

Module III



TRANSACTION MANAGEMENT & CONCURRENCY CONTROL

Transactions



- Collections of operations that form a single logical unit of work are called **transactions**.
- A database system must ensure proper execution of transactions despite failures—either the entire transaction executes, or none of it does.
- **Transaction** is a unit of program execution that accesses and possibly updates various data items.

What is a Transaction?



- Any action that reads from and/or writes to a database may consist of
 - Simple SELECT statement to generate a list of table contents
 - A series of related UPDATE statements to change the values of attributes in various tables
 - A series of INSERT statements to add rows to one or more tables
 - A combination of SELECT, UPDATE, and INSERT statements

What is a Transaction?



- A *logical* unit of work that must be either entirely completed or aborted
- Successful transaction changes the database from one *consistent* state to another
 - One in which all data integrity constraints are satisfied
- Most real-world database transactions are formed by two or more database requests
 - The equivalent of a single SQL statement in an application program or transaction

Evaluating Transaction Results



- Not all transactions update the database
- SQL code represents a transaction because database was accessed
- Improper or incomplete transactions can have a devastating effect on database integrity
 - Some DBMSs provide means by which user can define enforceable constraints based on business rules
 - Other integrity rules are enforced automatically by the DBMS when table structures are properly defined, thereby letting the DBMS validate some transactions

Transaction Properties(ACID properties)



● Atomicity

- Requires that **all** operations (SQL requests) of a transaction be completed; if not, then the transaction is aborted
- A transaction is treated as a single, indivisible, logical unit of work
- This “**all-or-none**” property is referred to as atomicity.

Consistency

- Consistency property ensures that the database must remain in **the consistent state before the start of transaction and after the transaction is over.**
- Consistency states that only valid data will be written to the database.
- If for some reason a transaction is executed that violates the database consistency rules the entire transaction will be rolled back.



● **Isolation**

- Data used during execution of a transaction cannot be used by second transaction until first one is completed
- Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j , *it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished.*



- After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
- Durability can be implemented by writing all transaction into a **transaction log** that can be used to create a system state right before failure.
- A transaction can only be regarded as committed after it is written safely in the log.
- For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.
- These properties are called the **ACID properties**.

Transaction State

● A transaction must be in one of the following states:

- **Active:-**

- The initial state; the transaction stays in this state while it is executing.

- **Partially committed:-**

- After the final statement has been executed.

- **Failed:-**

- After the discovery that normal execution can no longer proceed.

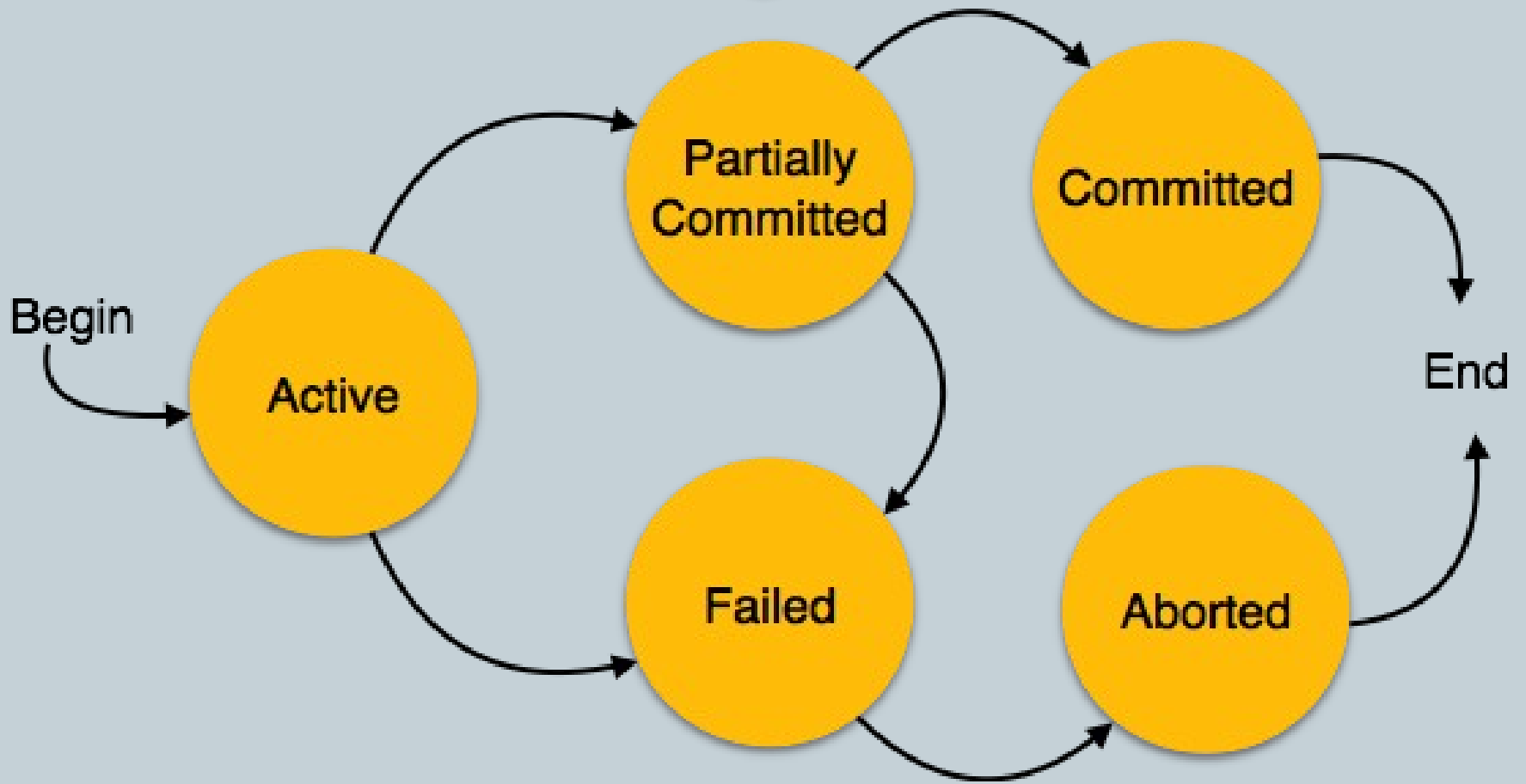
Transaction State

- **Aborted:-**

- After the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction

- **Committed:-**

- After successful completion



Transaction Management with SQL



- ANSI has defined standards that govern SQL database transactions
- Transaction support is provided by two SQL statements: COMMIT and ROLLBACK
- ANSI standards require that, when a transaction sequence is initiated by a user or an application program, it must continue through all succeeding SQL statements until one of four events occurs

Transaction Management with SQL



1. A COMMIT statement is reached- all changes are permanently recorded within the database
2. A ROLLBACK is reached – all changes are aborted and the database is restored to a previous consistent state
3. The end of the program is successfully reached – equivalent to a COMMIT
4. The program abnormally terminates and a rollback occurs

The Transaction Log



- Keeps track of all transactions that update the database. It contains:
 - A record for the beginning of transaction
 - For each transaction component (SQL statement)
 - Type of operation being performed (update, delete, insert)
 - Names of objects affected by the transaction (the name of the table)
 - “Before” and “after” values for updated fields
 - Pointers to previous and next transaction log entries for the same transaction
 - The ending (COMMIT) of the transaction
- Increases processing overhead but the ability to restore a corrupted database is worth the price

The Transaction Log



- Increases processing overhead but the ability to restore a corrupted database is worth the price
- If a system failure occurs, the DBMS will examine the log for all uncommitted or incomplete transactions and it will restore the database to a previous state
- The log is itself a database and to maintain its integrity many DBMSs will implement it on several different disks to reduce the risk of system failure

A Transaction Log



TABLE 9.1 A TRANSACTION LOG

TRL ID	TRX NUM	PREV PTR	NEXT PTR	OPERATION	TABLE	ROW ID	ATTRIBUTE	BEFORE VALUE	AFTER VALUE
341	101	Null	352	START	****Start Transaction				
352	101	341	363	UPDATE	PRODUCT	1558-QW1	PROD_QOH	25	23
363	101	352	365	UPDATE	CUSTOMER	10011	CUST_BALANCE	525.75	615.73
365	101	363	Null	COMMIT	**** End of Transaction				



TRL_ID = Transaction log record ID

PTR = Pointer to a transaction log record ID

TRX_NUM = Transaction number

(Note: The transaction number is automatically assigned by the DBMS.)

Transactions and schedules



- A transaction is seen by the dbms as a series or list of actions.
- Actions include read and writes of database object.
- Assume that an object O is always read into a program variable that is also named O
- Denote transaction T reading an object O as $R_T(O)$
- Similarly writing as $W_T(O)$



- Each transaction must specify as its final action either commit or abort
- Abort_T and Commit_T
- Schedule is a list of actions from a set of transactions,
- Schedule represents an actual or potential execution sequence.



- T1 T2
R(A)
W(A)
R(B)
W(B)
R(C)
W(C)



- A schedule that contains either abort or commit for each transactions is called **complete schedule**.
- If transactions are executed from start to finish, one by one---**serial schedule**

Concurrent execution of Transactions



- Transaction processing system usually allow multiple transaction to run concurrently.
- Allowing multiple transaction to run concurrently and allowing multiple transaction to update data concurrently causes several complications with consistency of data.
- Ensuring consistency with concurrency require an extra work.

Concurrent execution of Transactions

● Two reasons to allow concurrency are:-

- Improve throughput and resource utilization:-(Throughput – Number of transactions that can be executed in a given amount of time.)
- Reduced waiting time.



- There may be a mix of transactions running on a system, some short and some long.
- If transactions are run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction.
- But concurrent execution reduces the unpredictable delays in running transactions.

TYPES OF SCHEDULE



- **1. Serial Schedule**
- **2. Non-serial Schedule**
- **3. Serializable schedule**

1. Serial Schedule

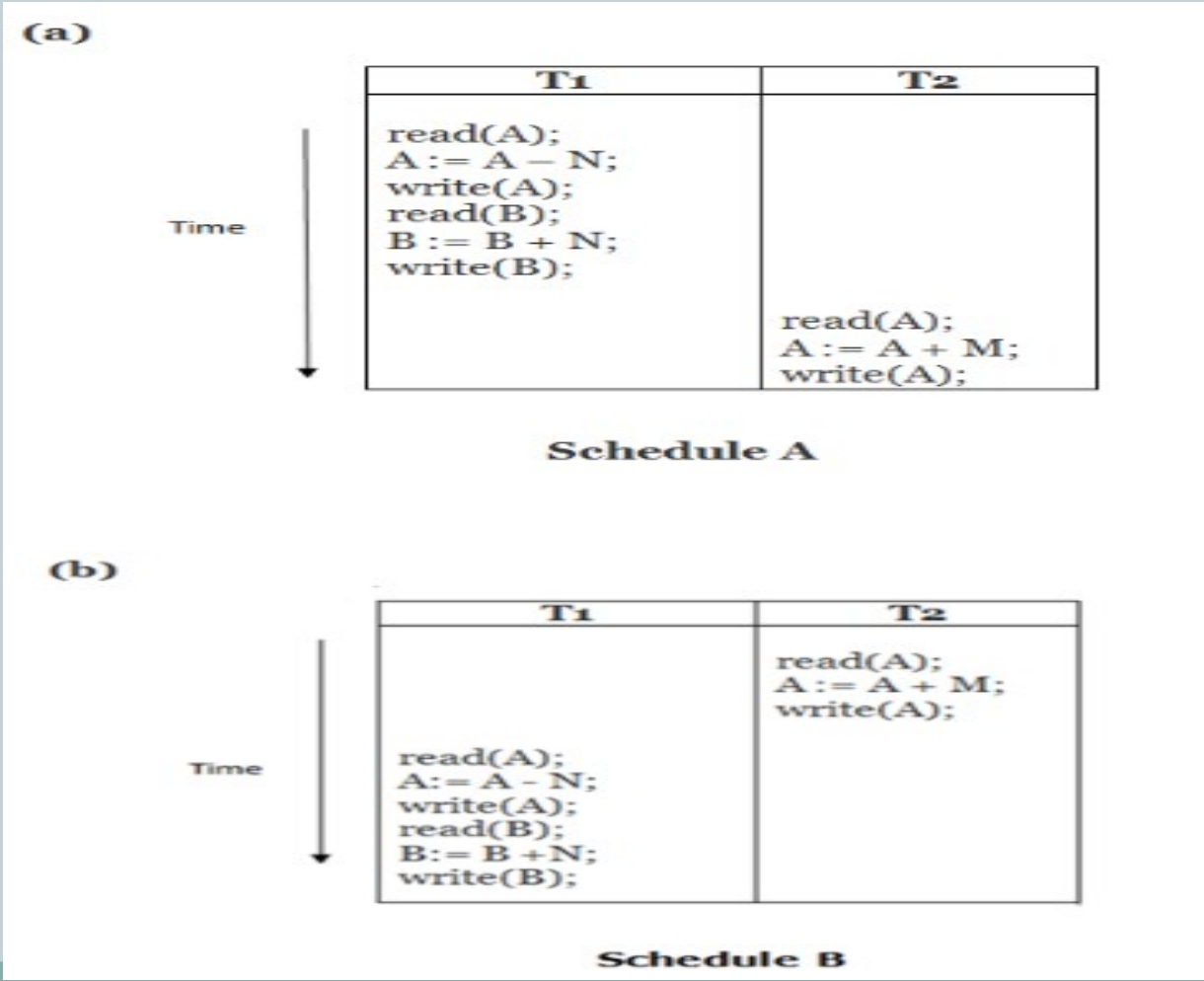


- The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction.
- In the serial schedule, when the first transaction completes its cycle, then the next transaction is executed.
- For example: Suppose there are two transactions T1 and T2 which have some operations.



- Execute all the operations of T1 which was followed by all the operations of T2.
- Execute all the operations of T2 which was followed by all the operations of T1.
- In the given (a) figure, Schedule A shows the serial schedule where T1 followed by T2.
- In the given (b) figure, Schedule B shows the serial schedule where T2 followed by T1.

- If it has no interleaving of operations, then there are the following two possible outcomes:

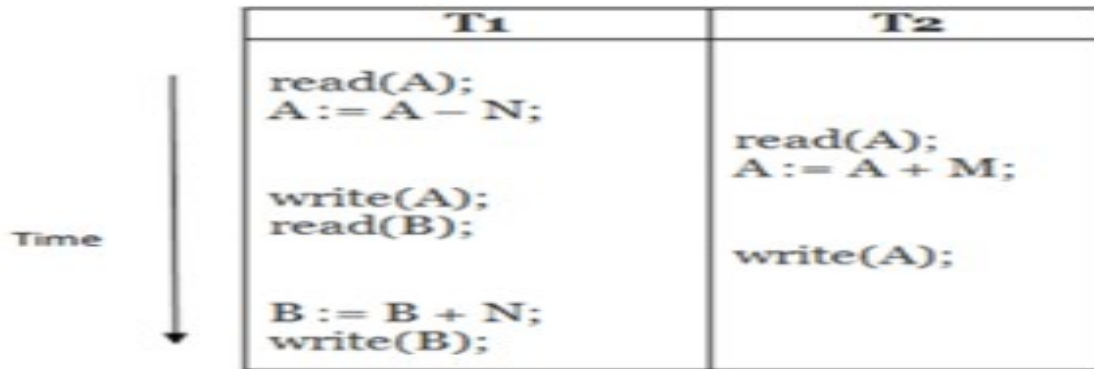


2. Non-serial Schedule/ Concurrent Execution

- If interleaving of operations is allowed, then there will be non-serial schedule.
- 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.

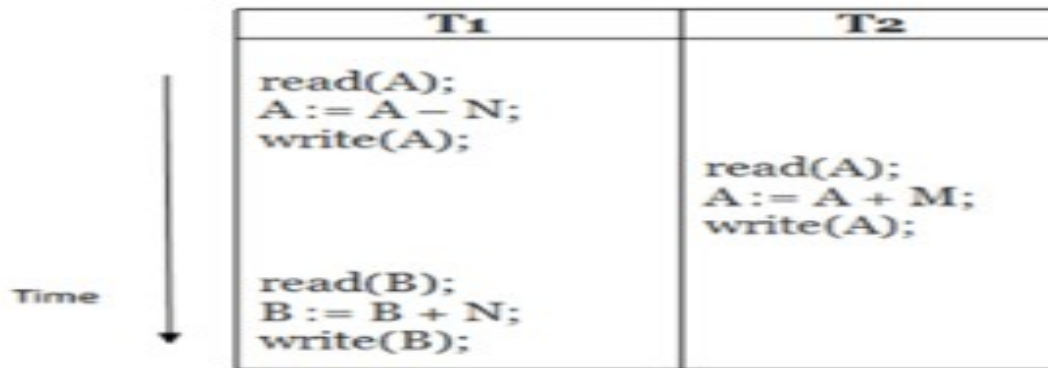
Non-serial Schedule

(c)



Schedule C

(d)



Schedule D

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.
- following problems occur with the Concurrent Execution of the operations:
- **Problem 1: Lost Update Problems (W - W Conflict)**
- **Dirty Read Problems (W-R Conflict)**
- **Unrepeatable Read Problem (W-R Conflict)/
*Inconsistent Retrievals Problem***

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.*
- **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	T_x	T_y
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

- At time t_1 , transaction T_x reads the value of account A, i.e., \$300 (only read).
- At time t_2 , transaction T_x deducts \$50 from account A that becomes \$250 (only deducted and not updated/write).
- Alternately, at time t_3 , transaction T_y reads the value of account A that will be \$300 only because T_x didn't update the value yet.
- At time t_4 , transaction T_y adds \$100 to account A that becomes \$400 (only added but not updated/write).
- At time t_6 , 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.
- Similarly, at time t_7 , transaction T_y writes the values of account A, so it will write as done at time t_4 that will be \$400. It means the value written by T_x is lost, i.e., \$250 is lost.

Dirty Read Problems (W-R Conflict) / Uncommitted Data



- 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.*

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 t_1 , transaction T_x reads the value of account A, i.e., \$300.
- At time t_2 , transaction T_x adds \$50 to account A that becomes \$350.
- At time t_3 , transaction T_x writes the updated value in account A, i.e., \$350.
- Then at time t_4 , transaction T_y reads account A that will be read as \$350.
- Then at time t_5 , transaction T_x rolls back 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) / *Inconsistent Retrievals Problem*



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

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

Serializability



- When multiple transactions run concurrently, then it may give rise 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.
- If a given schedule of 'n' transactions is found to be equivalent to some serial schedule of 'n' transactions, then it is called as a **serializable schedule**.

Difference between Serial Schedules and Serializable Schedule



- The only difference between serial schedules and serializable schedules is that-
- In serial schedules, only one transaction is allowed to execute at a time i.e. no concurrency is allowed.
- Whereas in serializable schedules, multiple transactions can execute simultaneously i.e. concurrency is allowed.

Types of Serializability

Types of Serializability

Conflict Serializability

View Serializability

Conflict Serializability



- A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.
- Let us consider a schedule S in which there are two consecutive instructions I_i and I_j of transactions T_i and T_j , respectively ($i \neq j$).



- If I_i and I_j refer to different data items, then we can swap I_i and I_j , without affecting the results of any instruction in the schedule.
- However, if I_i and I_j refer to the same data item Q , then the order of the two steps may matter.

□ There are four cases we need to consider

□ $I_i = \text{read}(Q)$, $I_j = \text{read}(Q)$, the order of I_i and I_j does not matter

□ $I_i = \text{read}(Q)$, $I_j = \text{Write}(Q)$,

↘ If I_i comes before I_j , then T_i does not read the value of Q that is written by T_j in instruction I_j . thus the order of I_i and I_j matters

□ $I_i = \text{Write}(Q)$, $I_j = \text{read}(Q)$, the order of I_i and I_j matter

□ $I_i = \text{write}(Q)$, $I_j = \text{write}(Q)$, the order of I_i and I_j does not matter, however the value obtained by the next $\text{read}(Q)$ instn is affected.



- 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.
- A Schedule S is conflict serializable if it is conflict equivalent



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

Precedence Graph



- **Precedence Graph** or **Serialization Graph** is used commonly to test Conflict Serializability of a schedule.
- It is a directed Graph (V, E) consisting of a set of nodes $V = \{T_1, T_2, T_3, \dots, T_n\}$ and a set of directed edges $E = \{e_1, e_2, e_3, \dots, e_m\}$.

- The graph contains **one node for each Transaction T_i** .
- An edge e_i is of the form $T_j \rightarrow T_k$.
- where T_j is the starting node of e_i and T_k is the ending node of e_i .
- An edge e_i is constructed between nodes T_j to T_k if one of the operations in T_j appears in the schedule before some conflicting operation in T_k .

ALGORITHM

- Create a node T in the graph for each participating transaction in the schedule.
- Check for conflicting instructions in the schedule:-
 - For the conflicting operation `read_item(X)` and `write_item(X)` (**RW Conflict**) – If a Transaction T_i executes a `read_item(X)` after T_j executes a `write_item(X)`, draw an edge from T_i to T_j in the graph.
 - For the conflicting operation `write_item(X)` and `read_item(X)` (**i.e WR conflict**) – If a Transaction T_i executes a `write_item(X)` after T_j executes a `read_item(X)`, draw an edge from T_i to T_j in the graph.

ALGORITHM

- **For the conflicting operation `write_item(X)` and `write_item(X)` (i.e WW conflict)–** If a Transaction T_j executes a `write_item (X)` after T_i executes a `write_item (X)`, draw an edge from T_i to T_j in the graph.
- **The Schedule S is serializable if there is no cycle in the precedence graph.**
- **If there is no cycle in the precedence graph, it means we can construct a serial schedule S' which is conflict equivalent to the schedule S .**

PROBLEM 1

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

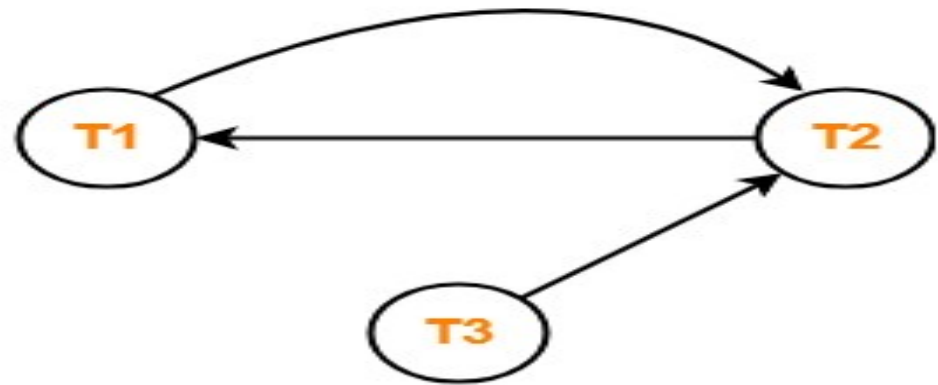
Step-01:

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.

Example: conflict serializable and conflict equivalent

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3 – showing only the read and write instructions.

CONFLICT EQUIVALENT

- Using precedence graph we found that the schedule 3 is conflict serializable since no cycles formed in graph.
- If a schedule S can be transformed into a schedule S' by a series of swapping of non conflicting instruction ,then we can say S and S' are **conflict equivalent**.
- Adjacent non conflicting pairs are swapped by position

Example :CONFLICT EQUIVALENT



- Consider schedule 3 which is conflict serializable.

T_1	T_2
read(A) write(A)	
	read(A) write(A)
read(B) write(B)	
	read(B) write(B)

Schedule 3 — showing only the read and write instructions.

Example : conflict equivalent(conti..)

- To find the conflict equivalent of the schedule 3 we need to perform certain swapping, i.e **swapping of positions of non conflicting adjacent instructions** in T1 and T2

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

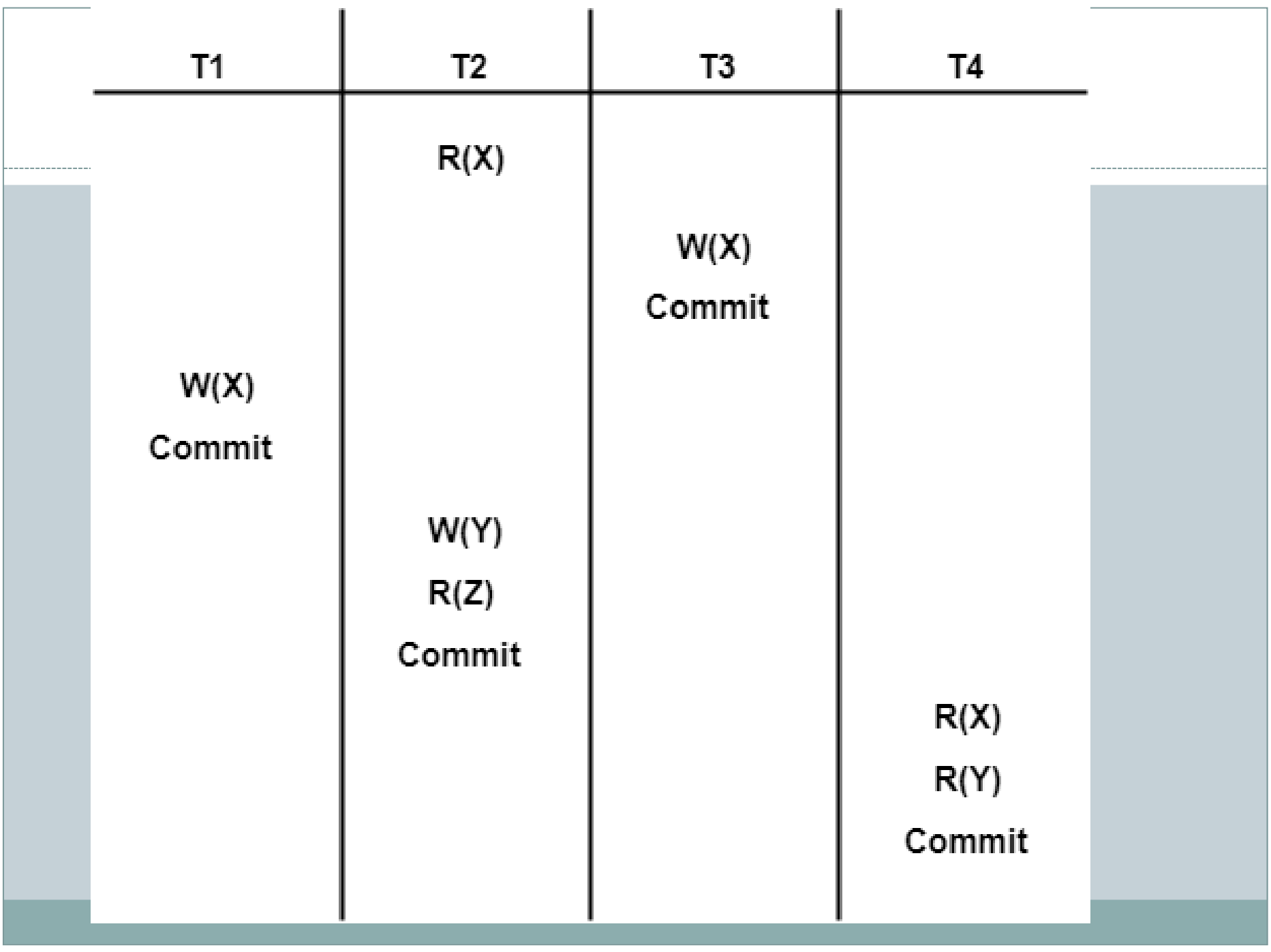
Schedule 5 — schedule 3 after swapping of a pair of instructions.

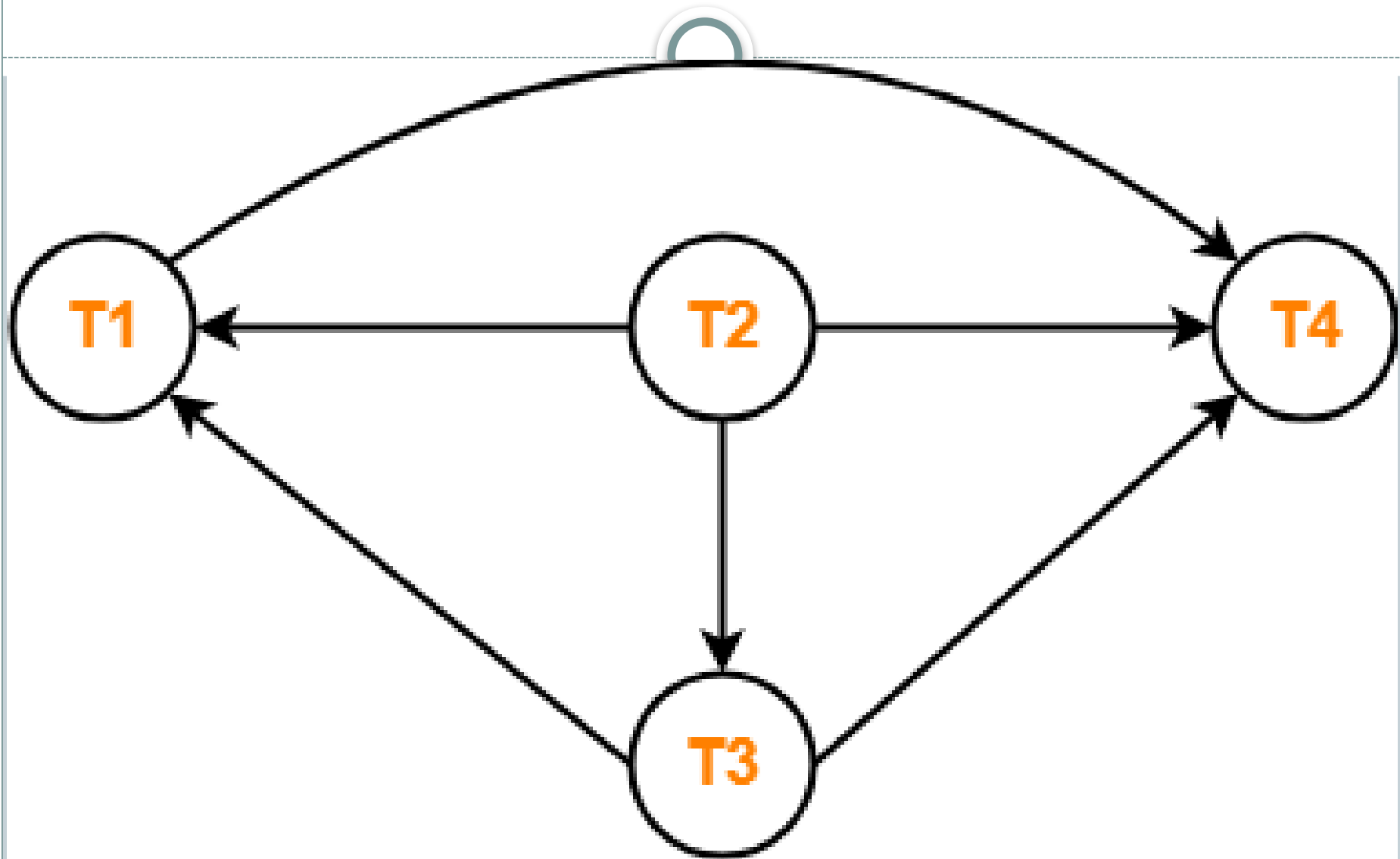
Example : conflict equivalent(conti..)

After a series of swapping we will get a serial schedule which is **conflict equivalent** of schedule 3.

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6 — a serial schedule that is equivalent to schedule 3.







- **Question: Consider the following schedules involving two transactions. Which one of the following statement is true?**
- S1: $R_1(X) R_1(Y) R_2(X) R_2(Y) W_2(Y) W_1(X)$
S2: $R_1(X) R_2(X) R_2(Y) W_2(Y) R_1(Y) W_1(X)$
- Both S1 and S2 are conflict serializable
- Only S1 is conflict serializable
- Only S2 is conflict serializable
- None



- Only S2 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.

- Two schedules S1 and S2 are said to be **view equivalent** if below conditions are satisfied .

- 1. Initial Read
- 2. Updated Read
- 3. Final Write

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
Read(A)	Write(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 T_i is reading A which is updated by T_j then in S2 also, T_i should read A which is updated by T_i .

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

Schedule S1

T1	T2	T3
Write(A)	Write(A)	
		<u>Read(A)</u>

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
Write(A)	Read(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.



- Condition 1 and 2 ensure that each transaction reads the same value in both schedules S1 and S2(so perform same computation).
- Condition 3 together with conditions 1 and 2 ensures both schedules result in same final state.



- Every conflict serializable schedule is also view serializable .
- But all view serializable are not conflict serializable.
- Blind writes appear in any view serializable schedule that is not 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**..
- 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-



- 1. For each data item Q , if transaction T_i reads the initial value of Q in schedule S , then transaction T_i must in schedule S also read the initial value of Q .
 - “Initial reads must be same for all data items”
- If transaction T_i reads a data item that has been updated by the transaction T_j in schedule S_1 , then in schedule S_2 also, transaction T_i must read the same data item that has been updated by transaction T_j .
 - “Write-read sequence must be same.”
-



- For each data item Q , the transaction that perform the final write(Q) operation in schedule S must perform the final Write(Q) operation in schedule in S ".
 - “Final writers must be same for all data items”.

How to check whether a given 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.
- 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 a value is known 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.

EXAMPLE :



- To check whether S is view serializable:-

Example:

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

Schedule S

With 3 transactions, the total number of possible schedule

ON

$$= 3! = 6$$

S1 = <T1 T2 T3>

S2 = <T1 T3 T2>

S3 = <T2 T3 T1>

S4 = <T2 T1 T3>

S5 = <T3 T1 T2>

S6 = <T3 T2 T1>

Taking first schedule S1:

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

EXAMPLE:- solution



- 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.

EXAMPLE:-SOLUTION (conti..)



- The first schedule S1 satisfies all three conditions, so we don't need to check another schedule.
- Hence, view equivalent serial schedule of S is S1:

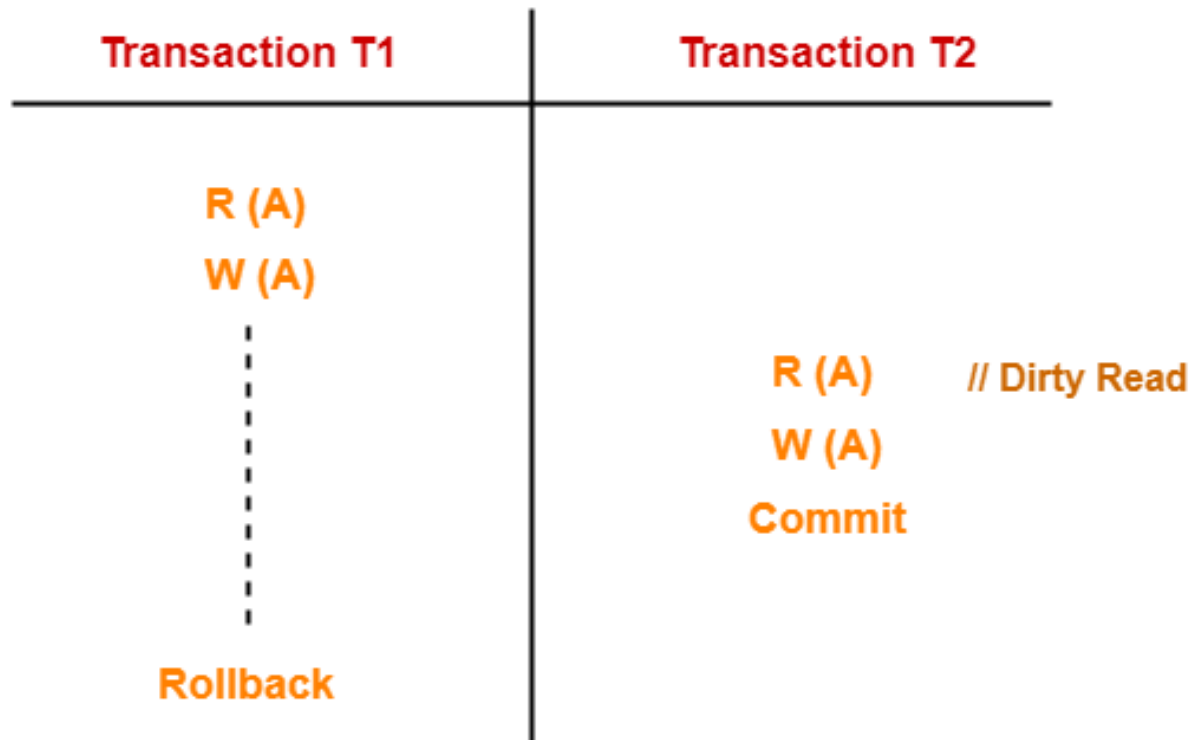
T1 → T2 → T3

Irrecoverable Schedules-

- If in a schedule,
 - A transaction performs a *dirty read operation* from an uncommitted transaction
 - And commits before the transaction from which it has read the value then such a schedule is known as an **Irrecoverable Schedule**.

Example: Irrecoverable schedule

Consider the following schedule-



Irrecoverable Schedule

Example: Irrecoverable schedule

- **In the above example**

- T2 performs a dirty read operation.
- T2 commits before T1.
- T1 fails later and roll backs.
- The value that T2 read now stands to be incorrect.
- T2 can not recover since it has already committed.
- So the above schedule is an irrecoverable schedule.

Recoverable Schedules-



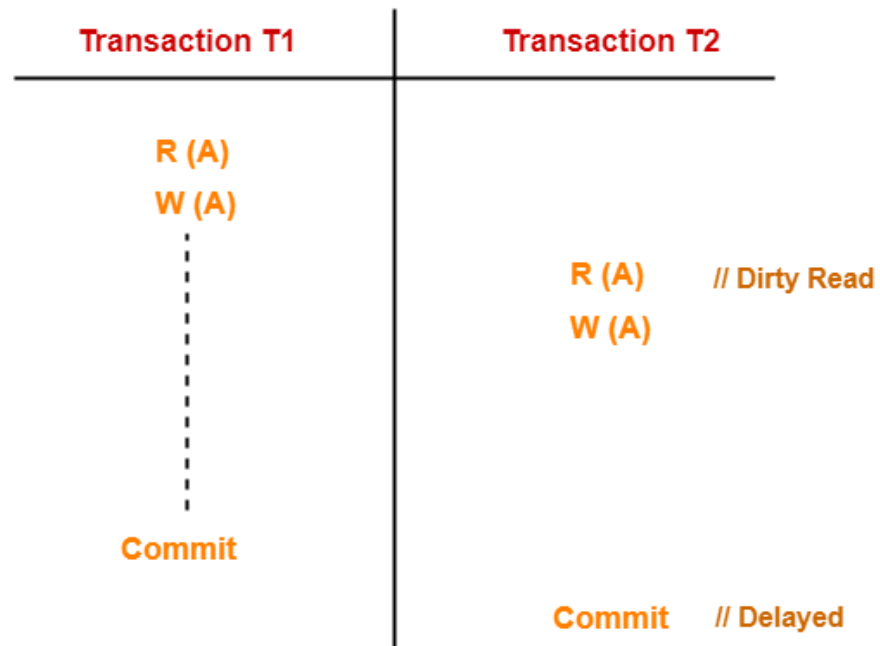
- If in a schedule,

- A transaction performs a *dirty read operation* from an uncommitted transaction.
- And its commit operation is delayed till the uncommitted transaction either commits or roll backs then such a schedule is known as a **Recoverable Schedule**.

EXAMPLE: Recoverable Schedules-



Consider the following schedule-



Recoverable Schedule



- In the above example T2 performs a dirty read operation.
- The commit operation of T2 is delayed till T1 commits or roll backs.
- T1 commits later.
- T2 is now allowed to commit.



- In case, T1 would have failed, T2 has a chance to recover by rolling back.
- Since the commit operation of the transaction that performs the dirty read is delayed.
- This ensures that it still has a chance to recover if the uncommitted transaction fails later.

Recoverable schedule



- **Two types:**
 - **Cascadeless schedule**
 - **Cascading schedule**

CASCADING SCHEDULE

- Even if a schedule is recoverable, to recover correctly from failure of transaction T_i , we may have to roll back several transactions.
- Such situations occur if transactions have read data written by T_i .

T_8	T_9	T_{10}
read(A) read(B) write(A)	read(A) write(A)	read(A)
abort		

Figure 14.15 Schedule 10.



- In the above example, transaction T8 has been aborted.
- T8 must be rolled back.
- Since T9 is dependent on T8, T9 must be rolled back. Since T10 is dependent on T9, T10 must be rolled back.
- The phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called **cascading rollback**.



• Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work.

Cascadeless schedule

- A cascadeless schedule is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i , the commit operation of T_i appears before the read operation of T_j .

- This type of schedule is called **cascadeless schedule**.

Cascadeless Schedule

T10	T11
read(A)	
write(A)	
	read(B)
commit	
	read(A)



CONCURRENCY CONTROL IN DATABASES

CONCURRENCY CONTROL

- When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved.
- To ensure it, the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called **concurrency control schemes**.
- Concurrency control can be performed by the dbms with various methods such as **locking methods**, **timestamp methods**, etc.

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.

Lock based protocol

- One way to achieve serializability is to access data items in a **mutually exclusive manner**.
- That is ,when a transaction is accessing a data item no other transaction is allowed to modify that data item.
- This can be achieved by holding a **lock** on the data item.

Locks

- Modes in which a data item may be locked.
 - Shared mode
 - Exclusive mode

Shared mode

- If a transaction T_i has obtained a shared -mode lock (denoted by S) on item Q , then T_i can read, but cannot write Q .
- Any other transaction can obtain the same lock, on same data item at same time.
- Denoted by **Lock-S(Q)**

Locks

Exclusive mode

- If a transaction T_i has obtained an exclusive-mode lock (denoted by X) on item Q , then T_i can both read and write Q .
- Any other transaction cannot obtain either exclusive/shared lock.
- Denoted by **lock $-X(Q)$**

Lock compatibility



	S	X
S	true	false
X	false	false

Figure 15.1 Lock-compatibility matrix comp.



- A transaction requests a shared lock on data item Q by executing the lock-S(Q) instruction.
- Similarly, a transaction requests an exclusive lock through the lock-X(Q) instruction.
- A transaction can unlock a data item Q by the unlock(Q) instruction.
- To access a data item, transaction T_i must first lock that item.




- If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released.

- Thus, T_i is made to wait until all incompatible locks held by other transactions have been released.

Granting of locks

- When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted.
- Some times a transaction may be starved.
- We can avoid starvation of transaction by granting locks in the following manner.



When a transaction T_i requests a lock on a data item Q in a particular mode M , the concurrency-control manager grants the lock provided that,

- 1. There is no other transaction holding a lock on Q in a mode that conflict with M .
- 2. There is no other transaction that is waiting for a lock on Q and that made its lock request before T_i .



```
T1:lock-X(B);  
  read (B);  
  B:=B-50;  
  write(B);  
  unlock(B);  
  lock-X(A);  
  read(A);  
  A:=A+50;  
  write(A);  
  unlock(A);
```

```
T2:lock-S(A)  
  read(A)  
  unlock(A)  
  lock-S(B)  
  read(B)  
  unlock(B)  
  display(A+B)
```

T1	T2	Concurrency-cntrl manager
Lock-X(B)		Grant-X(B,T1)
Read(B) B:=B-50 Wite(B) Unlock(B)		
	Lock-S(A)	Grant-S(A,T2)
	Read(A) Unlock(A) Lock-S(B)	
		Grant-S(B,T2)
	Read B,unlock(B),display(A+B)	
Lock-X(A)		Grant-X(A,T1)
Read(A) A:=A+50		

Two - Phase Locking (2PL)



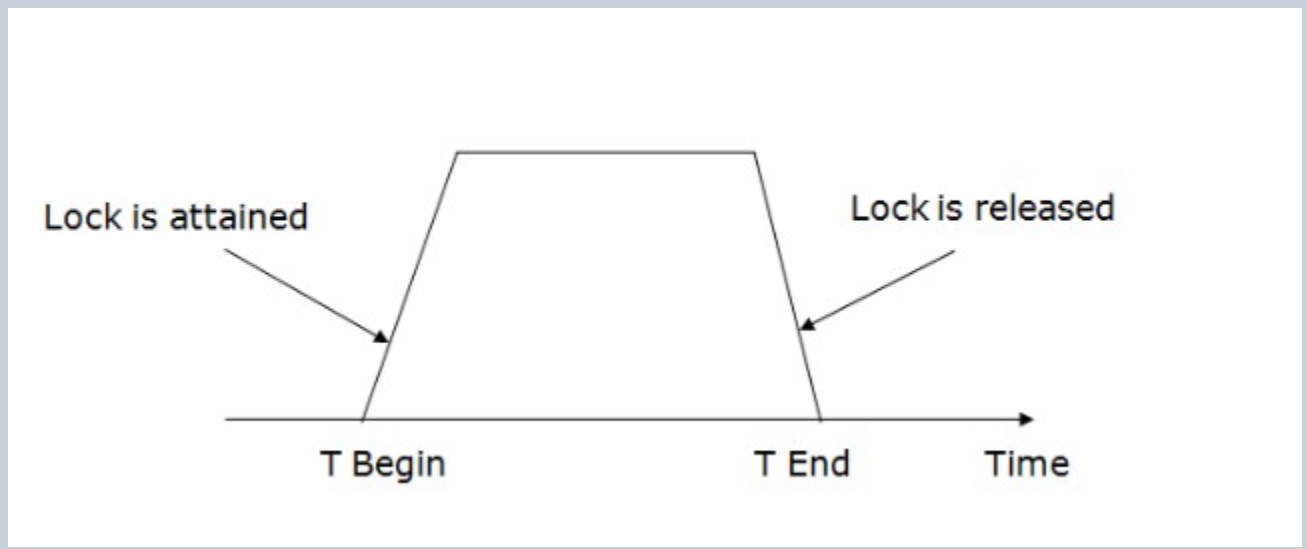
- Two-Phase locking protocol which is also known as a 2PL protocol.
- This protocol ensures conflict serializability.
- This protocol consists of 2 phases:
 - **Growing phase**
 - **Shrinking phase**

phases

- **Growing phase-** A transaction may obtain locks but may not release any locks.
- The point in schedule transaction had obtained its final lock is called **locking point**(end of growing phase.
- **Shrinking phase-**A transaction may release locks but may not get any new locks in this phase.



- Initially transaction is in the growing phase. The transaction acquires locks as needed.
- Once the transaction releases a lock it enters the shrinking phase and cannot issue any lock requests.



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	—	—

Transaction T_1 :

- The growing Phase is from steps 1-3.
- The shrinking Phase is from steps 5-7.
- Lock Point at 3

Transaction T_2 :

- The growing Phase is from steps 2-6.
- The shrinking Phase is from steps 8-9.
- Lock Point at 6



- Two phase locking does not ensure freedom from **deadlock**.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Figure 15.7 Schedule 2.



- Cascading rollback may occur under two-phase locking.
- Cascading rollbacks can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol**.

Deadlock in 2PL

- T3 is holding an exclusive mode lock on B and T4 is requesting a shared-mode lock on B, T4 is waiting for T3 to unlock B.
- Similarly, since T4 is holding a shared-mode lock on A and T3 is requesting an exclusive-mode lock on A, T3 is waiting for T4 to unlock A.
- We have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called **deadlock**.
- When deadlock occurs, the system must roll back one of the two transactions.



- Deadlock are necessary evil associated with locking, if we want to prevent inconsistent states.
- The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time

Concurrency Control Based on Timestamp Ordering

- Another method for determining the serializability order is to select an ordering among transactions.
- The most common method for doing so is to use a **timestamp-ordering scheme**.

TIMESTAMPS

- With each transaction T_i in the system, we associate a **unique fixed timestamp**, denoted by $TS(T_i)$.
- This timestamp is assigned by the database system before the transaction T_i starts execution.
- If a transaction T_i has been assigned timestamp $TS(T_i)$, and a new transaction T_j enters the system, then $TS(T_i) < TS(T_j)$.



● There are two simple methods for implementing this scheme:

- **Use the value of the system clock as the timestamp.** A transaction's timestamp is equal to the value of the clock when the transaction enters the system.
- **Use a logical counter** that is incremented after a new timestamp has been assigned. A transaction's timestamp is equal to the value of the counter when the transaction enters the system.

● To implement this scheme, we associate with each data item Q two timestamp values:

- **W-timestamp(Q)** denotes the largest timestamp of any transaction that executed $\text{write}(Q)$ successfully.
- **R-timestamp(Q)** denotes the largest timestamp of any transaction that executed $\text{read}(Q)$ successfully.

● These timestamps are updated whenever a new $\text{read}(Q)$ or $\text{write}(Q)$ instruction is executed.

The Timestamp-Ordering Protocol

- The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.
- This protocol operates as follows:
 1. Suppose that transaction T_i issues $\text{read}(Q)$.
 - a. If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, then T_i needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and T_i is rolled back.
 - b. If $\text{TS}(T_i) \geq \text{W-timestamp}(Q)$, then $\text{read}(Q)$ is executed and $\text{R-timestamp}(Q)$ is set to the maximum of $\text{R-timestamp}(Q)$ and $\text{TS}(T_i)$.

Timestamp-Ordering Protocol(conti..)

2. Suppose that transaction T_i issues $\text{write}(Q)$.

- a. If $\text{TS}(T_i) < \text{R-timestamp}(Q)$ then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls T_i back.
- b. If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, then T_i is attempting to write an obsolete value of Q . Hence, the system rejects this write operation and rolls T_i back.
- c. Otherwise, the system executes the write operation and sets $\text{W-timestamp}(Q)$ to $\text{TS}(T_i)$.

Timestamp-Ordering Protocol(conti..)

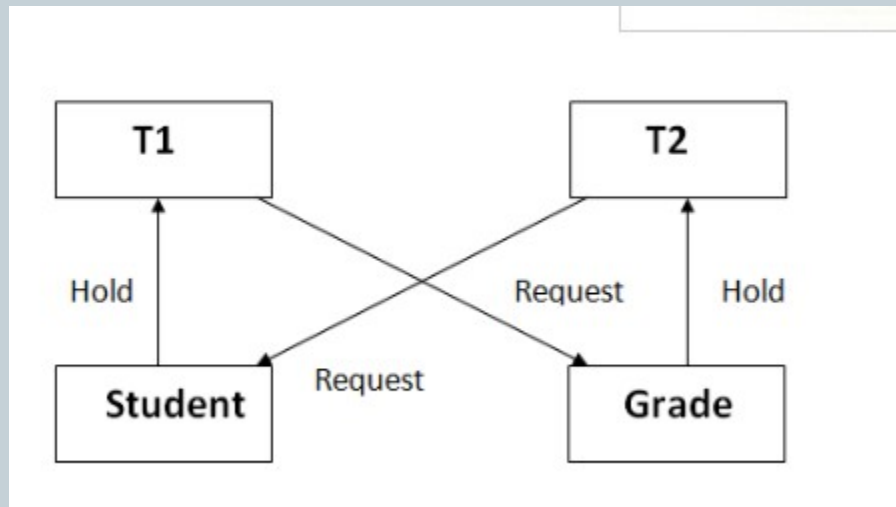
- If a transaction T_i is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.



- The timestamp-ordering protocol ensures **conflict serializability**.
- The protocol ensures **freedom from deadlock**, since no transaction ever waits.

Deadlock

- In a database, a deadlock is an unwanted situation in which two or more transactions are waiting indefinitely for one another to give up locks.



Deadlock Avoidance



- 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.

Deadlock Detection



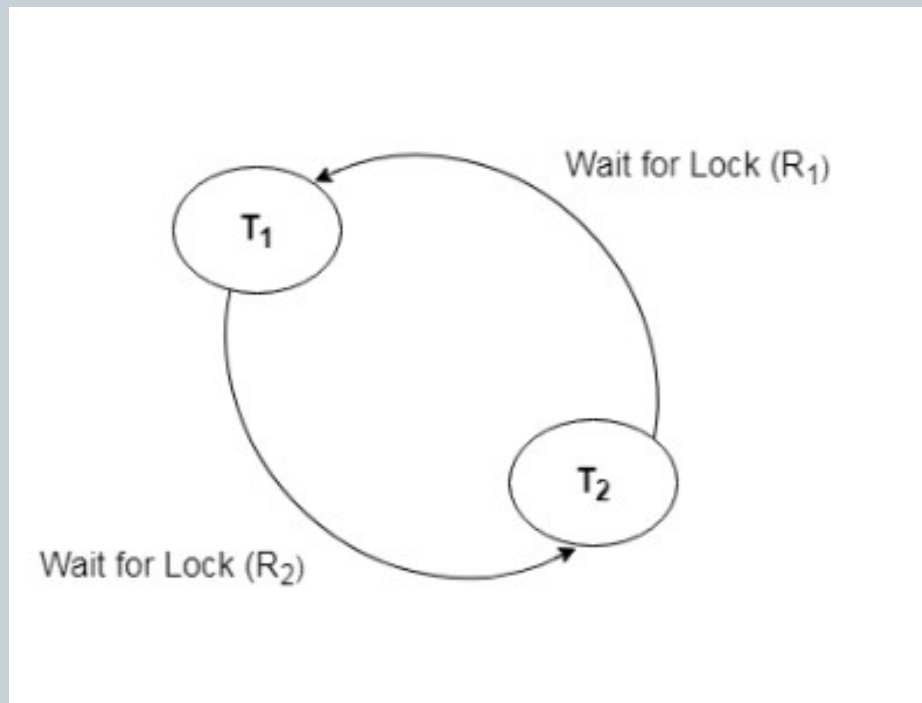
- When a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not.
- The lock manager maintains a **Wait for the graph** to detect the deadlock cycle in the database.

Wait for Graph



- 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.**
- 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.

Wait for Graph



Deadlock Prevention

- 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.
- Two schemes for deadlock prevention:
 - Wait-Die scheme
 - Wound wait scheme

Wait-Die scheme



- In this scheme, If a transaction requests a resource that is locked by another transaction, then the DBMS simply checks the timestamp of both transactions and allows the older transaction to wait until the resource is available for execution.
- There are two transactions T_i and T_j and let $TS(T)$ is a timestamp of any transaction T .
- If T_2 holds a lock by some other transaction and T_1 is requesting for resources held by T_2 then the following actions are performed by DBMS:

Wait-Die scheme



- Check if $TS(T_i) < TS(T_j)$ - If T_i is the older transaction and T_j has held some resource, then it allows T_i to wait until resource is available for execution. That means if a younger transaction has locked some resource and an older transaction is waiting for it, then an older transaction is allowed to wait for it till it is available
- If T_i is an older transaction and has held some resource with it and if T_j is waiting for it, then T_j is killed and restarted later with random delay but with the same timestamp. i.e. if the older transaction has held some resource and the younger transaction waits for the resource, then the younger transaction is killed and restarted with a very minute delay with the same timestamp.
This scheme allows the older transaction to wait but kills the younger one.

Wound wait scheme



- In 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.

- 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.

Optimistic Methods for Concurrency Control



- The optimistic method of concurrency control is based on the assumption that conflicts of database operations are rare and that it is better to let transactions run to completion and only check for conflicts before they commit.
- An optimistic concurrency control method is also known as validation or certification methods.
- No checking is done while the transaction is executing. The optimistic method does not require locking or timestamping techniques. Instead, a transaction is executed without restrictions until it is committed.



- In optimistic methods, each transaction moves through the following phases:
- Read phase.
- Validation or certification phase.
- Write phase.

- **(i)** During read phase, the transaction reads the database, executes the needed computations and makes the updates to a private copy of the the database values. All update operations of the transactions are recorded in a temporary update file, which is not accessed by the remaining transactions.
- **(ii)** During the validation phase, the transaction is validated to ensure that the changes made will not affect the integrity and consistency of the database. If the validation test is positive, the transaction goes to a write phase. If the validation test is negative, he transaction is restarted and the changes are discarded.
- **(iii)** During the write phase, the changes are permanently applied to the database.

Database Recovery

Management-Transaction Recovery



- When a database fails it must possess the facilities for fast recovery.
- 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.



- Recovery techniques are heavily dependent upon the existence of a special file known as a **system log**.
- It contains information about the start and end of each transaction and any updates which occur in the **transaction**.
- The log keeps track of all transaction operations that affect the values of database items.
- This information is needed to recover from transaction failure



- The log is kept on disk `start_transaction(T)`: This log entry records that transaction T starts the execution.
- `read_item(T, X)`: This log entry records that transaction T reads the value of database item X.
- `write_item(T, X, old_value, new_value)`: This log entry records that transaction T changes the value of the database item X from `old_value` to `new_value`. The old value is sometimes known as a before image of X, and the new value is known as an afterimage of X.



- **commit(T):** This log entry records that transaction T has completed all accesses to the database successfully and its effect can be committed (recorded permanently) to the database.
- **abort(T):** This records that transaction T has been aborted
- .**checkpoint:** Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

recovery process



- **Undoing** – If a transaction crashes, then the recovery manager may undo transactions i.e. reverse the operations of a transaction. This involves examining a transaction for the log entry `write_item(T, x, old_value, new_value)` and setting the value of item `x` in the database to `old_value`. There are two major techniques for recovery from non-catastrophic transaction failures: deferred updates and immediate updates.



- **Deferred update** – This technique does not physically update the database on disk until a transaction has reached its commit point. Before reaching commit, all transaction updates are recorded in the local transaction workspace. If a transaction fails before reaching its commit point, it will not have changed the database in any way so UNDO is not needed. It may be necessary to REDO the effect of the operations that are recorded in the local transaction workspace, because their effect may not yet have been written in the database. Hence, a deferred update is also known as the **No-undo/redo algorithm**



- **Immediate update** – In the immediate update, the database may be updated by some operations of a transaction before the transaction reaches its commit point. However, these operations are recorded in a log on disk before they are applied to the database, making recovery still possible. If a transaction fails to reach its commit point, the effect of its operation must be undone i.e. the transaction must be rolled back hence we require both undo and redo. This technique is known as **undo/redo algorithm**.



- **Caching/Buffering** – In this one or more disk pages that include data items to be updated are cached into main memory buffers and then updated in memory before being written back to disk. A collection of in-memory buffers called the DBMS cache is kept under control of DBMS for holding these buffers. A directory is used to keep track of which database items are in the buffer. A dirty bit is associated with each buffer, which is 0 if the buffer is not modified else 1 if modified.



- **Shadow paging** – It provides atomicity and durability. A directory with n entries is constructed, where the i th entry points to the i th database page on the link. When a transaction began executing the current directory is copied into a shadow directory. When a page is to be modified, a shadow page is allocated in which changes are made and when it is ready to become durable, all pages that refer to original are updated to refer new replacement page.