* record



Consider the above diagram for the example given, each node in the tree can be locked individually. As in the 2-phase locking protocol, it shall use shared and exclusive lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also implicitly locks all the descendants of that node in the same lock mode. For example, if transaction Ti gets an explicit lock on file Fc in exclusive mode, then it has an implicit lock in exclusive mode on all the records belonging to that file. It does not need to lock the individual records of Fc explicitly. this is the main difference between Tree Based Locking and Hierarchical locking for multiple granularity.

Now, with locks on files and records made simple, how does the system determine if the root node can be locked? One possibility is for it to search the entire tree but the solution nullifies the whole purpose of the multiple-granularity locking scheme. A more efficient way to gain this knowledge is to introduce a new lock mode, called Intention lock mode.

**Intention Mode Lock** –

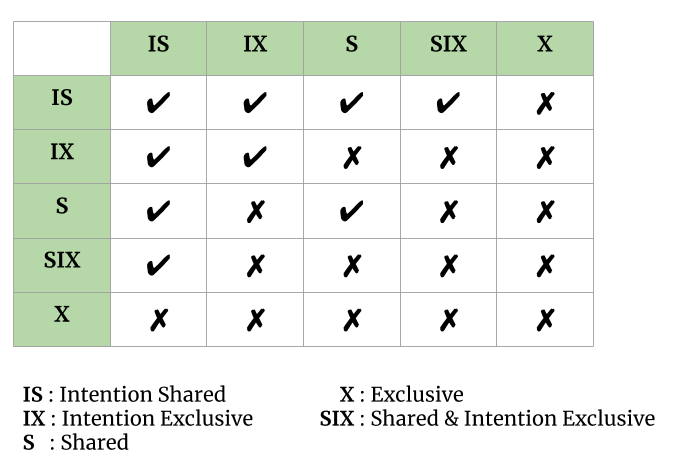
In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

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

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

Shared & Intention-Exclusive (SIX): the sub-tree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive mode locks.

The compatibility matrix for these lock modes are described below:



The multiple-granularity locking protocol uses the intention lock modes to ensure serializability. It requires that a transaction Ti that attempts to lock a node must follow these protocols:

* Transaction Ti must follow the lock-compatibility matrix.
* Transaction Ti must lock the root of the tree first, and it can lock it in any mode.
* Transaction Ti can lock a node in S or IS mode only if Ti currently has the parent of the node locked in either IX or IS mode.
* Transaction Ti can lock a node in X, SIX, or IX mode only if Ti currently has the parent of the node locked in either IX or SIX mode.
* Transaction Ti can lock a node only if Ti has not previously unlocked any node (i.e., Ti is two phase).
* Transaction Ti can unlock a node only if Ti currently has none of the children of the node locked.

As an illustration of the protocol, consider the tree given above and the transactions:

* Say transaction T1 reads record Ra2 in file Fa. Then, T2 needs to lock the database, area A1, and Fa in IS mode (and in that order), and finally to lock Ra2 in S mode.
* Say transaction T2 modifies record Ra9 in file Fa . Then, T2 needs to lock the database, area A1, and file Fa (and in that order) in IX mode, and at last to lock Ra9 in X mode.
* Say transaction T3 reads all the records in file Fa. Then, T3 needs to lock the database and area A1 (and in that order) in IS mode, and at last to lock Fa in S mode.
* Say transaction T4 reads the entire database. It can do so after locking the database in S mode.

Note that transactions T1, T3 and T4 can access the database concurrently. Transaction T2 can execute concurrently with T1, but not with either T3 or T4.

This protocol enhances concurrency and reduces lock overhead.Deadlock are still possible in the multiple-granularity protocol, as it is in the two-phase locking protocol

**Characteristics of Good Concurrency Protocol**

An ideal concurrency control DBMS mechanism has the following objectives:

* Must be resilient to site and communication failures.
* It allows the parallel execution of transactions to achieve maximum concurrency.
* Its storage mechanisms and computational methods should be modest to minimize overhead.
* It must enforce some constraints on the structure of atomic actions of transactions.

**Deadlock:**

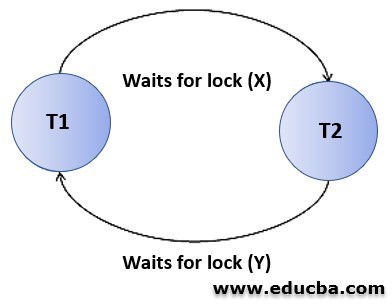
A Deadlock in [DBMS can be termed as the](https://www.educba.com/what-is-dbms/) undesirable condition which appears when a process waits for a resource indefinitely whereas this resource is detained by another process. In order to understand the deadlock concept better, let us consider a transaction T1 which has a lock on a few rows in the table Employee and it requires to update some rows in another table Salary. Also, there exists another transaction T2 that has a lock on the table Salary and it also requires updating a few rows in the Employee table which already is held by the transaction T1. In this situation both the transactions wait for each other to release the lock and the processes end up waiting for each other to release the resources. As a result of the above scenario, none of the tasks gets completed and this is known as deadlock.

**How to detect Deadlock in DBMS?**

The deadlock can be detected by the resource scheduler who checks all the resources allocated to the different processes. The deadlock should be avoided instead of terminating and restarting the transaction so that both resources as well time is not wasted. One of the methods for detecting deadlock is Wait-For-Graph which is suitable for smaller databases.

**Wait-For-Graph:**

A graph is created based on the transactions and locks on the resource in this method. A deadlock occurs if the graph which is created has a closed-loop. For all transactions waiting the resources are maintained by DBMS and also are checked to see if there is any closed loop. Let us consider two transactions T1 and T2 where T1 requests for a resource held by T2. The wait-for-graph in this scenario draws an arrow from T1 to T2 and when the resource is released by T2, the arrow gets deleted. For example, T1 requests for a lock X on a resource, which is held by T2, a directed edge is created from T1 to T2. When T2 releases the resource X, the edge T1 locks the resource and the directed edge between T1 and T2 is dropped.



**How to prevent Deadlock in DBMS?**

In DBMS, all the operations are analyzed and inspected by it to find if there is any possibility of occurrence of deadlock and in case of the possibility of deadlock, the transaction is not allowed to be processed. Primarily, the timestamp at which the transactions have begun is examined by the DMBS and based on this the transactions are ordered. The deadlock can be prevented by using schemes that use the timestamp of the transactions to calculate the occurrence of deadlock.

**1. Wait- Die Scheme**

In this scheme, when a transaction requests for the resource which is already held by another transaction, then the timestamps of the transactions are scanned by the DBMS and the older transaction waits till the resource becomes available. Let us consider two transactions T1 and T2 and the timestamp of the transaction be denoted by TS. If T1 requests for resources held by T2 and a lock exists on T2 by some other transaction,

Below are the steps followed:

* Whether TS(T1) < TS(T2) is examined, and if T1 is the older transaction between T1 and T2 and some resource is held by the transaction T2, then it permits T1 to await the resource to be available for execution.
* If T1 is the older transaction that has held some resource and T2 is waiting for the resource held by T1, then T2 gets killed and later it will be restarted with the same timestamp but with a random delay.

**2. Wound Wait Scheme**

In this scheme, if T1 is the older transaction in between transactions T1 and T2, and when T2 requests for the resource which is held by the transaction T1, then the younger transaction i.e. T2 waits until T1 releases the resource. But when the resource held by the younger transaction T2 is requested by the older transaction T1, T2 is forced by T1 to kill the transaction in order to release the resource held by it and afterward T2 is restarted with a delay but with the same timestamp.

**Recovery from deadlock:**

After detecting the deadlock, the next important step a system has to take is to recover the system from the deadlock state. This can be achieved through rolling back one or more transactions. But the difficult part is to choose one or more transactions as victims.

Recovery from deadlock can be done in three steps;

**1. Selection of victim:** Given a set of deadlocked transactions (transactions that formed cycles), we have to identify the transaction or transactions that are to be rolled-back for successful relief from deadlock state. This is done by identifying the transaction or transactions that are cost minimum. This would mean many things. The following guidelines would help us.

Guidelines to choose victim:

* The length of the transaction – We need to choose the transaction which is younger.
* The data items used by the transaction – The transactions that are used less number of data items.
* The data items that are to be locked – The transaction that needs to lock many more data items compared to that are already locked.
* How many transactions to be rolled back? – We need to choose transaction or transactions that would cause less number of other transactions to be rolled back (cascading rollback)

**2. Rollback:** Once we have identified the transaction or transactions that are to be rolled back, then rollback them. This can be done in two ways;

Full rollback – Simplest of two. Rollback the transaction up to its starting point. That is, abort the transaction and restart. It will also abort other transactions that have used the data item changed by the rolled back transaction.

Partial rollback – it might be the effective one, if the system maintains additional information like the order of lock requests, save points etc.

**3. Starvation:** In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result this transaction never completes can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

**Two Phase Commit Protocol (Distributed Transaction Management):**

Consider we are given with a set of grocery stores where the head of all store wants to query about the available sanitizers inventory at all stores in order to move inventory store to store to make balance over the quantity of sanitizers inventory at all stores. The task is performed by a single transaction **T** that’s component **Tn** at the **nth** store and a store **S0** corresponds to **T0** where the manager is located. The following sequence of activities are performed by **T** are below:

**a)** Component of transaction**( T ) T0** is created at the head-site(head-office).

**b) T0** sends messages to all the stores to order them to create components **Ti.**

**c)** Every **Ti** executes a query at the store “**i”** to discover the quantity of available sanitizers inventory and reports this number to **To.**

**d)** Each store receives instruction and update the inventory level and made shipment to other stores where require.

But there are some problems that we may face during the execution of this process:

1) Atomicity property may be violated because any store(Sn) may be instructed twice to send the inventory that may leave the database in an inconsistent state. To ensure atomicity property Transaction T must either commit at all the sites, or it must abort at all sites.

2) However, the system at store Tn may crash, and the instructions from T0 are never received by Tn because of any network issue and any other reason. So the question arises what will happen to the distributed transaction running whether it abort or commit? Whether it recover or not?

**Two-Phase Commit Protocol:** This protocol is designed with the core intent to resolve the above problems, Consider we have multiple distributed databases which are operated from different servers(sites) let’s say S1, S2, S3, ….Sn. Where every Si made to maintains a separate log record of all corresponding activities and the transition T has also been divided into the subtransactions T1, T2, T3, …., Tn and each Ti are assigned to Si. This all maintains by a separate transaction manager at each Si. We assigned anyone site as a Coordinator.

Some points to be considered regarding this protocol:

a) In a two-phase commit, we assume that each site logs actions at that site, but there is no global log.

b) The coordinator(Ci), plays a vital role in doing confirmation whether the distributed transaction would abort or commit.

c) In this protocol messages are made to send between the coordinator(Ci) and the other sites. As each message is sent, its logs are noted at each sending site, to aid in recovery should it be necessary.

**The two phases of this protocol are as follow:**

Phase-1st–

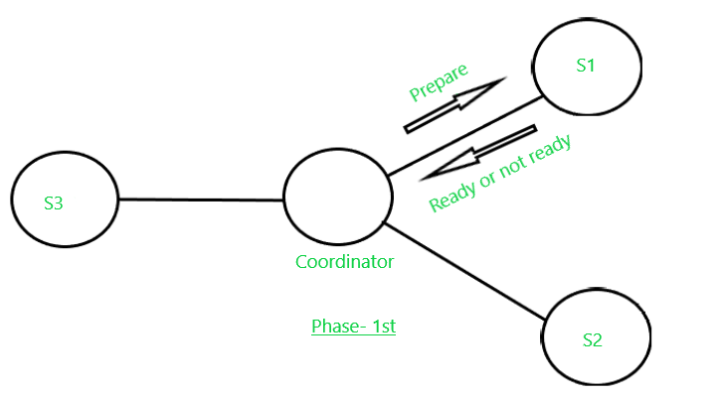
a) Firstly, the coordinator(Ci) places a log record <Prepare T> on the log record at its site.

b) Then, the coordinator(Ci) sends a Prepare T message to all the sites where the transaction(T) executed.

c) Transaction manager at each site on receiving this message Prepare T decides whether to commit or abort its component(portion) of T. The site can delay if the component has not yet completed its activity, but must eventually send a response.

d) If the site doesn’t want to commit, so it must write on log record <no T>, and local Transaction manager sends a message abort T to Ci.

e) If the site wants to commit, it must write on log record <ready T>, and local Transaction manager sends a message ready T to Ci. Once the ready T message at Ci is sent nothing can prevent it to commit its portion of transaction T except Coordinator(Ci).



Phase- 2nd–

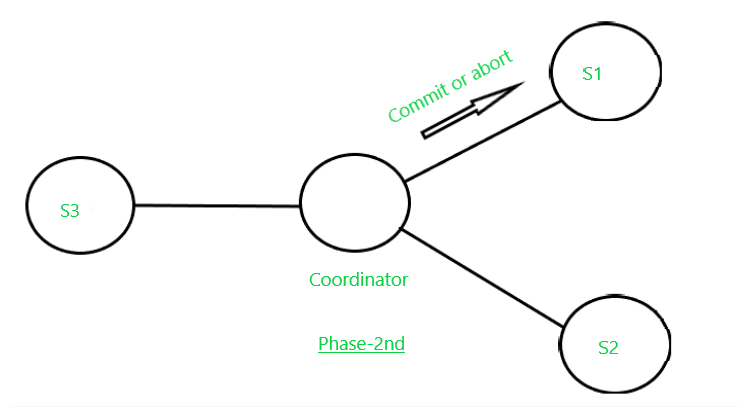
The Second phase started as the response abort T or commit T receives by the coordinator(Ci) from all the sites that are collaboratively executing the transaction T. However, it is possible that some site fails to respond; it may be down, or it has been disconnected by the network.  In that case, after a suitable timeout period will be given, after that time it will treat the site as if it had sent abort T. The fate of the transaction depends upon the following points:

a)  If the coordinator receives ready T from all the participating sites of T, then it decides to commit T. Then, the coordinator writes on its site log record <Commit T>  and sends a message commit T to all the sites involved in T.

b) If a site receives a commit T message, it commits the component of T at that site, and write it in log records <Commit T>.

c) If a site receives the message abort T, it aborts T and writes the log record <Abort T>.

d) However, if the coordinator has received abort T from one or more sites, it logs <Abort T> at its site and then sends abort T messages to all sites involved in transaction T.



**Disadvantages:**

a) The major disadvantage of the Two-phase commit protocol is faced when the Coordinator site failure may result in blocking, so a decision either to commit or abort Transaction(T) may have to be postponed until coordinator recovers.

b) Blocking Problem: Consider a scenario, if a Transaction(T) holds locks on data-items of active sites, but amid the execution, if the coordinator fails and the active sites keep no additional log-record except <readt T> like <abort> or <commit>. So, it becomes impossible to determine what decision has been made(whether to <commit> /<abort>). So, In that case, the final decision is delayed until the Coordinator is restored or fixed. In some cases, this may take a day or long hours to restore and during this time period, the locked data items remain inaccessible for other transactions(Ti). This problem is known as Blocking Problem.