# Transaction

- o   The transaction is a set of logically related operation. It contains a group of tasks.
- o   A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.

**Example:** Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:

**X's Account**

1. Open_Account(X)
2. Old_Balance = X.balance
3. New_Balance = Old_Balance - 800
4. X.balance = New_Balance
5. Close_Account(X)

**Y's Account**

Open_Account(Y)

1. Old_Balance = Y.balance
2. New_Balance = Old_Balance + 800
3. Y.balance = New_Balance
4. Close_Account(Y)

# Operations of Transaction:

Following are the main operations of transaction:

**Read(X):** Read operation is used to read the value of X from the database and stores it in a buffer in main memory.

**Write(X):** Write operation is used to write the value back to the database from the buffer.

Let's take an example to debit transaction from an account which consists of following operations:

1. 1.  R(X);
2. 2.  X = X - 500;
3. 3.  W(X);

Let's assume the value of X before starting of the transaction is 4000.

- The first operation reads X's value from database and stores it in a buffer.
- The second operation will decrease the value of X by 500. So buffer will contain 3500.
- The third operation will write the buffer's value to the database. So X's final value will be 3500.

But it may be possible that because of the failure of hardware, software or power, etc. that transaction may fail before finished all the operations in the set.

**For example:** If in the above transaction, the debit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the bank.

To solve this problem, we have two important operations:

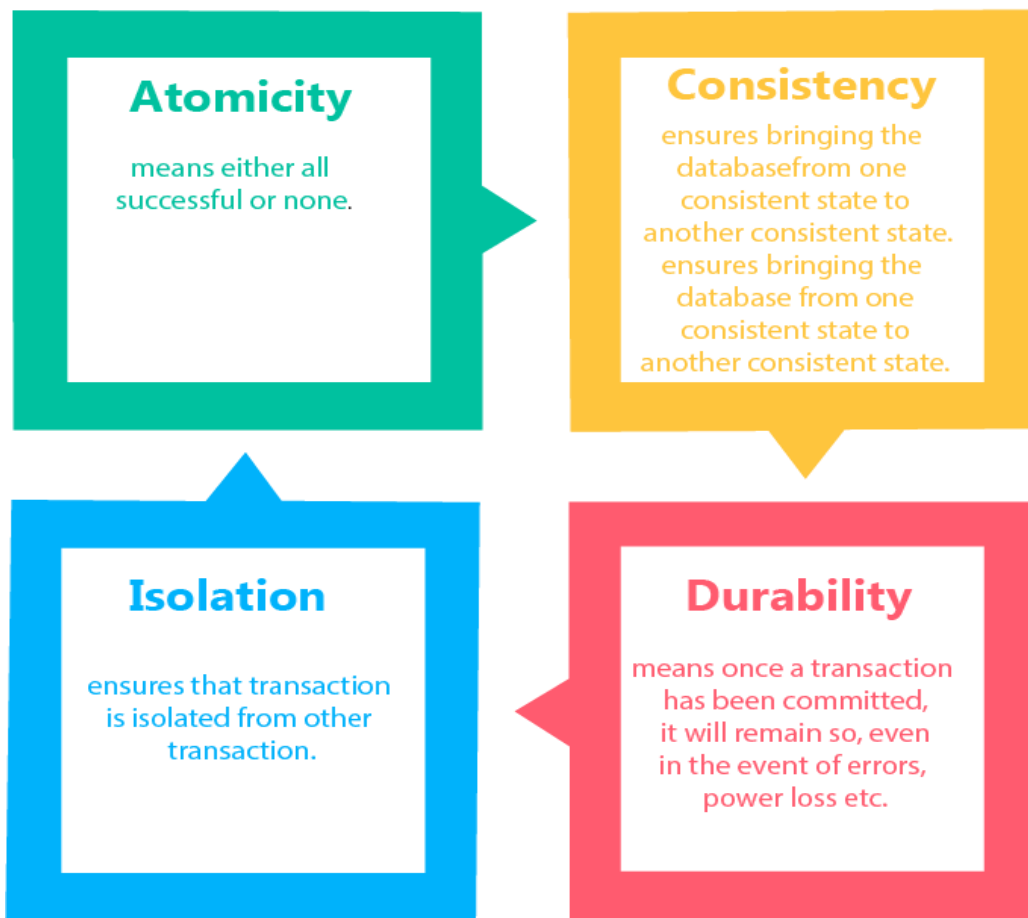**Commit:** It is used to save the work done permanently.

**Rollback:** It is used to undo the work done.

# Transaction property

The transaction has the four properties. These are used to maintain consistency in a database, before and after the transaction.

## Property of Transaction

1. Atomicity
2. Consistency
3. Isolation
4. Durability

## Atomicity

- It states that all operations of the transaction take place at once if not, the transaction is aborted.

- There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.

Atomicity involves the following two operations:

**Abort:** If a transaction aborts then all the changes made are not visible.

**Commit:** If a transaction commits then all the changes made are visible.

**Example:** Let's assume that following transaction T consisting of T1 and T2. A consists of Rs 600 and B consists of Rs 300. Transfer Rs 100 from account A to account B.

| T1 | T2 |
| --- | --- |
| Read(A)<br>A:=                                                          A-100<br>Write(A) | Read(B)<br>Y:=                                                          Y+100<br>Write(B) |

After completion of the transaction, A consists of Rs 500 and B consists of Rs 400.

If the transaction T fails after the completion of transaction T1 but before completion of transaction T2, then the amount will be deducted from A but not added to B. This shows the inconsistent database state. In order to ensure correctness of database state, the transaction must be executed in entirety.

# Consistency

- o The integrity constraints are maintained so that the database is consistent before and after the transaction.
- o The execution of a transaction will leave a database in either its prior stable state or a new stable state.
- o The consistent property of database states that every transaction sees a consistent database instance.
- o The transaction is used to transform the database from one consistent state to another consistent state.

**For example:** The total amount must be maintained before or after the transaction.

1. Total before T occurs = 600+300=900
2. Total after T occurs= 500+400=900

Therefore, the database is consistent. In the case when T1 is completed but T2 fails, then inconsistency will occur.

# Isolation

- o It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.

- In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1 ends.
- The concurrency control subsystem of the DBMS enforced the isolation property.

# Durability

- The durability property is used to indicate the performance of the database's consistent state. It states that the transaction made the permanent changes.
- They cannot be lost by the erroneous operation of a faulty transaction or by the system failure. When a transaction is completed, then the database reaches a state known as the consistent state. That consistent state cannot be lost, even in the event of a system's failure.
- The recovery subsystem of the DBMS has the responsibility of Durability property.

# States of Transaction

In a database, the transaction can be in one of the following states -
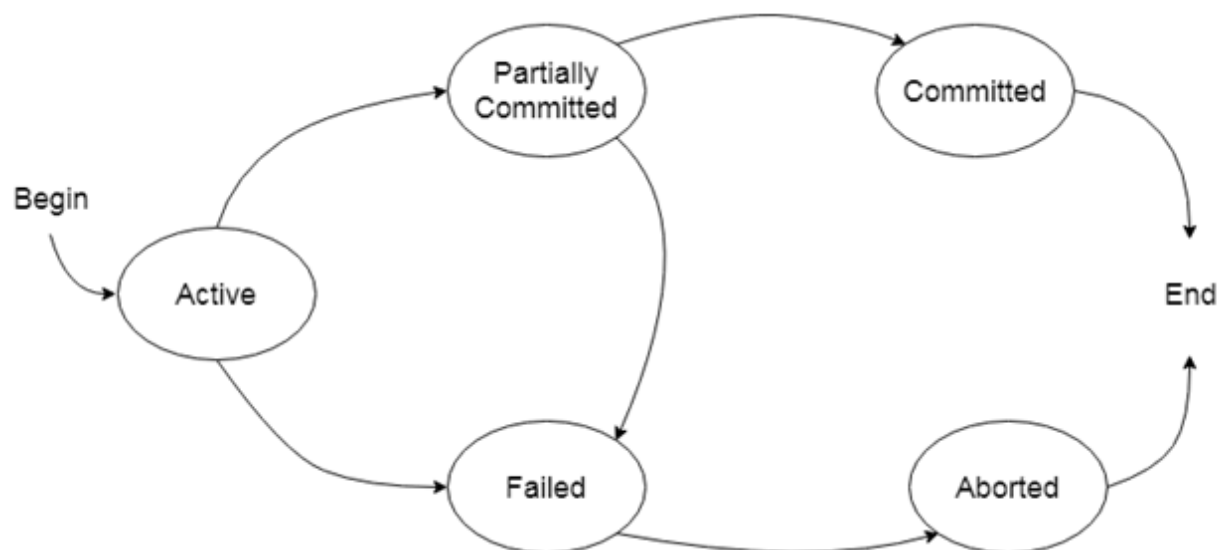


## Active state

- The active state is the first state of every transaction. In this state, the transaction is being executed.
- For example: Insertion or deletion or updating a record is done here. But all the records are still not saved to the database.

## Partially committed

- o In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database.
- o In the total mark calculation example, a final display of the total marks step is executed in this state.

## Committed

A transaction is said to be in a committed state if it executes all its operations successfully. In this state, all the effects are now permanently saved on the database system.

## Failed state

- o If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
- o In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

## Aborted

- o If any of the checks fail and the transaction has reached a failed state then the database recovery system will make sure that the database is in its previous consistent state. If not then it will abort or roll back the transaction to bring the database into a consistent state.
- o If the transaction fails in the middle of the transaction then before executing the transaction, all the executed transactions are rolled back to its consistent state.
- o After aborting the transaction, the database recovery module will select one of the two operations:
    1. Re-start the transaction
    2. Kill the transaction

# Schedule

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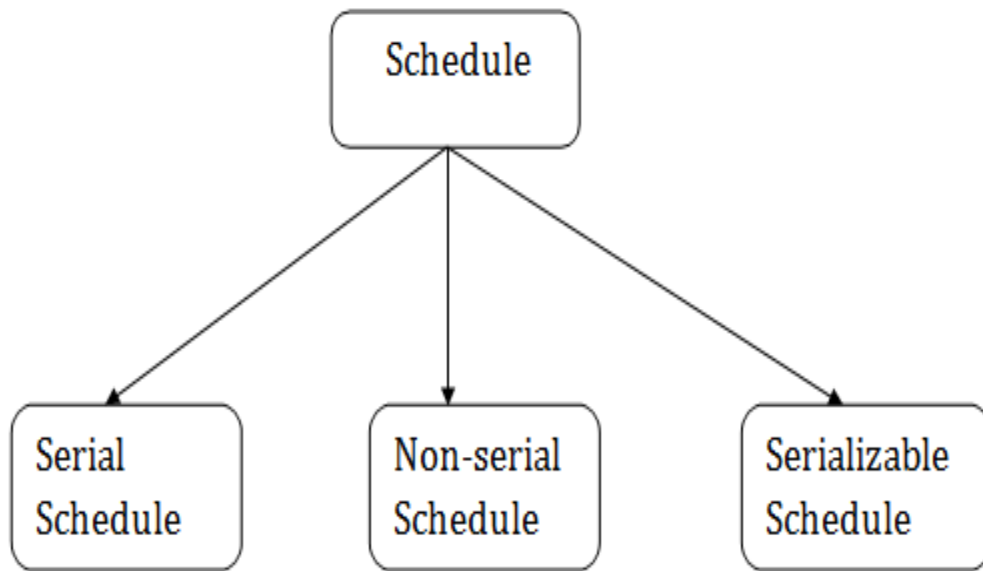
# 1. Serial Schedule

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

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

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

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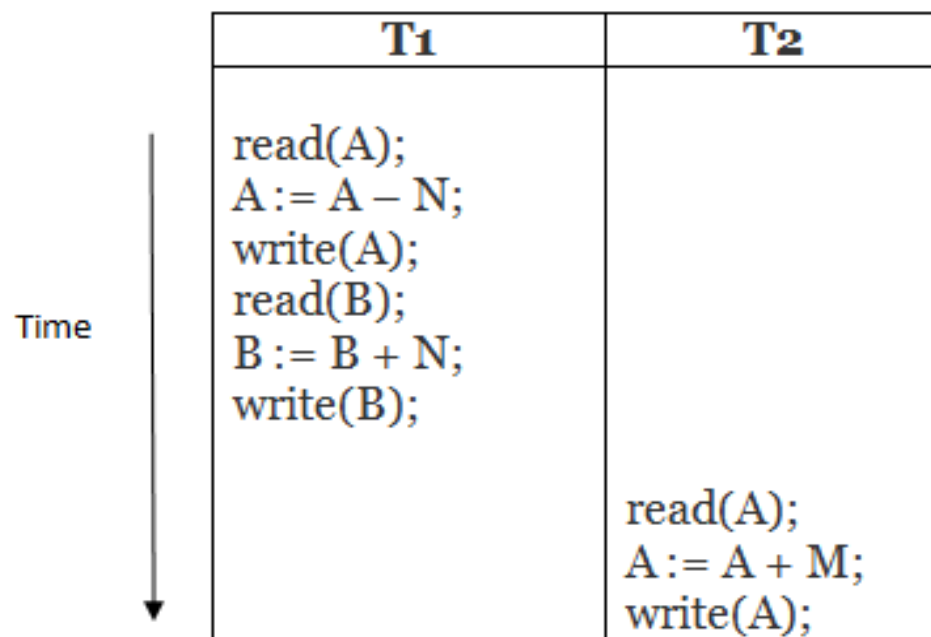
# 2. Non-serial Schedule

   o   If interleaving of operations is allowed, then there will be non-serial schedule.
   o   It contains many possible orders in which the system can execute the individual operations of the transactions.

- In the given figure (c) and (d), Schedule C and Schedule D are the non-serial schedules. It has interleaving of operations.
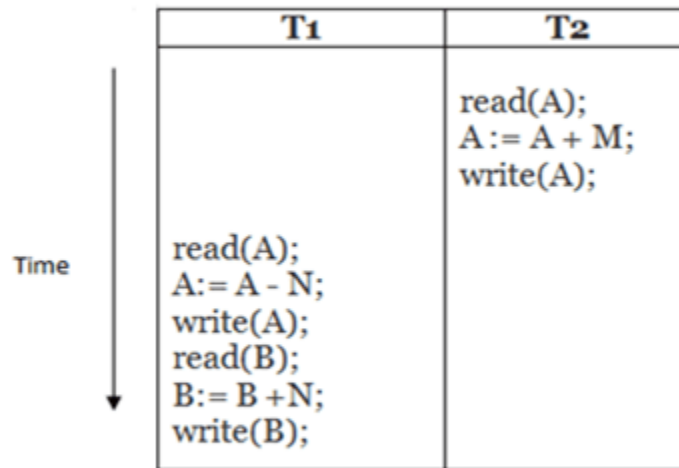
# 3. Serializable schedule

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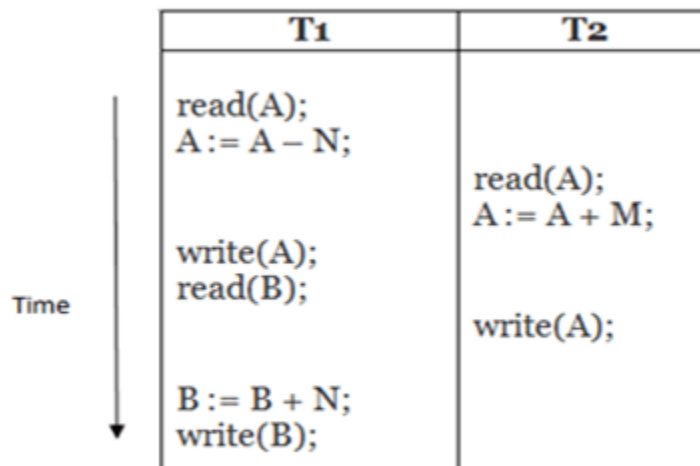- 

**(a)**

Time

| T1 | T2 |
|---|---|
| read(A);<br>A := A − N;<br>write(A);<br>read(B);<br>B := B + N;<br>write(B); | |
| | read(A);<br>A := A + M;<br>write(A); |

**Schedule A**

**(b)**

| T1 | T2 |
|---|---|
| | read(A);<br>A := A + M;<br>write(A); |
| read(A);<br>A := A - N;<br>write(A);<br>read(B);<br>B := B + N;<br>write(B); | |

Time

**Schedule B**

**(c)**

| T1 | T2 |
|---|---|
| read(A);<br>A := A - N; | |
| | read(A);<br>A := A + M; |
| write(A);<br>read(B); | |
| | write(A); |
| B := B + N;<br>write(B); | |

Time

**Schedule C**

**(d)**

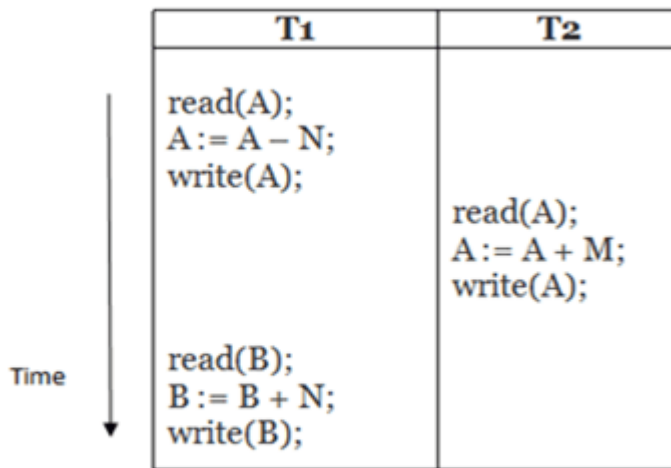| | T₁ | T₂ |
|---|---|---|
| | read(A);<br>A := A − N;<br>write(A); | |
| | | read(A);<br>A := A + M;<br>write(A); |
| Time | read(B);<br>B := B + N;<br>write(B); | |

**Schedule D**

- **Here,**

- Schedule A and Schedule B are serial schedule.
- Schedule C and Schedule D are Non-serial schedule.

# Testing of Serializability

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

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

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

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

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

## Precedence graph for Schedule S
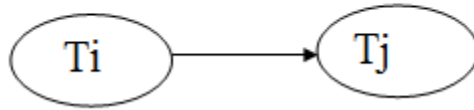


- o  If a precedence graph contains a single edge Ti → Tj, then all the instructions of Ti are executed before the first instruction of Tj is executed.
- o  If a precedence graph for schedule S contains a cycle, then S is non-serializable. If the precedence graph has no cycle, then S is known as serializable.

**For example:**

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

Time

**Schedule S1**

**Explanation:**

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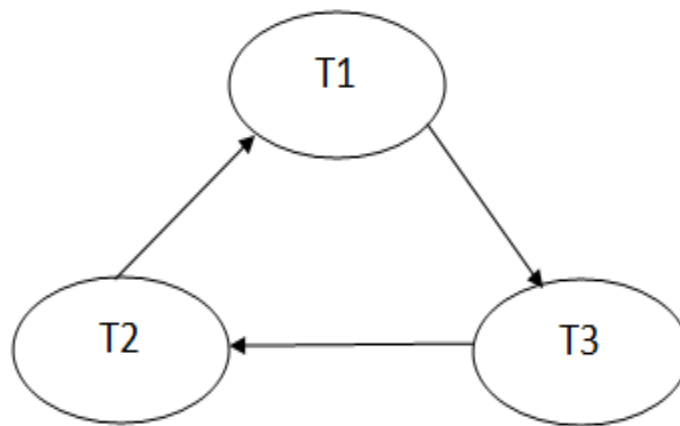**Write(C):** C is subsequently read by T1, so add edge T3 → T1
**Write(A):** A is subsequently read by T2, so add edge T1 → T2
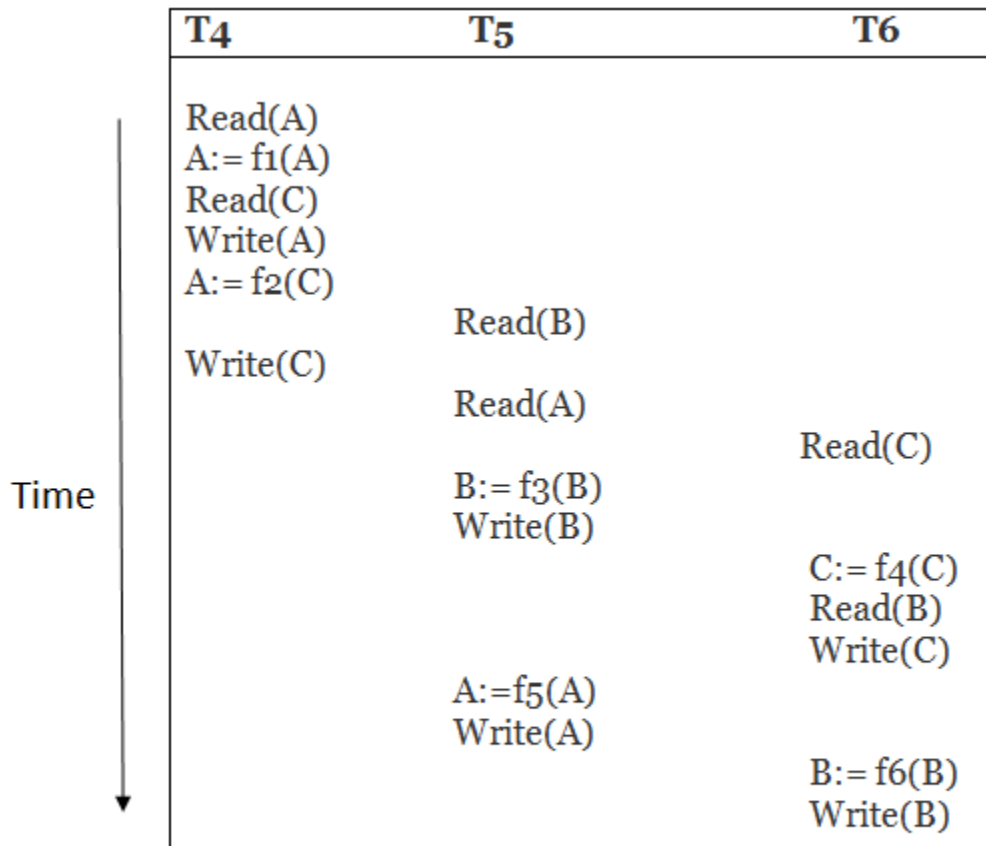**Write(A):** In T2, no subsequent reads to A, so no new edges
**Write(C):** In T1, no subsequent reads to C, so no new edges
**Write(B):** In T3, no subsequent reads to B, so no new edges

## Precedence graph for schedule S1:



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

| T4 | T5 | T6 |
|---|---|---|
| Read(A)<br>A:= f1(A)<br>Read(C)<br>Write(A)<br>A:= f2(C) | | |
| | Read(B) | |
| Write(C) | | |
| | Read(A) | |
| | | Read(C) |
| | B:= f3(B)<br>Write(B) | |
| | | C:= f4(C)<br>Read(B)<br>Write(C) |
| | A:=f5(A)<br>Write(A) | |
| | | B:= f6(B)<br>Write(B) |

Time

**Schedule S2**

**Explanation:**

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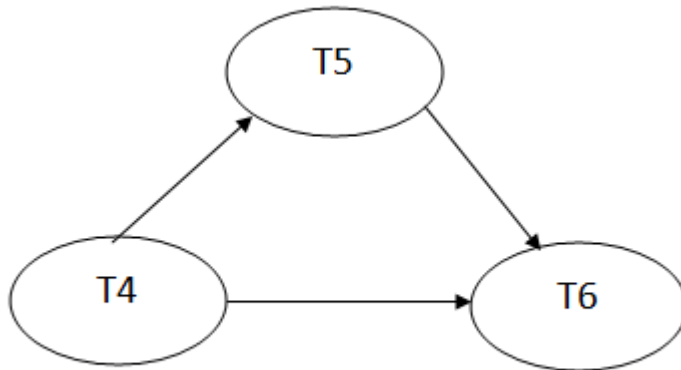**Write(C):** C is subsequently read by T6, so add edge T4 → T6
**Write(B):** A is subsequently read by T6, so add edge T5 → T6
**Write(C):** In T6, no subsequent reads to C, so no new edges
**Write(A):** In T5, no subsequent reads to A, so no new edges
**Write(B):** In T6, no subsequent reads to B,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

**Precedence graph for schedule S2:**



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

# Conflict Serializable Schedule

- o A schedule is called conflict serializability if after swapping of non-conflicting operations, it can transform into a serial schedule.
- o The schedule will be a conflict serializable if it is conflict equivalent to a serial schedule.

## Conflicting Operations

The two operations become conflicting if all conditions satisfy:

1. Both belong to separate transactions.
2. They have the same data item.
3. They contain at least one write operati

## Example:

Swapping is possible only if S1 and S2 are logically equal.

**1. T1: Read(A)    T2: Read(A)**

| T1 | T2 |
|---|---|
| Read(A) | |
| | Read(A) |

Swapped ⟹

| T1 | T2 |
|---|---|
| | Read(A) |
| Read(A) | |

**Schedule S1**                    **Schedule S2**

Here, S1 = S2. That means it is non-conflict.

**2. T1: Read(A)    T2: Write(A)**

| T1 | T2 |
|---|---|
| Read(A) | |
| | Write(A) |

Swapped ⟹
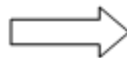
| T1 | T2 |
|---|---|
| | Write(A) |
| Read(A) | |

**Schedule S1**                    **Schedule S2**

Here, S1 ≠ S2. That means it is conflict.

# Conflict Equivalent

In the conflict equivalent, one can be transformed to another by swapping non-conflicting operations. In the given example, S2 is conflict equivalent to S1 (S1 can be converted to S2 by swapping non-conflicting operations).

Two schedules are said to be conflict equivalent if and only if:

1. They contain the same set of the transaction.

2. If each pair of conflict operations are ordered in the same way.

## Example:

**Non-serial schedule**

| T1 | T2 |
|---|---|
| Read(A)<br>Write(A) | |
| | Read(A)<br>Write(A) |
| Read(B)<br>Write(B) | |
| | Read(B)<br>Write(B) |

Schedule S1

**Serial Schedule**

| T1 | T2 |
|---|---|
| Read(A)<br>Write(A)<br>Read(B)<br>Write(B) | |
| | Read(A)<br>Write(A)<br>Read(B)<br>Write(B) |

Schedule S2

Schedule S2 is a serial schedule because, in this, all operations of T1 are performed before starting any operation of T2. Schedule S1 can be transformed into a serial schedule by swapping non-conflicting operations of S1.

**After swapping of non-conflict operations, the schedule S1 becomes:**

| T1 | T2 |
|---|---|
| Read(A)<br>Write(A)<br>Read(B)<br>Write(B) | |
| | Read(A)<br>Write(A)<br>Read(B)<br>Write(B) |

Since, S1 is conflict serializable.

# View Serializability

- o A schedule will view serializable if it is view equivalent to a serial schedule.

- o If a schedule is conflict serializable, then it will be view serializable.

- o The view serializable which does not conflict serializable contains blind writes.

## View Equivalent

Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:

## 1. Initial Read

An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.

| T1 | T2 |
|---|---|
| Read(A) | |
| | Write(A) |
| | |

**Schedule S1**

| T1 | T2 |
|---|---|
| | Write(A) |
| Read(A) | |
| | |

**Schedule S2**

Above two schedules are view equivalent because Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

## 2. Updated Read

In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.

| T1 | T2 | T3 |
|---|---|---|
| Write(A) | | |
| | Write(A) | |
| | | Read(A) |

**Schedule S1**

| T1 | T2 | T3 |
|---|---|---|
| | Write(A) | |
| Write(A) | | |
| | | Read(A) |

**Schedule S2**

Above two schedules are not view equal because, in S1, T3 is reading A updated by T2 and in S2, T3 is reading A updated by T1.

## 3. Final Write

A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.

| T1 | T2 | T3 |
|---|---|---|
| Write(A) | | |
| | Read(A) | |
| | | Write(A) |

**Schedule S1**

| T1 | T2 | T3 |
|---|---|---|
| | Read(A) | |
| Write(A) | | |
| | | Write(A) |

**Schedule S2**

Above two schedules is view equal because Final write operation in S1 is done by T3 and in S2, the final write operation is also done by T3.

**Example:**

| T1 | T2 | T3 |
|---|---|---|
| Read(A) | | |
| | Write(A) | |
| Write(A) | | |
| | | Write(A) |

**Schedule S**

With 3 transactions, the total number of possible schedule

1. = 3! = 6
2. S1 = <T1 T2 T3>
3. S2 = <T1 T3 T2>
4. S3 = <T2 T3 T1>
5. S4 = <T2 T1 T3>
6. S5 = <T3 T1 T2>
7. S6 = <T3 T2 T1>

**Taking first schedule S1:**

| T1 | T2 | T3 |
|---|---|---|
| Read(A) Write(A) | Write(A) | Write(A) |

**Schedule S1**

**Step 1:** final updation on data items

In both schedules S and S1, there is no read except the initial read that's why we don't need to check that condition.

**Step 2:** Initial Read

The initial read operation in S is done by T1 and in S1, it is also done by T1.

**Step 3:** Final Write

The final write operation in S is done by T3 and in S1, it is also done by T3. So, S and S1 are view Equivalent.

The first schedule S1 satisfies all three conditions, so we don't need to check another schedule.

**Hence, view equivalent serial schedule is:**

1. T1 → T2 → T3

# Recoverability of Schedule

2. Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.

3.

| T1 | T1's buffer space | T2 | T2's buffer space | Database |
|---|---|---|---|---|
| | | | | A = 6500 |
| Read(A); | A = 6500 | | | A = 6500 |
| A = A - 500; | A = 6000 | | | A = 6500 |
| Write(A); | A = 6000 | | | A = 6000 |
| | | Read(A); | A = 6000 | A = 6000 |
| | | A =A + 1000; | A = 7000 | A = 6000 |
| | | Write(A); | A = 7000 | A = 7000 |
| | | Commit; | | |
| Failure Point | | | | |
| Commit; | | | | |

4. The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1, but T2 can't be rollback because it already committed. So this type of schedule is known as irrecoverable schedule.

5. **Irrecoverable schedule:** The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit.

6.

| T1 | T1's buffer space | T2 | T2's buffer space | Database |
|---|---|---|---|---|
| | | | | A = 6500 |
| Read(A); | A = 6500 | | | A = 6500 |
| A = A - 500; | A = 6000 | | | A = 6500 |
| Write(A); | A = 6000 | | | A = 6000 |
| | | Read(A); | A = 6000 | A = 6000 |
| | | A =A + 1000; | A = 7000 | A = 6000 |
| | | Write(A); | A = 7000 | A = 7000 |
| Failure Point | | | | |
| Commit; | | | | |
| | | Commit; | | |

7. The above table 2 shows a schedule with two transactions. Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, we have to rollback T1. T2 should be rollback because T2 has read the value written by T1. As it has not committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with cascade rollback.

8. **Recoverable with cascading rollback:** The schedule will be recoverable with cascading rollback if Tj reads the updated value of Ti. Commit of Tj is delayed till commit of Ti.

9.

| T1 | T1's buffer space | T2 | T2's buffer space | Database |
|---|---|---|---|---|
| | | | | A = 6500 |
| Read(A); | A = 6500 | | | A = 6500 |
| A = A - 500; | A = 6000 | | | A = 6500 |
| Write(A); | A = 6000 | | | A = 6000 |
| Commit; | | Read(A); | A = 6000 | A = 6000 |
| | | A =A + 1000; | A = 7000 | A = 6000 |
| | | Write(A); | A = 7000 | A = 7000 |
| | | Commit; | | |

10. The above Table 3 shows a schedule with two transactions. Transaction T1 reads and write A and commits, and that value is read and written by T2. So this is a cascade less recoverable schedule.

# Failure Classification

To find that where the problem has occurred, we generalize a failure into the following categories:

1. Transaction failure
2. System crash
3. Disk failure

## 1. Transaction failure

The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transaction or process is hurt, then this is called as transaction failure.

Reasons for a transaction failure could be -

1. **Logical errors:** If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.

2. **Syntax error:** It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. **For example,** The system aborts an active transaction, in case of deadlock or resource unavailability.

## 2. System Crash

- o System failure can occur due to power failure or other hardware or software failure. **Example:** Operating system error.

  **Fail-stop assumption:** In the system crash, non-volatile storage is assumed not to be corrupted.

## 3. Disk Failure

- o It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.

- o Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

# Log-Based Recovery

- o The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.

- o If any operation is performed on the database, then it will be recorded in the log.

- o But the process of storing the logs should be done before the actual transaction is applied in the database.

Let's assume there is a transaction to modify the City of a student. The following logs are written for this transaction.

- o When the transaction is initiated, then it writes 'start' log.
    1. <Tn, Start>
- o When the transaction modifies the City from 'Noida' to 'Bangalore', then another log is written to the file.

    1. <Tn, City, 'Noida', 'Bangalore' >
- o When the transaction is finished, then it writes another log to indicate the end of the transaction.

    1. <Tn, Commit>

There are two approaches to modify the database:

## 1. Deferred database modification:

- o The deferred modification technique occurs if the transaction does not modify the database until it has committed.

- o In this method, all the logs are created and stored in the stable storage, and the database is updated when a transaction commits.

## 2. Immediate database modification:

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

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

# Recovery using Log records

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

1. If the log contains the record <Ti, Start> and <Ti, Commit> or <Ti, Commit>, then the Transaction Ti needs to be redone.

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

# Checkpoint

- o The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.

- o The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked, and the transaction is executed then using the steps of the transaction, the log files will be created.

- o When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.

- The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

# Recovery using Checkpoint

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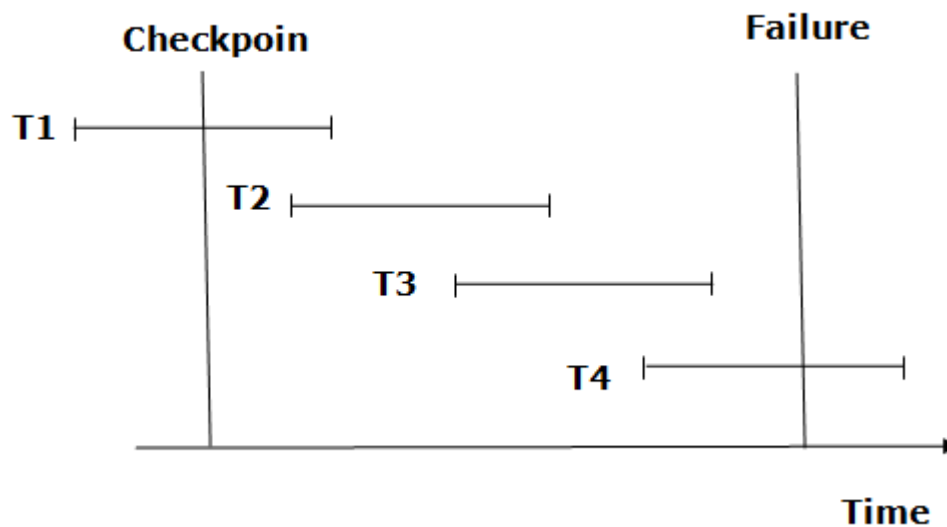


- The recovery system reads log files from the end to start. It reads log files from T4 to T1.
- Recovery system maintains two lists, a redo-list, and an undo-list.
- The transaction is put into redo state if the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>. In the redo-list and their previous list, all the transactions are removed and then redone before saving their logs.
- **For example:** In the log file, transaction T2 and T3 will have <Tn, Start> and <Tn, Commit>. The T1 transaction will have only <Tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts T1, T2 and T3 transaction into redo list.
- The transaction is put into undo state if the recovery system sees a log with <Tn, Start> but no commit or abort log found. In the undo-list, all the transactions are undone, and their logs are removed.

**For example:** Transaction T4 will have <Tn, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed amid.

# Deadlock in DBMS

A deadlock is a condition where two or more transactions are waiting indefinitely for one another to give up locks. Deadlock is said to be one of the most feared complications in DBMS as no task ever gets finished and is in waiting state forever.

**For example:** In the student table, transaction T1 holds a lock on some rows and needs to update some rows in the grade table. Simultaneously, transaction T2 holds locks on some rows in the grade table and needs to update the rows in the Student table held by Transaction T1.

Now, the main problem arises. Now Transaction T1 is waiting for T2 to release its lock and similarly, transaction T2 is waiting for T1 to release its lock. All activities come to a halt state and remain at a standstill. It will remain in a standstill until the DBMS detects the deadlock and aborts one of the transactions.
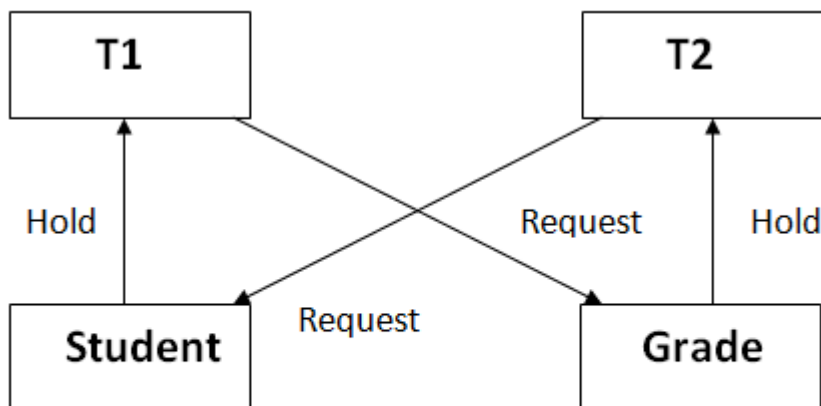


**Figure:** Deadlock in DBMS

## Deadlock Avoidance

o  When a database is stuck in a deadlock state, then it is better to avoid the database rather than aborting or restating the database. This is a waste of time and resource.

o  Deadlock avoidance mechanism is used to detect any deadlock situation in advance. A method like "wait for graph" is used for detecting the deadlock situation but this method

is suitable only for the smaller database. For the larger database, deadlock prevention method can be used.

# Deadlock Detection

In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not. The lock manager maintains a Wait forthe graph to detect the deadlock cycle in the database.

## Wait for Graph

- o   This is the suitable method for deadlock detection. In this method, a graph is created based on the transaction and their lock. If the created graph has a cycle or closed loop, then there is a deadlock.

- o   The wait for the graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph.

The wait for a graph for the above scenario is shown below:
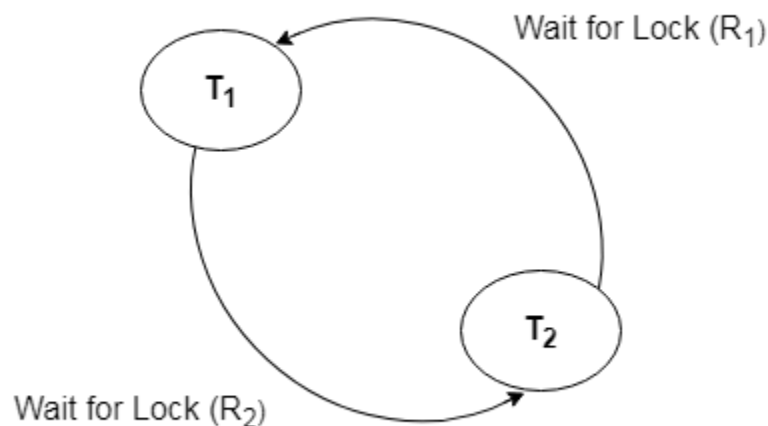
# Deadlock Prevention

- o Deadlock prevention method is suitable for a large database. If the resources are allocated in such a way that deadlock never occurs, then the deadlock can be prevented.

- o The Database management system analyzes the operations of the transaction whether they can create a deadlock situation or not. If they do, then the DBMS never allowed that transaction to be executed.

## Wait-Die scheme

In this scheme, if a transaction requests for a resource which is already held with a conflicting lock by another transaction then the DBMS simply checks the timestamp of both transactions. It allows the older transaction to wait until the resource is available for execution.

Let's assume there are two transactions Ti and Tj and let TS(T) is a timestamp of any transaction T. If T2 holds a lock by some other transaction and T1 is requesting for resources held by T2 then the following actions are performed by DBMS:

1. Check if TS(Ti) < TS(Tj) - If Ti is the older transaction and Tj has held some resource, then Ti is allowed to wait until the data-item is available for execution. That means if the older transaction is waiting for a resource which is locked by the younger transaction, then the older transaction is allowed to wait for resource until it is available.

2. Check if TS($T_i$) < TS(Tj) - If Ti is older transaction and has held some resource and if Tj is waiting for it, then Tj is killed and restarted later with the random delay but with the same timestamp.

## Wound wait scheme

- o In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then older transaction forces younger one to kill the transaction and release the resource. After the minute delay, the younger transaction is restarted but with the same timestamp.

- o If the older transaction has held a resource which is requested by the Younger transaction, then the younger transaction is asked to wait until older releases it.

# DBMS Concurrency Control

Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on a database.

But before knowing about concurrency control, we should know about concurrent execution.

## Concurrent Execution in DBMS

- o In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.

- o While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.

- o The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database. Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

## Problems with Concurrent Execution

In a database transaction, the two main operations are **READ** and **WRITE** operations. So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent. So, the following problems occur with the Concurrent Execution of the operations:

### Problem 1: Lost Update Problems (W - W Conflict)

The problem occurs *when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent.*

**For example:**

**Consider the below diagram where two transactions $T_X$ and $T_Y$, are performed on the same account A where the balance of account A is $300.**

| Time | Tx | Ty |
|------|------|------|
| $t_1$ | READ (A) | — |
| $t_2$ | A = A - 50 | |
| $t_3$ | — | READ (A) |
| $t_4$ | — | A = A + 100 |
| $t_5$ | — | — |
| $t_6$ | WRITE (A) | — |
| $t_7$ | | WRITE (A) |

LOST UPDATE PROBLEM

- o    At time t1, transaction $T_X$ reads the value of account A, i.e., $300 (only read).
- o    At time t2, transaction $T_X$ deducts $50 from account A that becomes $250 (only deducted and not updated/write).
- o    Alternately, at time t3, transaction $T_Y$ reads the value of account A that will be $300 only because $T_X$ didn't update the value yet.
- o    At time t4, transaction $T_Y$ adds $100 to account A that becomes $400 (only added but not updated/write).
- o    At time t6, transaction $T_X$ writes the value of account A that will be updated as $250 only, as $T_Y$ didn't update the value yet.
- o    Similarly, at time t7, transaction $T_Y$ writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by $T_X$ is lost, i.e., $250 is lost.

Hence data becomes incorrect, and database sets to inconsistent.

## Dirty Read Problems (W-R Conflict)

The dirty read problem occurs *when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is*

*accessed by another transaction. There comes the Read-Write Conflict between both transactions.*

**For example:**

**Consider two transactions T$_X$ and T$_Y$ in the below diagram performing read/write operations on account A where the available balance in account A is $300:**

| Time | T$_X$ | T$_Y$ |
|------|------|------|
| t$_1$ | READ (A) | — |
| t$_2$ | A = A + 50 | — |
| t$_3$ | WRITE (A) | — |
| t$_4$ | — | READ (A) |
| t$_5$ | SERVER DOWN ROLLBACK | — |

**DIRTY READ PROBLEM**

- At time t1, transaction T$_X$ reads the value of account A, i.e., $300.
- At time t2, transaction T$_X$ adds $50 to account A that becomes $350.
- At time t3, transaction T$_X$ writes the updated value in account A, i.e., $350.
- Then at time t4, transaction T$_Y$ reads account A that will be read as $350.
- Then at time t5, transaction T$_X$ rollbacks due to server problem, and the value changes back to $300 (as initially).
- But the value for account A remains $350 for transaction T$_Y$ as committed, which is the dirty read and therefore known as the Dirty Read Problem.

## Unrepeatable Read Problem (W-R Conflict)

*Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.*

**For example:**

**Consider two transactions, T$_X$ and T$_Y$, performing the read/write operations on account A, having an available balance = \$300. The diagram is shown below:**

| Time | T$_X$ | T$_Y$ |
|------|-------|-------|
| t$_1$ | READ (A) | — |
| t$_2$ | — | READ (A) |
| t$_3$ | — | A = A + 100 |
| t$_4$ | — | WRITE (A) |
| t$_5$ | READ (A) | — |

**UNREPEATABLE READ PROBLEM**

o  At time t1, transaction T$_X$ reads the value from account A, i.e., \$300.

o  At time t2, transaction T$_Y$ reads the value from account A, i.e., \$300.

o  At time t3, transaction T$_Y$ updates the value of account A by adding \$100 to the available balance, and then it becomes \$400.

o  At time t4, transaction T$_Y$ writes the updated value, i.e., \$400.

o  After that, at time t5, transaction T$_X$ reads the available value of account A, and that will be read as \$400.

o  It means that within the same transaction T$_X$, it reads two different values of account A, i.e., \$ 300 initially, and after updation made by transaction T$_Y$, it reads \$400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.

Thus, in order to maintain consistency in the database and avoid such problems that take place in concurrent execution, management is needed, and that is where the concept of Concurrency Control comes into role.

# Concurrency Control

Concurrency Control is the working concept that is required for controlling and managing the concurrent execution of database operations and thus avoiding the inconsistencies in the database. Thus, for maintaining the concurrency of the database, we have the concurrency control protocols.

## Concurrency Control Protocols

The concurrency control protocols ensure the *atomicity, consistency, isolation, durability* and *serializability* of the concurrent execution of the database transactions. Therefore, these protocols are categorized as:

- o Lock Based Concurrency Control Protocol
- o Time Stamp Concurrency Control Protocol
- o Validation Based Concurrency Control Protocol

We will understand and discuss each protocol one by one in our next sections.

# Lock-Based Protocol

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

**1. Shared lock:**

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

**2. Exclusive lock:**

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

# There are four types of lock protocols available:

## 1. Simplistic lock protocol

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

## 2. Pre-claiming Lock Protocol

- o Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.

- o Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.

- o If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.

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

- o until all the locks are granted.
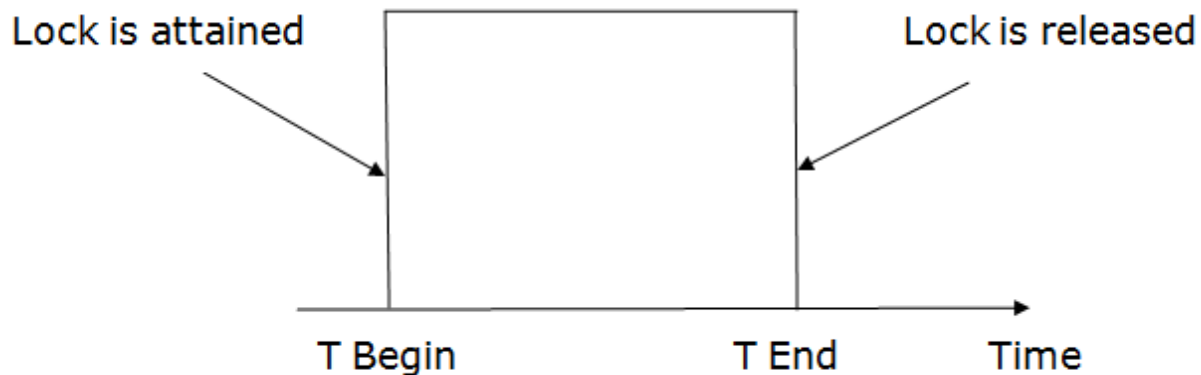


## 3. Two-phase locking (2PL)

- o The two-phase locking protocol divides the execution phase of the transaction into three parts.

- o In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.

- o   In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
- o   In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.
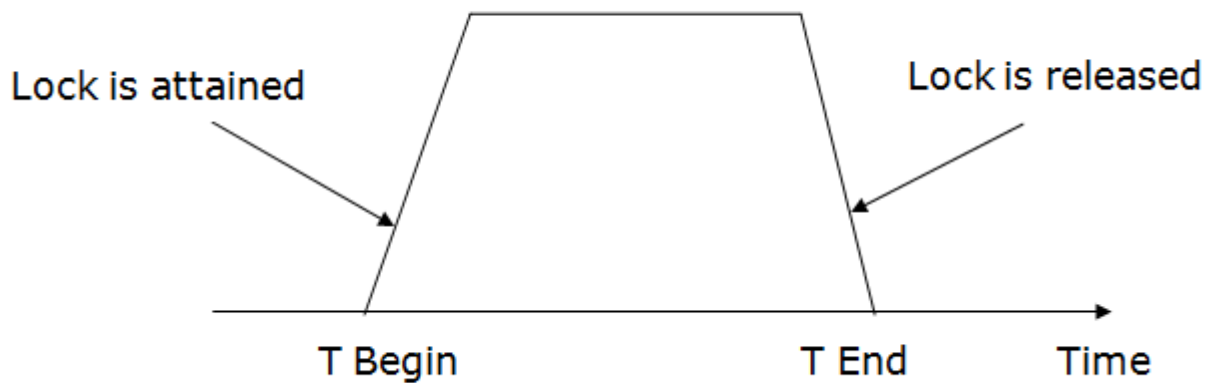


There are two phases of 2PL:

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

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

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

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

**Example:**

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

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

**Transaction T1:**

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

**Transaction T2:**

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

# 4. Strict Two-phase locking (Strict-2PL)

- o The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.

- o The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.

- o Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.

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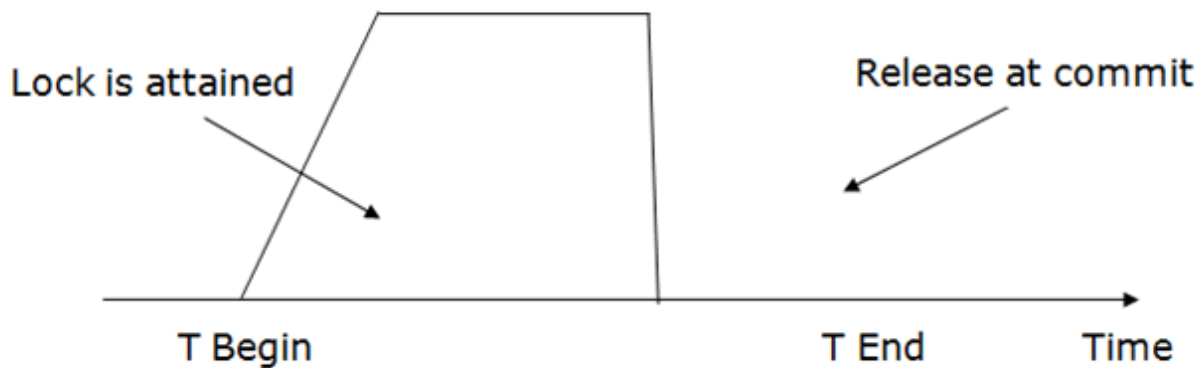


It does not have cascading abort as 2PL does.