TRANSACTION MANAGEMENT

**Topic:Transaction concept**:

* A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
* A transaction can be defined as a logical unit of work on the database. This may be an entire program, a piece of a program, or a single command (like the SQL commands such as INSERT or UPDATE), and it may engage in any number of operations on the database.
* A **transaction** is a set of logically related operations. For example, you are transferring money from your bank account to your friend’s account, the set of operations would be like this:

## Simple Transaction Example

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

This whole set of operations can be called a transaction.

**Transactions access data using two operations:**

* **read(x):** which transfers the data item X from the database to local buffer belonging to the transaction that executed the read operation
* **write(x):** which transfers the data item X from the local buffer of the transaction that executed the write back to the database

Ti: **read**(*A*)

*A* := *A –* 50

**write**(*A*)

**read**(*B*)

*B* := *B +* 50

**write**(*B)*

In the above example,Ti is transaction,we are transferring amount rs.50 from accountA to accountB

**Topic: ACID Properties in DBMS/Transaction Properties**

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

**1)Atomicity**  
By this, we mean that either the entire transaction takes place at once or doesn’t happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves following two operations.  
—**Abort**: If a transaction aborts, changes made to database are not visible.  
—**Commit**: If a transaction commits, changes made are visible.  
Atomicity is also known as the ‘All or nothing rule’.

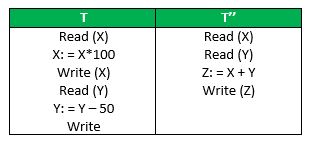
Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.



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

**2)Consistency**  
This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to correctness of a database. Referring to the example above,  
The total amount before and after the transaction must be maintained.  
Total **before T** occurs = **500 + 200 = 700**.  
Total **after T occurs** = **400 + 300 = 700**.  
Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2**fails. As a result T is incomplete.

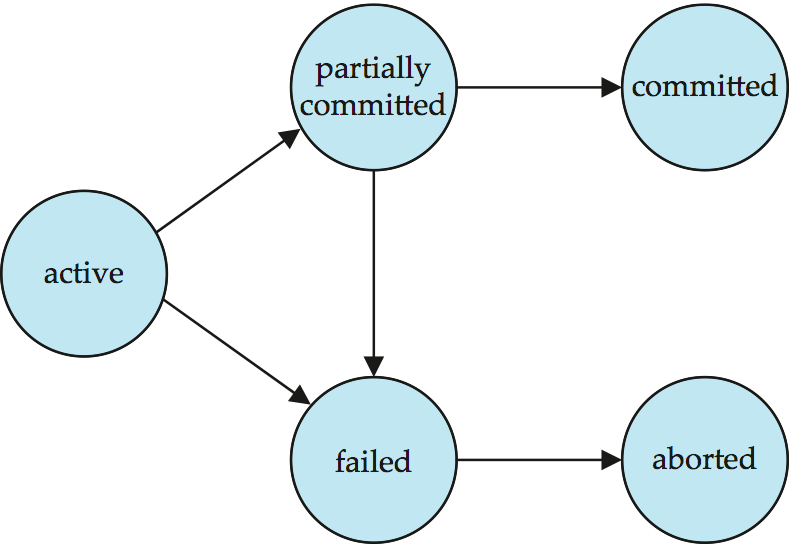
**3)Isolation**  
This property ensures that multiple transactions can occur concurrently without leading to inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular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



Suppose **T** has been executed till **Read (Y)** and then **T’’** starts. As a result , interleaving of operations takes place due to which **T’’** reads correct value of **X** but incorrect value of **Y**and sum computed by  
**T’’: (X+Y = 50, 000+500=50, 500)**  
is thus not consistent with the sum at end of transaction:  **T: (X+Y = 50, 000 + 450 = 50, 450)**.  
This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after a they have been made to the main memory.

**4)Durability:**  
This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if system failure occurs. These updates now become permanent and are stored in a non-volatile memory. The effects of the transaction, thus, are never lost.

## **Topic: Transaction States/Simple Transaction model**



## A transaction when executing , it must be in one of the following states:

## 1)Active State

## Aa transaction is a sequence of operations. If a transaction is in execution then it is said to be in active state. It doesn’t matter which step is in execution, until unless the transaction is executing, it remains in active state.

## 2)Partially Committed State

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

The reason why we have this state is because a transaction can fail during execution so if we are making the changes in the actual database instead of local memory, database may be left in an inconsistent state in case of any failure**. This state helps us to rollback the changes made to the database in case of a failure during execution.**

## 3)Committed State

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

## 4)Failed State

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

## 5)Aborted State

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

**Topic: Storage structure:**

* To understand how to ensure the atomicity and durability of a transaction , we must gain a better understanding of how the various data items in the database may be stored and accessed
* Depends on their speed, capacity and resilience to failure storage media are classified into the following:

**1. Volatile Storage** :

These are the primary memory devices in the system, and are placed along with the CPU.  These memories can store only small amount of data

Information does not survive system crashes-data in these memories will be lost on failure

* + Eg: Main Memory, Cache Memory
  + It is extremely fast (due to speed of the memory access & possible to access any data item directly

**2.Nonvolatile storage:**

These are secondary memories and are huge in size.

Information survives system crashes

* + **Eg:** Second storage devices like magnetic disks, tapes, flash storage etc
  + It is slower than volatile storage(due to random access)
  + Suspectible to failure may loss inforamtion

**3. Stable Storage:**

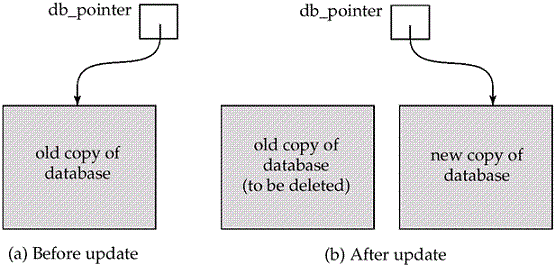
This is said to be third form of memory structure but it is same as non volatile memory. In this case, copies of same non volatile memories are stored at different places. This is because, in case of any crash and data loss, data can be recovered from other copies. This is even helpful if there one of non-volatile memory is lost due to fire or flood. It can be recovered from other network location. The failure can be because of system program, bug in a program, user, or system crash,transaction failure,disk failure

Information never lost

* + To implement this the information is replicated in several nonvolatile storage media
  + Updates must be done with care

Topic: **Transaction atomicity and durability/Implementation of atomicity and Durability**

* Recovery-management component of a database system can support atomicity and durability.
* Atomicity can be implemented by **shadow copy scheme.**
* It makes copies of the database, called shadow copies .
* It assumes that the database is simply a file on disk.
* A pointer called db-pointer is maintained on disk; it points to the current copy of the database
* In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database.
* All updates are done on the new database copy leaving the original copy untouched
* If the transaction
  + **Aborted**, it deletes the new copy, the old copy of the database has not been affected
  + **Successful,** it is committed as, new copy of the database written out to disk, db-pointer moves to the new copy and old copy is deleted



**Topic: Transaction Isolation:**

* Multiple transactions are allowed to run concurrently in the system.
* Advantages are:
  + **Increased throughput and resource utilization**
    - Eg. one transaction can be using the CPU while another is reading from or writing to the disk
  + **Reduced response time** for transactions: short transactions need not wait behind long ones.
* Concurrency may leads data base to enter into inconsistency state despite the correctness of individual transaction.
* To prevent from this, a variety of mechanisms called concurrency-control schemes are used

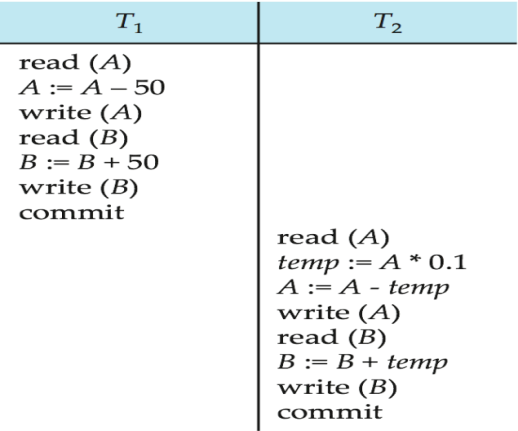
**Topic: Schedules:**

**Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed

* + A schedule for a set of transactions must consist of all instructions of those transactions
  + Must preserve the order in which the instructions appear in each individual transaction.
* A transaction that successfully completes its execution will have a **commit** instructions as the last statement
  + By default transaction assumed to execute commit instruction as its last step
* A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement

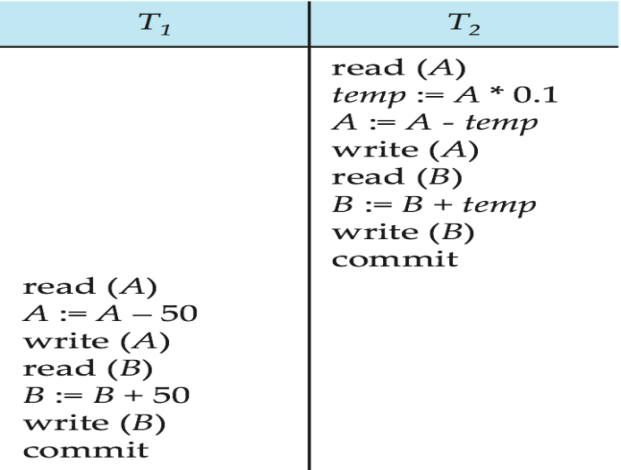
**Schedule1:**

* Let *T*1 transfer $50 from *A* to *B*, and *T*2 transfer 10% of the balance from *A* to *B.*
* An example of a **serial** schedule in which *T*1 is followed by *T*2 :



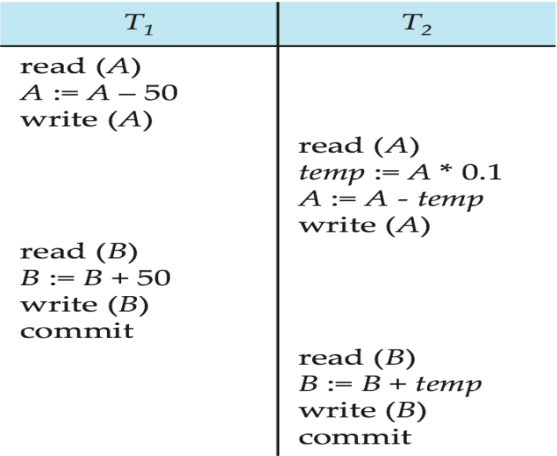
**Schedule 2:**

* A **serial** schedule in which *T*2 is followed by *T*1 :



**Schedule 3:**

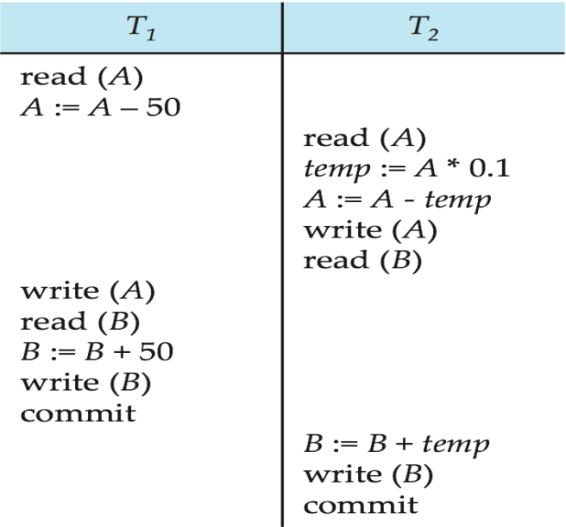
* Let *T*1 and *T*2 be the transactions defined previously*.* The following schedule is not a serial schedule, but it is **equivalent** to Schedule 1.



In schedules 1, 2 and 3, the sum “A + B” is preserved

**Schedule 4:**

The following concurrent schedule does not preserve the sum of “*A* + *B*”



**Topic: Serializability**

* **Basic Assumption** – Each transaction preserves database consistency.
* Thus, serial execution of a set of transactions preserves database consistency.
* A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. **conflictserializability**

2. **viewserializability**

**Conflicting Instructions:**

Two operations are said to be in conflict, if they satisfy all the following three conditions:

1. Both the operations should belong to different transactions.  
2. Both the operations are working on same data item.  
3. At least one of the operation is a write operation.

* Let *li* and *lj* be two Instructions of transactions *Ti* and *Tj* respectively. Instructions *li* and *lj***conflict** if and only if there exists some item *Q* accessed by both *li* and *lj*, and at least one of these instructions wrote *Q.*

1. *li* = **read**(*Q), lj =* **read**(*Q*). *li* and *lj*don’t conflict.  
2. *li* = **read**(*Q), lj =* **write**(*Q*). They conflict.  
3. *li* = **write**(*Q), lj =* **read**(*Q*). They conflict  
4. *li* = **write**(*Q), lj =* **write**(*Q*). They conflict

* Intuitively, a conflict between *li*and *lj* forces a (logical) temporal order between them.
  + If *li* and *lj* are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Example:

1) S1: T1 T2 S2: T1 T2

Read(A) after swapping write(A)

Write(A) read(A)

Here S1 is not equal to S2,so it is conflict

**Examples of conflict operations:**

1) T1 T2 2) T1 T2

Write(A) write(A)

Read(A) write(A)

3) T1 T2 4) T1 T2

Write(A) read(A)

Read(A) write(A)

**Examples of non-conflicting operations:**

1. T1 T2 2) T1 T2

Read(A) read(A)

Read(A) write(B)

1. T1 T2 4) T1 T2

Write(A) write(A)

Read(B) write(B)

**Conflict Serializability:**

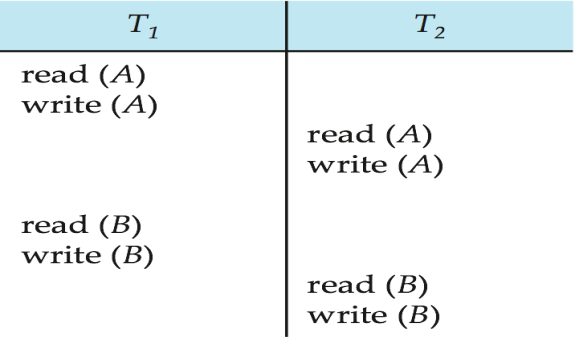
* If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent***.*

That is,A schedule is called conflict serializability if after swapping of non-conflicting operations, it can transform into a serial schedule.

* We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Example: Schedule 1 can be transformed into Schedule 2 which is a serial schedule where *T*2 follows *T*1, by a series of swaps of non-conflicting instructions. Therefore, Schedule 1 is conflict serializable.

**Schedule 1**



After swapping the non-conflicting instructions read(B) of T1 and write(A) of T2 we get:

**T1 T2**

**Read(A)**

**Write(A)**

**Read(A)**

**Read(B)**

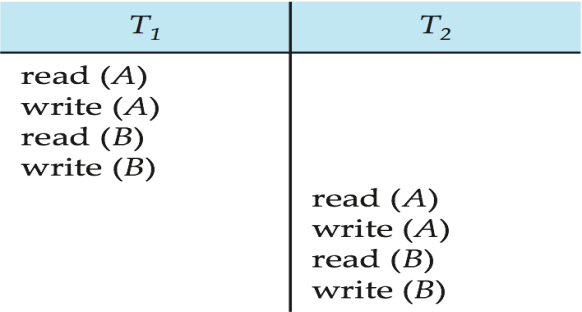
**Write(A)**

**Write(B)**

**Read(B)**

**Write(A)**

After swapping the non-conflicting instructions read(B) of T1 and read(A) of T2,write(B) of T1 and write(A) of T2 we get:**Schedule 2:**



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

**Example 2:**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)

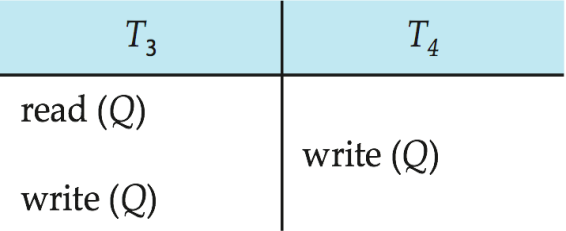
R(A)

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**.

**Example3**:The following schedule that is not conflict serializable:



* We are unable to swap instructions in the above schedule to obtain either the serial schedule <*T*3, *T*4>, or the serial schedule <*T*4, *T*3>.

**Precedence Graph For Testing Conflict Serializability:**

**Precedence Graph** or **Serialization Graph** is used commonly to test Conflict 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:

* 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.
* 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.

**Example:**

Consider a schedule S1:

T1 T2

R(x)

R(x)

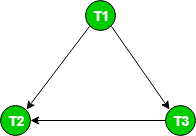
W(x) Precedence graph for S1:

W(x) In S1,R(x) of T2 and W(x) of T1 are conflict operations,W(x) of T1 and

W(x) of T2 are conflict operations.

Since the graph is cyclic, we can conclude that it is **not conflict serializable** to any schedule.

Example2:let us consider another example



Since the graph is acyclic, the schedule is conflict serializable. Performing Topological Sort on this graph would give us a possible serial schedule which is conflict equivalent to schedule S1.  
In Topological Sort, we first select the node with indegree 0, which is T1. This would be followed by T3 and T2.  
So, **S1 is conflict serializable** since it is **conflict equivalent** to the **serial schedule T1 T3 T2.**

* A schedule is conflict serializable if and only if its precedence graph is acyclic.
* **Precedence graph** *-* a direct graph where the vertices are the transactions (names).
* We draw an arc from *Ti*to *Tj*if the two transaction conflict, and *Ti*accessed the data item on which the conflict arose earlier.
* We may label the arc by the item that was accessed.

**View Serializabilty:**

* A schedule will view serializable if it is view equivalent to a serial schedule.
* If a schedule is conflict serializable, then it will be view serializable.
* The view serializable which does not conflict serializable contains blind writes.
* Every Conflict-serializable schedule is also view serializable, but there are view serializable schedules that are not conflict serializable.

**view equivalent :**The schedules S and *S1*are said to be view equivalent if they satisfy the following conditions :

1.Initial read: For each data item Q, if transaction T1 reads the initial value of Q in schedule S, then transaction T1should also read the initial value of Q in schedule S1.

2.Updated read:In schedule S, if T1 is reading Q which is updated(write(Q)) by T2 then in S2 also, T1 should read Q which is updated by T2.

3.Final write:For each data item Q, the transaction that performs the final write(Q) operation in schedule S,in schedule S1 also the same transaction must perform the final write(Q) operation. It means ,A final write must be the same between both the schedules.

**Example:**Let us consider the following schedule as S

T1 T2 T3

-----------------------------------------

Read(Q)

Write(Q)

Write(Q)

Write(Q)

The above schedule has 3 transactions, the total number of possible schedule we can form are 3!=6

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

**Taking first schedule S1:**

T1 T2 T3

-----------------------------------------

Read(Q)

Write(Q)

Write(Q)

Write(Q)

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

--updated read: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.

-- 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.

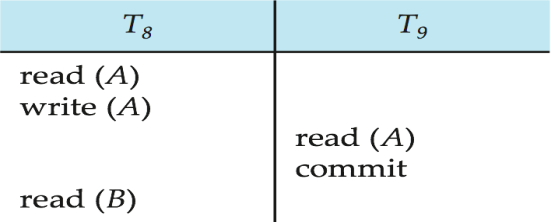
**Hence, view equivalent serial schedule is:**T1    →      T2    →    T3

--Transactions T2 and T3 performs write(Q) operations without having performed a read(Q) operation .Writes of this type are called **blind** writes. Blind writes appear in any view-serializable schedule that is not conflict serializable.

**Topic: Transaction Isolation and Atomicity:**

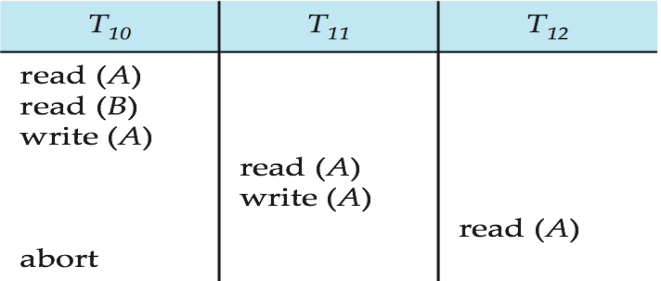
Transaction isolation and atomicity-means how database maintains atomicity during concurrent transaction(transaction isolation),atomicity will be successful if the transactions are recovered when a transaction failed due to some reason.

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.

* **Recoverable schedule** — Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules..If a transaction *Tj* reads a data item previously written by a transaction *Ti* , then the commit operation of *Ti***must** appear before the commit operation of *Tj.*
* The following schedule is not recoverable if *T9*commits immediately after the read(A) operation.  
   
* If *T*8 should abort, *T*9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

**Cascading Rollbacks:**

* **Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)**

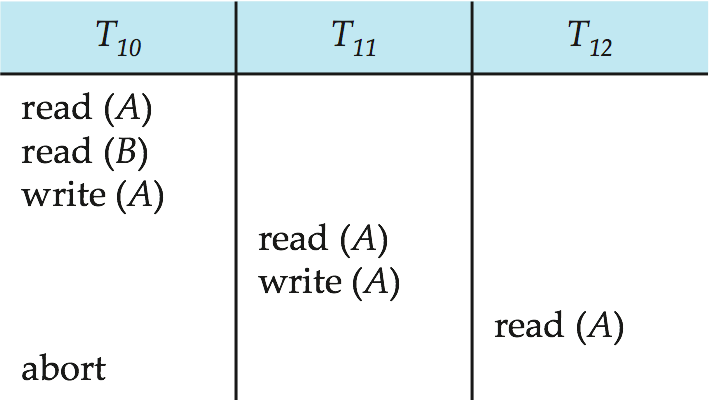
****If *T*10 fails, *T*11 and *T*12 must also be rolled back.

* Can lead to the undoing of a significant amount of work

**Cascadeless Schedules:**

* Cascadelessschedules — for each pair of transactions *Ti*and *Tj* such that *Tj* reads a data item previously written by *Ti*, the commit operation of *Ti* appears before the read operation of *Tj*.
* Every cascadeless schedule is also recoverable
* It is desirable to restrict the schedules to those that are cascadeless

Example of a schedule that is NOT cascadeless

****

**Topic: Transaction Isolation Levels:**

Isolation determines how transaction integrity is visible to other users and systems.

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 –

1. **Dirty Read –**A Dirty read is the situation when a transaction reads a data that has not yet been committed.
2. **Non Repeatable read –**Non Repeatable read occurs when a transaction reads same row twice, and get a different value each time.
3. **Phantom Read –**Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different.

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

**1)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.

**2)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.

**3)Repeatable read**—only committed records to be read, repeated reads of same record must return same value. 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.

**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.

* All the isolation levels disallow **dirty writes.**
* **Dirty Writes:** They disallow writes to a data item that has already been written by another transaction that has not yet committed or aborted
* By default isolation level in many systems is read committed.
* In SQL it is possible to set the isolation level by

“**set transaction isolation level serializable”;**