Transaction Processing

Module No. 5	Transaction Processing, Concurrency Control, and Recovery

Transaction Processing: Transaction and System concepts, Desirable properties of Transactions, Characterizing Schedules Based on Recoverability and Serializability, Concurrency Control: Two-Phase Locking, Timestamp Ordering, Database Recovery: Recovery Concepts, Immediate Update, Deferred Update, Shadow Paging.

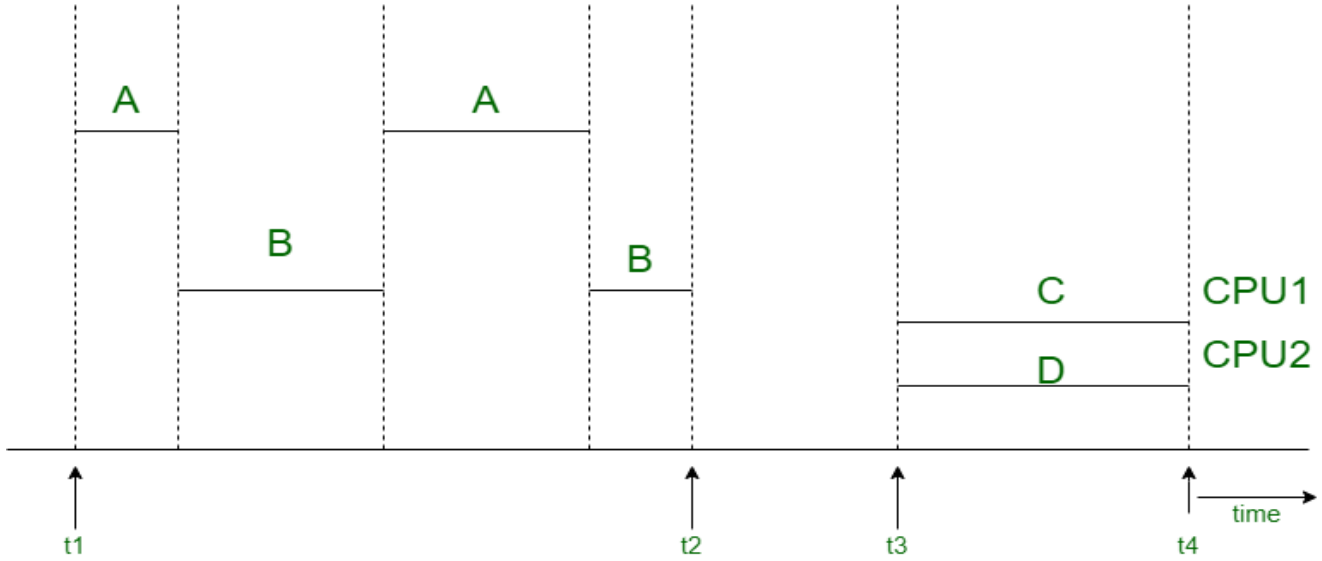
Single-User versus Multiuser Systems

- Single-user and multi-user database systems refer to the number of users who can access a database system simultaneously.

Single User Database Systems	Multi-User Database Systems
• A DBMS is a single-user if at most one user at a time can use the system.	• DBMS is a multi-user if many/multi-users can use the system and hence access the database concurrently.
• Single-User DBMSs are mostly restricted to personal computer systems.	• Most DBMSs are multi-user, like databases of airline reservation systems, banking databases, etc.
• Can only be accessed by the user who installed it or the user who is currently logged in.	• Can be accessed by users who are logged in to the network.

Modes of Concurrency

- A single central processing unit (CPU) can only execute at most one process at a time.
- However, multi-programming operating systems execute some commands from one process, then suspend that process and execute some commands from the next process, and so on.
- A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, the concurrent execution of processes is actually interleaved.

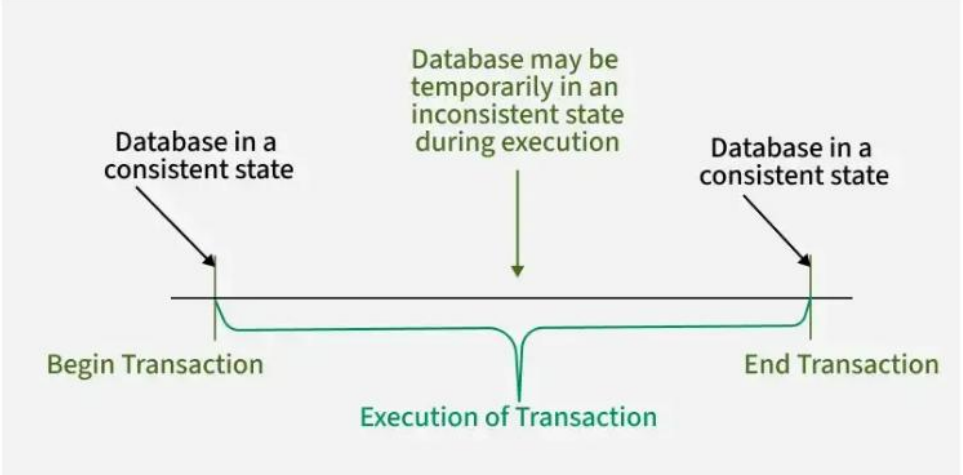


Interleaved processing vs Parallel processing of concurrent transactions.

- Interleaved processing: concurrent execution of processes is interleaved on a single CPU.
- Parallel processing: processes are concurrently executed on multiple CPUs.

Transactions

- A transaction is a unit of program execution that accesses and possibly updates various data items.
- The transaction consists of all operations executed between the statements begin and end of the transaction.
- Transaction boundaries: Begin and End transaction.
- Read-only transaction and Read-write transaction



- A transaction must see a consistent database
- During transaction execution the database may be inconsistent
- When the transaction is committed, the database must be consistent

Two main issues to deal with:

- Failures, e.g. hardware failures and system crashes
- Concurrency, for simultaneous execution of multiple transactions

Transactions

A's Account

```
Open_Account(A)
Old_Balance = A.balance
New_Balance = Old_Balance - 500
A.balance = New_Balance
Close_Account(A)
```

B's Account

```
Open_Account(B)
Old_Balance = B.balance
New_Balance = Old_Balance + 500
B.balance = New_Balance
Close_Account(B)
```

Transactions

Simple Model of A Database (for purposes of discussing transactions)

- A database – a collection of named data items.
- The size of a data item is called its granularity.
- Granularity of data - a field, a record, or a whole disk block (Concepts are independent of granularity).

Transaction operations:

- read (X): *Performs the reading operation of data item X from the database*
- write (X): *Performs the writing operation of data item X to the database*

Transactions

The basic unit of data transfer from disk to main memory is one disk page (disk block). Executing a read_item(X) command includes the following steps:

Executing a write_item(X) command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated disk block from the buffer back to disk (either immediately or at some later point in time).

Transactions

Read and Write Operations Cont'd

Two sample transactions. (a) Transaction T1. (b) Transaction T2.

(a) T_1

```
read_item (X);
X:=X-N;
write_item (X);
read_item (Y);
Y:=Y+N;
write_item (Y);
```

(b) T_2

```
read_item (X);
X:=X+M;
write_item (X);
```

Read set of a transaction

- Set of all items read

Write set of a transaction

- Set of all items written

Transactions

Concurrency Control

Transactions submitted by various users may execute concurrently

- Access and update the same database items.
- Some form of concurrency control is needed.

The lost update problem

- Occurs when two transactions that access the same database items have operations interleaved.
- Results in incorrect value of some database items.

Transactions

Why Concurrency Control is needed

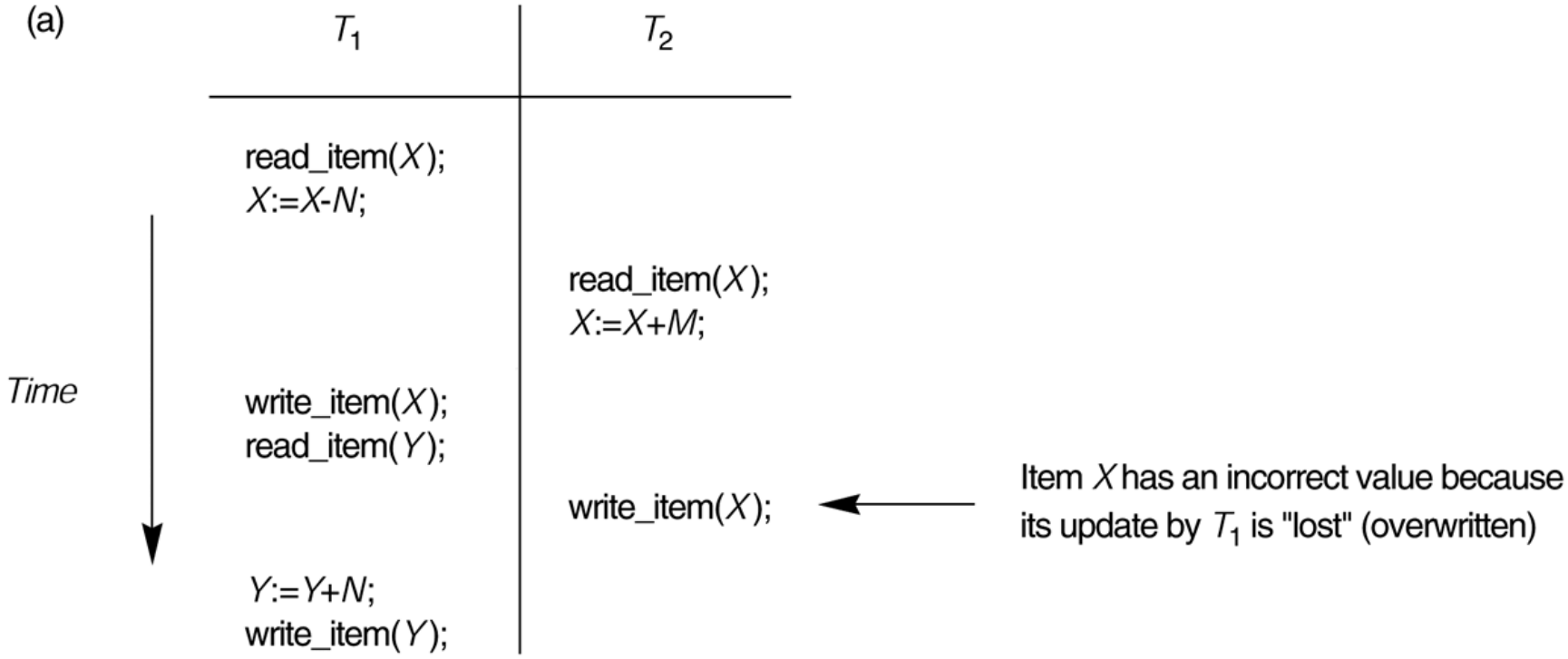
- The lost update problem.
- The temporary update problem.
- The incorrect summary problem.
- The Unrepeatable Read Problem

Transactions

Why Concurrency Control is needed

1. The Lost Update Problem

- This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.



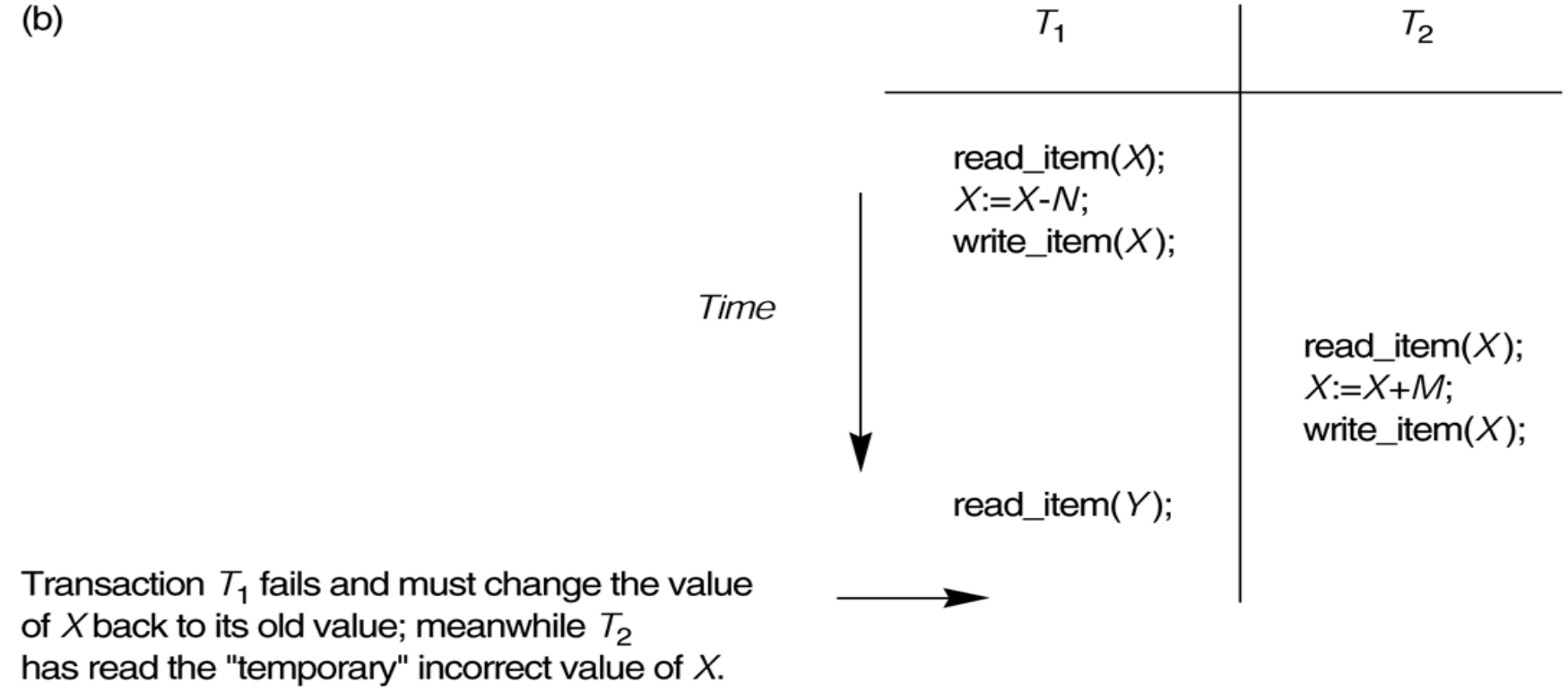
Transactions

Why Concurrency Control is needed

2. The Temporary Update (or Dirty Read) Problem

- This occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value.

(b)



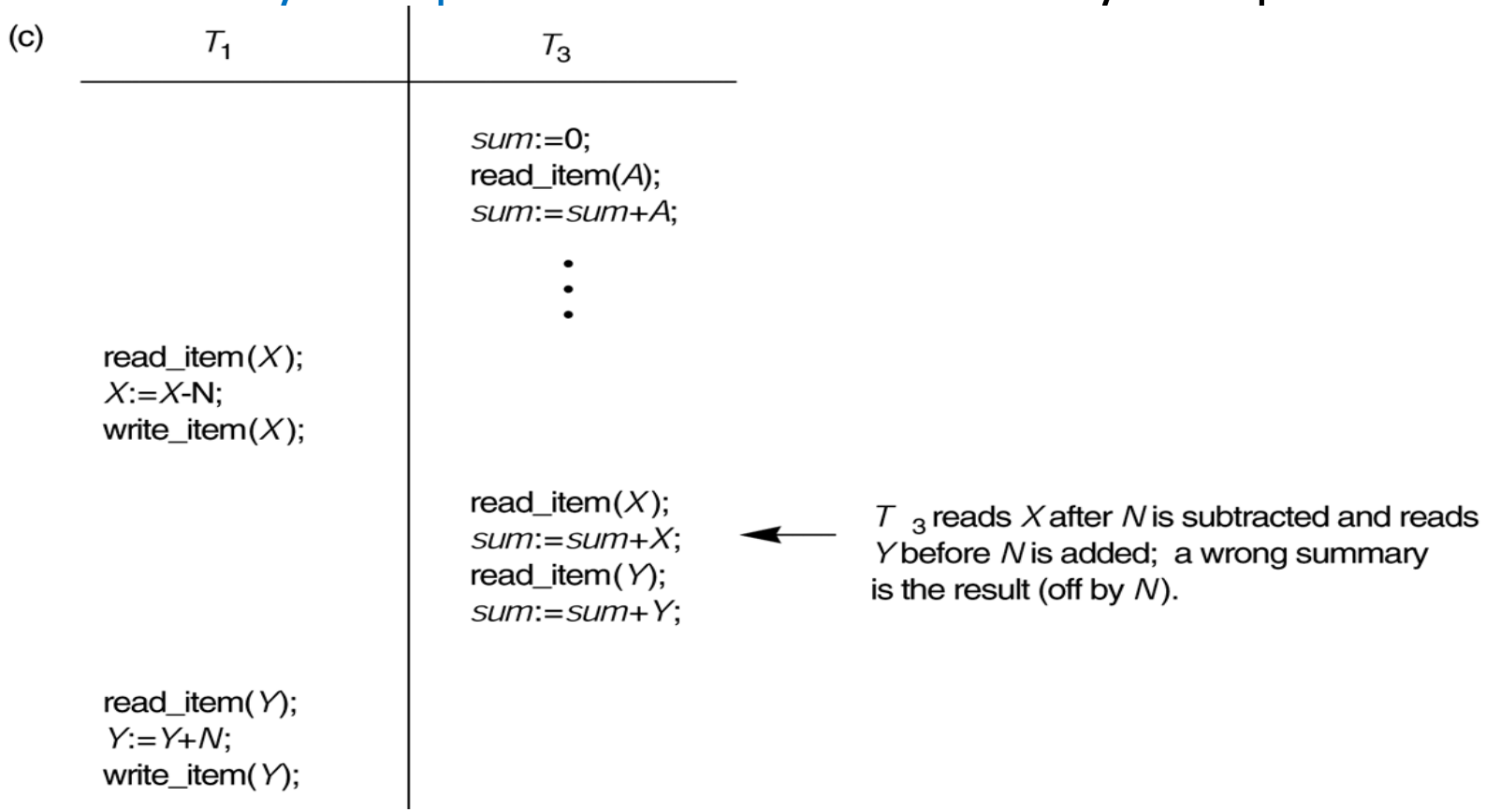
Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the "temporary" incorrect value of X.

Transactions

Why Concurrency Control is needed

3. The Incorrect Summary Problem

- If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.



Transactions

Why Concurrency Control is needed

4. The Unrepeatable Read Problem

- Transaction T reads the same item twice
- Value is changed by another transaction T' between the two reads
- T receives different values for the two reads of the same item

Transactions

Why recovery is needed (What causes a Transaction to fail)

1. **A computer failure (system crash):** A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.

2. **A transaction or system error**: Some operations in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of wrong parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

Transactions

Why recovery is needed (What causes a Transaction to fail)

3. Local errors or exception conditions detected by the transaction:

Certain conditions force cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.

4. Concurrency control enforcement:

The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.

Transactions

Why recovery is needed (What causes a Transaction to fail)

5. Disk failure:

Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

6. Physical problems and catastrophes:

This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

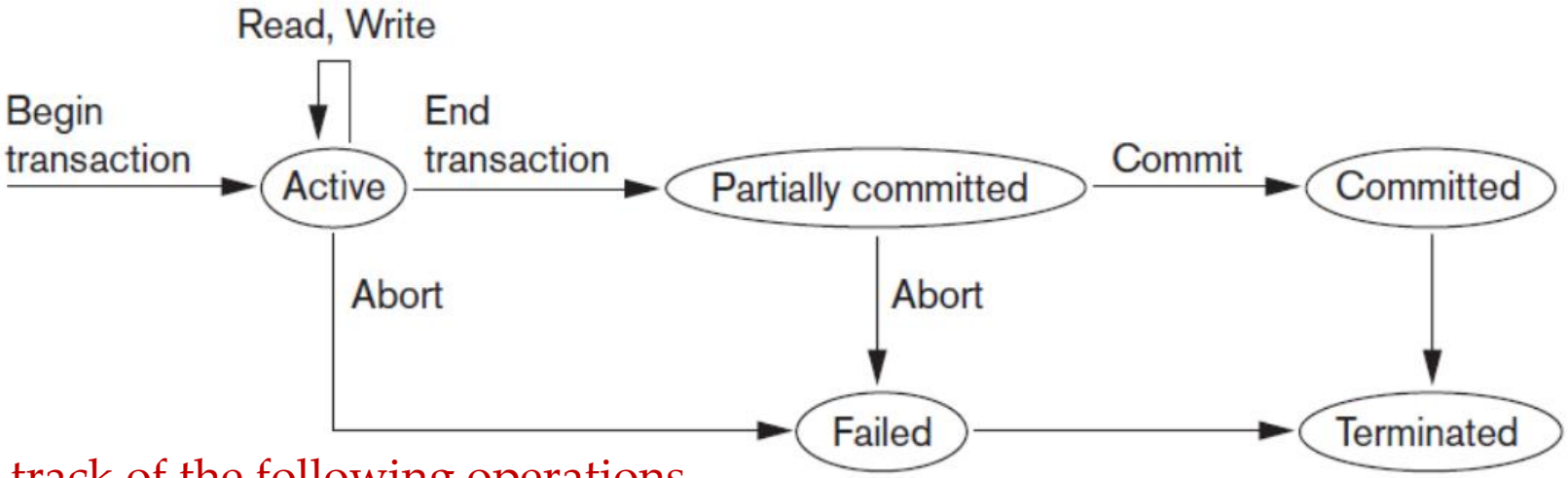
Transaction and System Concepts

- A transaction is an atomic unit of work that is either completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.

Transaction States:

- Active state
- Partially committed state
- Committed state
- Failed state
- Terminated State

Transaction and System Concepts



The recovery manager keeps track of the following operations

- begin_transaction: This marks the beginning of transaction execution.
- read or write: These specify read or write operations on the database items that are executed as part of a transaction.

Transaction and System Concepts

The recovery manager keeps track of the following operations

- end_transaction: This specifies that read and write transaction operations have ended and marks the end limit of transaction execution. At this point, it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.
- commit_transaction: This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
- rollback (or abort): This signals that the transaction has ended unsuccessfully so that any changes or effects that the transaction may have applied to the database must be undone.

Transaction and System Concepts

The recovery manager keeps track of the following operations

- undo: Similar to rollback except that it applies to a single operation rather than to a whole transaction.
- redo: This specifies that certain transaction operations must be redone to ensure that all the operations of a committed transaction have been applied successfully to the database.

Transaction and System Concepts

Types of log record:

1. [start_transaction,T]: Records that transaction T has started execution.
2. [write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.
3. [read_item,T,X]: Records that transaction T has read the value of database item X.
4. [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. [abort,T]: Records that transaction T has been aborted.

ACID Properties

To preserve the integrity of data, the database system must ensure:

- Atomicity: Either all operations of the transaction are properly reflected in the database or none are.
- Consistency: Execution of a transaction in isolation preserves the consistency of the database.
- Isolation: Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions; intermediate transaction results must be hidden from other concurrently executed transactions.
- Durability: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

ACID Properties

Example of Fund Transfer

- Let Ti be a transaction that transfers 50 from account A to B. This transaction can be illustrated as follows:

Transfer \$50 from account A to B:

T_i :
- read(A)
- $A := A - 50$
- write(A)
- read(B)
- $B := B + 50$
- write(B)

Consistency: the sum of A and B is unchanged by the execution of the transaction.

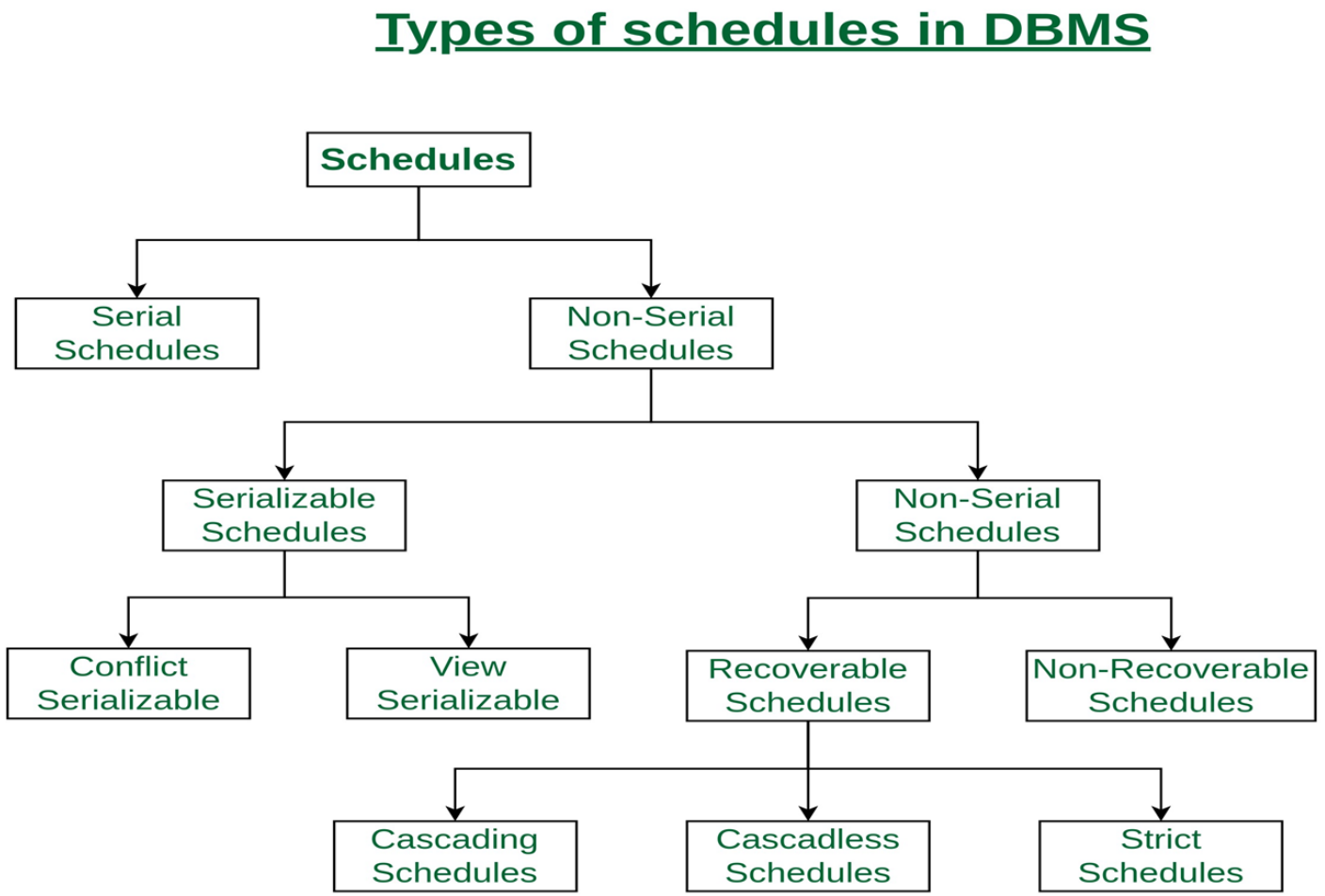
Atomicity: if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

Durability: once the user has been notified that the transaction has completed, the updates to the database by the transaction must persist despite failures.

Isolation: between steps 3 and 6, no other transaction should access the partially updated database, or else it will see an inconsistent state (the sum A + B will be less than it should be).

Schedules

- A schedule is the order in which the operations of multiple transactions appear for execution.



Example - Schedules

- Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B*. The following is a serial schedule (Schedule 1 in the text), in which T_1 is followed by T_2.

Schedule 1

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Let A=100, B=200

A=50, B=250

A=45, B=255

- The sum A+B is preserved.

Concurrent Schedule

- Let T_1 and T_2 be the transactions defined previously. The following schedule (Schedule 3 in the text) is not a serial schedule, but it is *equivalent* to Schedule 1.

Schedule 3

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Let A=100, B=200

A=50, B=200

A=45, B=200

A=45, B=250

A=45, B=255

The Schedules 1,2 and 3, the sum A + B is preserved.

Schedules

1. Serial Schedules

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

Serial schedules are always-

- Consistent
- Recoverable
- Cascadeless
- Strict

- A schedule where tasks (transactions) are executed without mixing with each other is called a "Serial Schedule" .
- In simple terms, transactions in a serial schedule are executed one at a time.

Schedules

1. Serial Schedules

Example

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

Transaction-2 starts its execution after the completion of Transaction-1.

Schedules

2. Non-serial Schedule

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

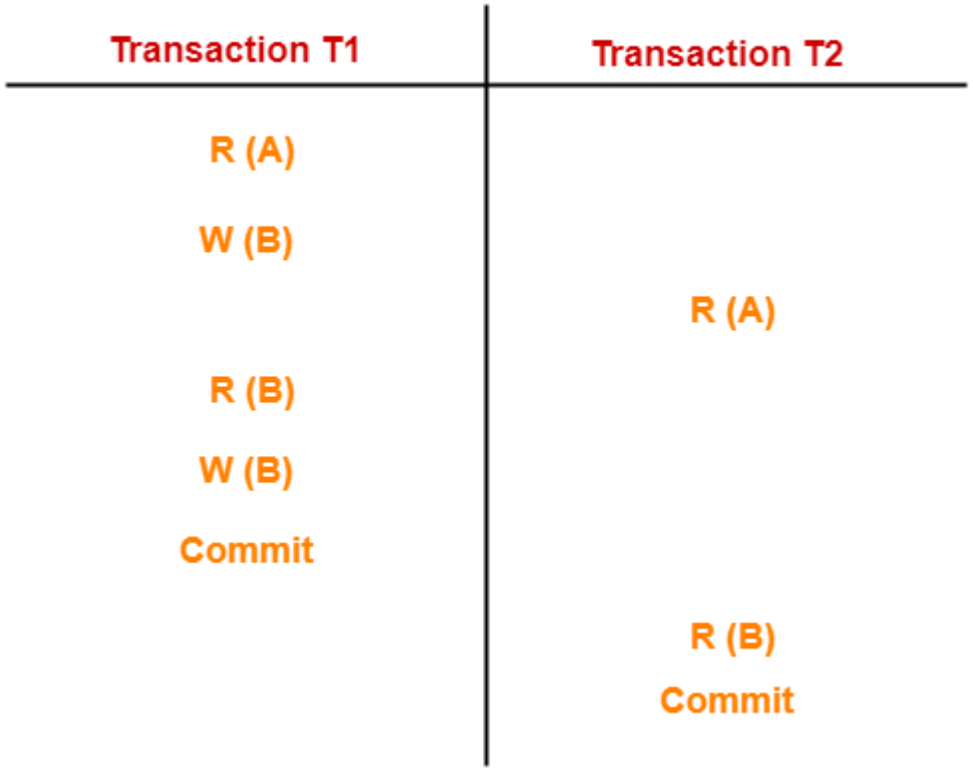
Non-serial schedules are NOT always

- Consistent
- Recoverable
- Cascadeless
- Strict

Schedules

2. Non-serial Schedule

Example



There are two transactions T1 and T2 executing concurrently.

- The operations of T1 and T2 are interleaved.
- So, this schedule is an example of a Non-Serial Schedule.

Schedule Notation

- S_a - A schedule of one or more transactions
- T_i - A transaction.
- $r_i(X)$ - Transaction T_i performs a READ of data item X.
- $w_i(X)$ - Transaction T_i performs a WRITE of data item X.
- a_i - Transaction i aborts.
- c_i - Transaction i commits.

Finding the Number of Schedules

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

Total Number of Schedules

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

$$\frac{(N1 + N2 + N3 + ... + Nn)!}{N1! \times N2! \times N3! \times .... \times Nn!}$$

Total Number of Serial Schedules

Total number of serial schedules
= Number of different ways of arranging n transactions
= n!

Total Number of Non-Serial Schedules

Total number of non-serial schedules

= Total number of schedules - Total number of serial schedules

Finding the Number of Schedules

Example

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

$$\text{Total number of schedules} = \frac{(2+3+4)!}{2! \times 3! \times 4!}$$

$$= 1260$$

Total Number of Serial Schedules

Total number of serial schedules
= Number of different ways of arranging 3 transactions
= 3!
= 6

Total Number of Non-Serial Schedules

= 1260 – 6

= 1254

Serializability in DBMS

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

Serializable Schedules

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

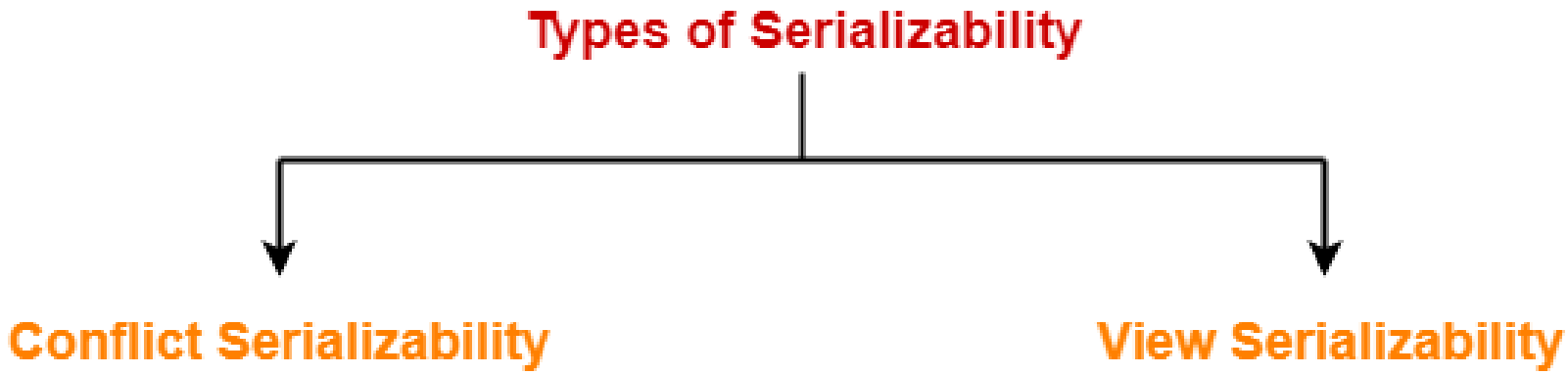
Characteristics

-Serializable schedules behave exactly the same as serial schedules. Thus, serializable schedules are always

- Consistent
- Recoverable
- Casacadeless
- Strict

Serializability in DBMS

Types of Serializability



Serializability in DBMS

1.Conflict Serializability

- If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then it is called as a conflict serializable schedule.

Conflicting Operations

- Both operations belong to different transactions
- Both operations are on the same data item
- At least one of the two operations is a write operation

Transaction T1	Transaction T2
R1 (A)	
W1 (A)	
	R2 (A)
R1 (B)	

This schedule,
•W1 (A) and R2 (A) are called as conflicting operations.
•This is because all the above conditions hold true for them.

- Write-Read (WR) Conflict: Reading Uncommitted data.
- Read-Write (RW) Conflict: Unrepeatable Reads
- Write-Write (WW) Conflict: Overwriting Uncommitted Data.

Serializability in DBMS

1.Conflict Serializability

- Instructions I_i and I_j of transactions T_i and T_j respectively, conflict if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.

1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j ------- don't conflict.(order does not matter)
2. I_i = **read**(Q), I_j = **write**(Q). --------- conflict. (If i comes before j, it does not read the value of Q. If j comes before i, then Ti reads the value of Q.Order of i and j matter here)
3. I_i = **write**(Q), I_j = **read**(Q). --------- conflict
4. I_i = **write**(Q), I_j = **write**(Q). --------- conflict (oder of these instructions does not affect either Ti or Tj. But the next read(Q) of S is affected.

- Conclusion: I and J conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a Write operation.

Testing for Serializability

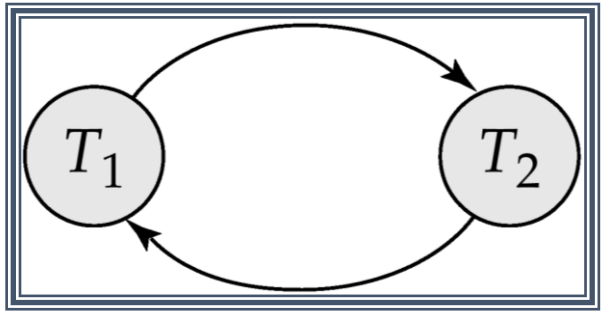
With the help of a precedence graph it can be tested.

Now, present a simple and efficient method for determining the conflict serializability of a schedule.

- Precedence graph — a direct graph G= (V,E) where the V is a set of vertices are the transactions participating in the schedule. E is set of Edges T_i -> T_j.

- Arcs Ti--->Tj exist when the following 3 conditions are satisfied

1. Ti executes write(Q) before Tj executes read(Q)
2. Ti executes read(Q) before Tj executes write(Q)
3. Ti executes write(Q) before Tj executes write(Q)

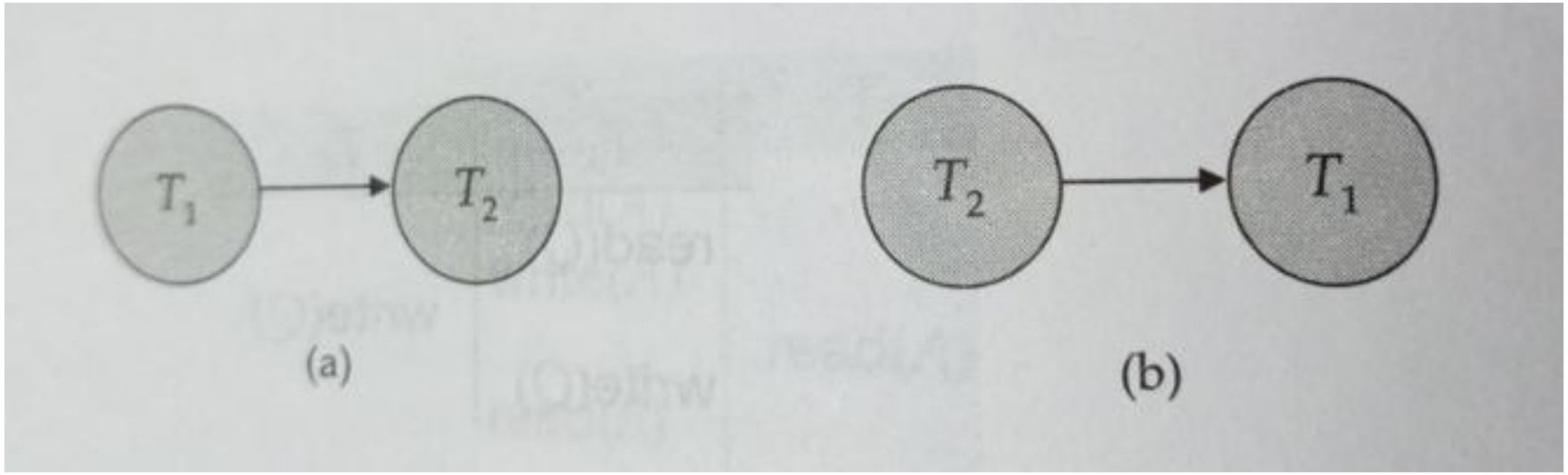


Testing for Serializability

- The precedence graph for schedule 1 and 2 are presented here.

  Fig. a) T1 -> T2 means all the instructions of T1 are executed before the first instruction of T2.

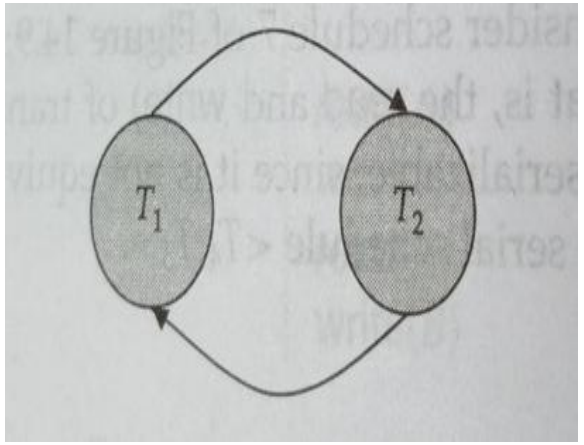
  Fig. b) T2 -> Similarly all the instructions of T2 are completed before the first instruction of T1



Testing for Serializability

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

- Precedence graph of schedule S below.



- It contains the edge T1 -> T2 because T1 executes read(A) before T2 execute Write(A)
- T2 ->T1 because T2 executes read(B) before T1 executes write(B)
- If the precedence graph has **a cycle**, then schedule S is not conflict serializable.

Checking Whether a Schedule is Conflict Serializable Or Not

Step 1: We will find and list all of the operations that are in conflict.

Step 2: Then we will draw one node for each transaction in a precedence graph.

Step 3: After that, we will draw an edge from Ti to Tj for each conflict pair, so that if Xi (V) and Yj (V) form a conflict pair, draw an edge from Ti to Tj.
This ensured that Ti is executed before Tj.

Step 4: At the end, we will check the graph to see if any cycles have formed. If no cycles are found, the schedule is conflict serializable; otherwise, it is not.

Checking Whether a Schedule is Conflict Serializable Or Not

Example

T_1	T_2	T_3
R(x)		
		R(y)
		R(x)
	R(y)	
	R(z)	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		

Step 1: We will find and list all of the operations that are in conflict.

Conflict Pair

1.Read [3] (x) - Write [1] (x)

2.Read [2] (y) - Write [3] (y)

3.Read [2] (z) - Write [1] (z)

4.Write [2] (z) - Read [1] (z)

5.Write [2] (z) - Write [1] (z)

Where I and j denote two different transactions Ti and Tj.

Checking Whether a Schedule is Conflict Serializable Or Not

Example cont'd

T_1	T_2	T_3
R(x)		
		R(y)
		R(x)
	R(y)	
	R(z)	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		

Conflict Pair

1.Read [3] (x) - Write [1] (x)

2.Read [2] (y) - Write [3] (y)

3.Read [2] (z) - Write [1] (z)

4.Write [2] (z) - Read [1] (z)

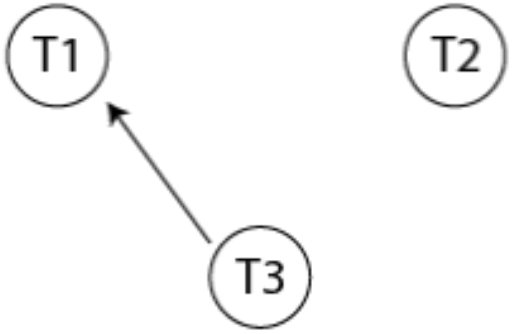
5.Write [2] (z) - Write [1] (z)

Step 2: We will create a precedence graph.

Here in this example, we have 3 transactions in this schedule. So we will create 3 nodes.

Precedence Graph

Step 3: We will draw an edge from Ti to Tj for each conflict pair.



Precedence Graph

1. Read [3] (x) - Write [1] (x)

- We will draw an edge from T[3] to T[1] for this conflict pair.

Checking Whether a Schedule is Conflict Serializable Or Not

Example cont'd

T_1	T_2	T_3
R(x)		
		R(y)
		R(x)
	R(y)	
	R(z)	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		

Conflict Pair

1.Read [3] (x) – Write [1] (x)

2.Read [2] (y) – Write [3] (y)

3.Read [2] (z) – Write [1] (z)

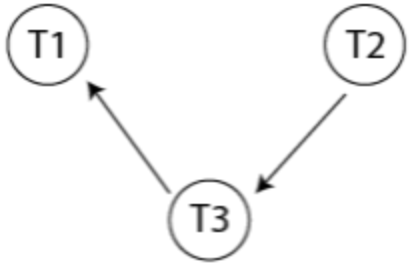
4.Write [2] (z) – Read [1] (z)

5.Write [2] (z) – Write [1] (z)

Step 3 Cont'd: We will draw an edge from Ti to Tj for each conflict pair.

2. Read [2] (y) – Write [3] (y)

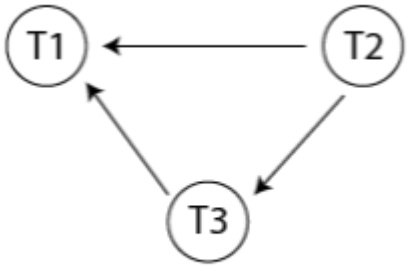
- We will draw an edge from T[2] to T[3] for this conflict pair.



Precedence Graph

3. Read [2] (z) – Write [1] (z)

- For this conflict pair, we will draw an edge from T[2] to T[1].



Precedence Graph

Checking Whether a Schedule is Conflict Serializable Or Not

Example cont'd

T_1	T_2	T_3
R(x)		
		R(y)
		R(x)
	R(y)	
	R(z)	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		

Conflict Pair

1.Read [3] (x) – Write [1] (x)

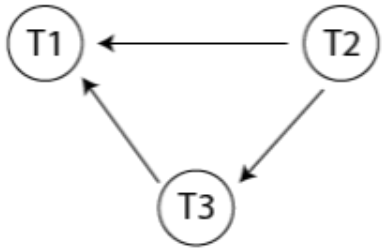
2.Read [2] (y) – Write [3] (y)

3.Read [2] (z) – Write [1] (z)

4.Write [2] (z) – Read [1] (z)

5.Write [2] (z) – Write [1] (z)

Step 3 Cont'd: We will draw an edge from Ti to Tj for each conflict pair.



Precedence Graph

4. Write [2] (z) – Read [1] (z)

- For this conflict pair, we will draw an edge from T[2] to T[1]. But we already drew an edge from T[2] to T[1] in conflict pair 3. so we will not draw an edge again..

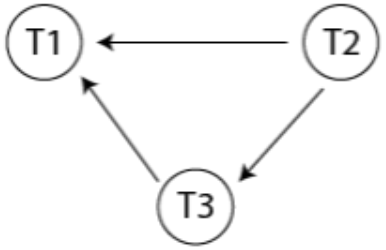
5.Write [2] (z) – Write [1] (z)

- For this conflict pair, we will draw an edge from T[2] to T[1]. But we already drew an edge from T[2] to T[1] in conflict pair 3. so we will not draw an edge again.

Checking Whether a Schedule is Conflict Serializable Or Not

Example cont'd

T_1	T_2	T_3
R(x)		
		R(y)
		R(x)
	R(y)	
	R(z)	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		



Precedence Graph

Step 4: Now we will check the graph to see if any cycles or loops have formed. Here in this example, we can not find out any cycles or loops. So this schedule is conflict serializable.

Conflict Pair

1.Read [3] (x) - Write [1] (x)

2.Read [2] (y) - Write [3] (y)

3.Read [2] (z) - Write [1] (z)

4.Write [2] (z) - Read [1] (z)

5.Write [2] (z) - Write [1] (z)

Serializability in DBMS

Conflict Equivalent

- Let us consider a schedule S in which there are two consecutive instructions I and J.
- If I and J refers different data items, then we swap I and J without affecting the results of any instruction in the schedule.
- However, if both i and j refer the same data itemQ, then the order of the two steps may matter.
- If S is conflict equivalent to a serial schedule S' then S is conflict serializable.

Serializability in DBMS

1.Conflict Serializability - Conflict Equivalent Example

- In the conflict equivalent, one can be transformed into 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).
- To illustrate the conflicting instructions, we consider schedule S.
- Only Read and Write operations are considered from dig1 as in dia2

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

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

Serializability in DBMS

1.Conflict Serializability – (Conflict Equivalent)Example

1. Write(A) of T1 conflicts with Read(A) of T2.

- Hence the order of Write (A) in T1 can not be swapped with Read(A) in T2.

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

T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Serializability in DBMS

1.Conflict Serializability – (Conflict Equivalent)Example

2. Write(A) of T2 does not conflict with Read(B) of T1 because T1 & T2 operates on different data items A and B

- Hence the order of Write (A) in T2 can be swapped with Read(B) in T1

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

T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T1	T2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Serializability in DBMS

1.Conflict Serializability – (Conflict Equivalent)Example

3. read(A) of T2 does not conflict with read(B) of T1

- Hence the order of read (A) in T2 can be swapped with read(B) in T1 .

T1	T2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

T1	T2
read(A)	
write(A)	
read(B)	
	read(A)
	write(A)
write(B)	
	read(B)
	write(B)

Serializability in DBMS

1.Conflict Serializability – (Conflict Equivalent)Example

4. write(B) in T1 does not conflict with write(A) in T2

- Hence the order of write(B) in T1 can be swapped with write(A) in T2 .

T1	T2
read(A)	
write(A)	
read(B)	
	read(A)
	write(A)
write(B)	
	read(B)
	write(B)

T1	T2
read(A)	
write(A)	
read(B)	
	read(A)
write(B)	
	write(A)
	read(B)
	write(B)

Serializability in DBMS

1.Conflict Serializability – (Conflict Equivalent)Example

5. write(B) in T1 does not conflict with read(A) in T2

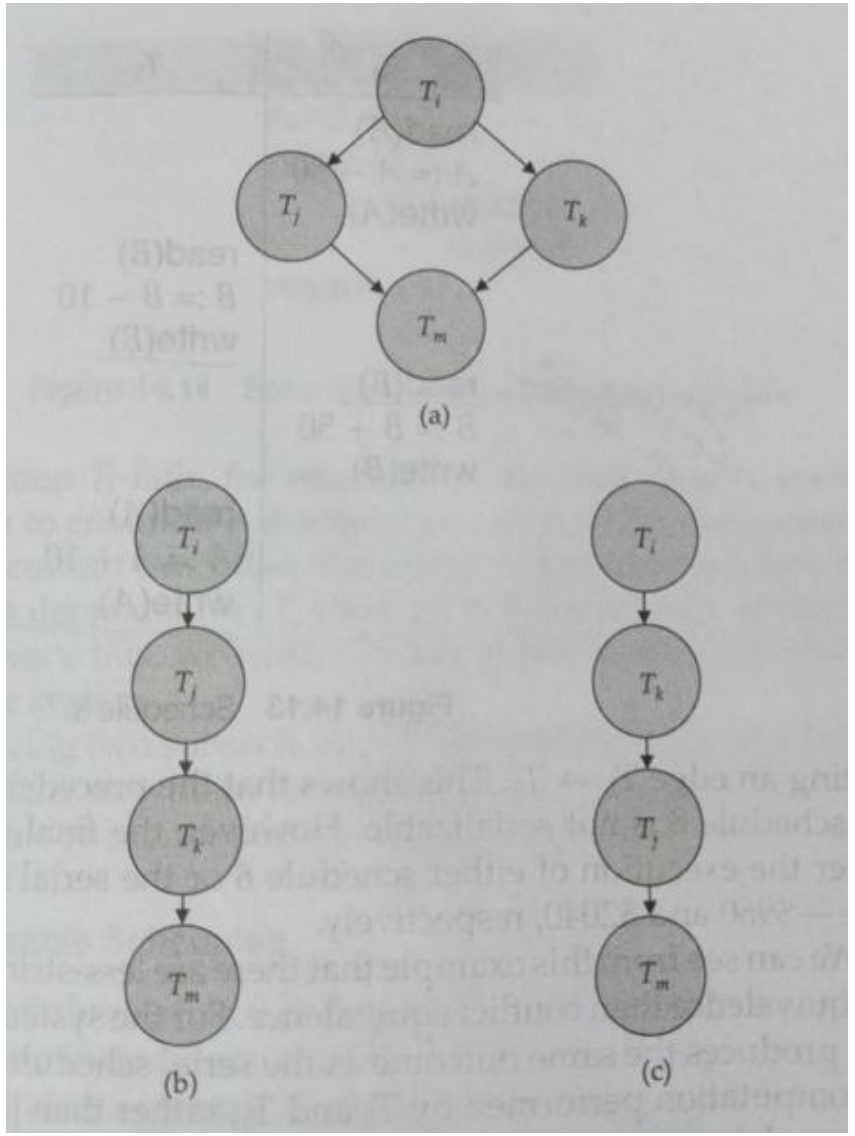
- Hence the order of write(B) in T1 can be swapped with read(A) in T2 .

T1	T2
read(A)	
write(A)	
read(B)	
	read(A)
write(B)	
	write(A)
	read(B)
	write(B)

T1	T2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Topological Sorting

- Several possible linear orders that can be obtained through a topological sort.



View Serializability

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

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

- **Initial Read**
- **Update Read**
- **Final Write**

	A	B	C
Initial Read	T1,T2	T3,T2	T2
Update Read	T1	T2	T3
Final Write	T1	T2	T3

A: T2→ T1
B: T3→ T2
C: T2→T3

So, it's a Conflict of Serializability.

if a schedule is conflict serializable, then it is surely view serializable.

View Serializability

Problem-04:

Check whether the given schedule S is view serializable or not. If yes, then give the serial schedule.

$$S : R_1(A) , W_2(A) , R_3(A) , W_1(A) , W_3(A)$$

View Serializability

For simplicity and better understanding, we can represent the given schedule pictorially as-

T1	T2	T3
R (A)		
	W (A)	
		R (A)
W (A)		
		W (A)

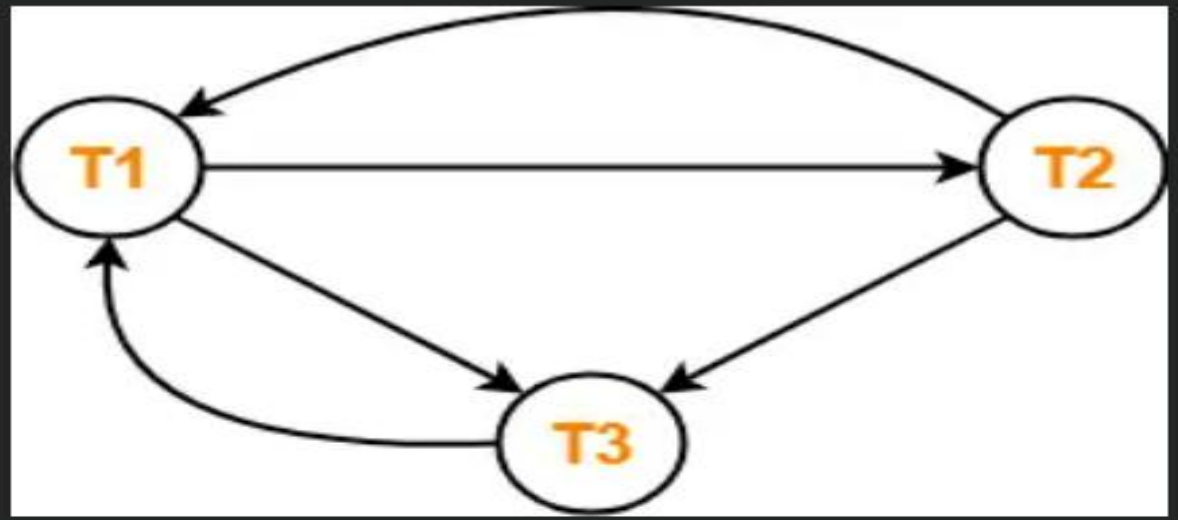
- We know, if a schedule is conflict serializable, then it is surely view serializable.
- So, let us check whether the given schedule is conflict serializable or not.

View Serializability

Step-01:

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

- $R_1(A)$, $W_2(A)$ $(T_1 \rightarrow T_2)$
- $R_1(A)$, $W_3(A)$ $(T_1 \rightarrow T_3)$
- $W_2(A)$, $R_3(A)$ $(T_2 \rightarrow T_3)$
- $W_2(A)$, $W_1(A)$ $(T_2 \rightarrow T_1)$
- $W_2(A)$, $W_3(A)$ $(T_2 \rightarrow T_3)$
- $R_3(A)$, $W_1(A)$ $(T_3 \rightarrow T_1)$
- $W_1(A)$, $W_3(A)$ $(T_1 \rightarrow T_3)$



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

Now,

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

View Serializability

Checking for Blind Writes-

- There exists a blind write W_2 (A) in the given schedule S.
- Therefore, the given schedule S may or may not be view serializable.

Now,

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

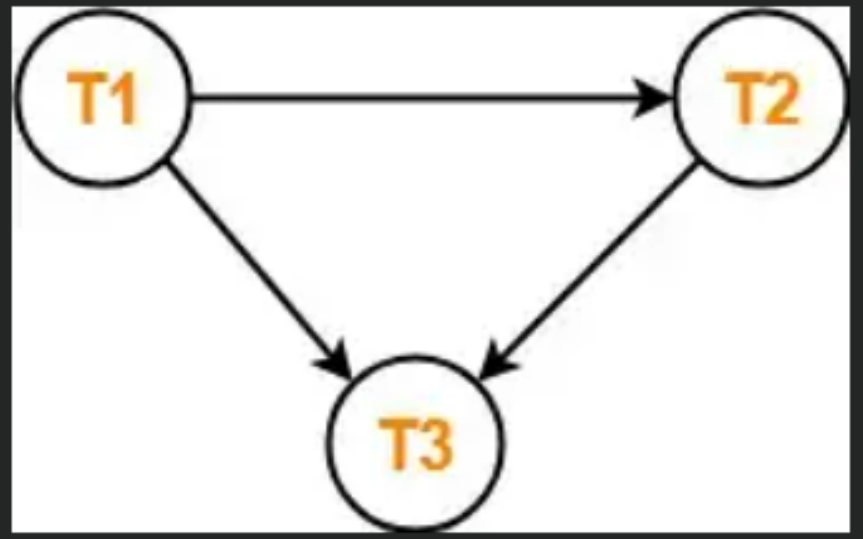
View Serializability

Drawing a Dependency Graph-

- T1 firstly reads A and T2 firstly updates A.
- So, T1 must execute before T2.
- Thus, we get the dependency **T1 → T2**.
- Final updation on A is made by the transaction T3.
- So, T3 must execute after all other transactions.
- Thus, we get the dependency **(T1, T2) → T3**.
- From write-read sequence, we get the dependency **T2 → T3**

View Serializability

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



- Clearly, there exists no cycle in the dependency graph.
- Therefore, the given schedule S is view serializable.
- The serialization order T1 → T2 → T3.

View Serializability

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

T1	T2
R (A)	
A = A + 10	
	R (A)
	A = A + 10
W (A)	
	W (A)
R (B)	
B = B + 20	
	R (B)
	B = B x 1.1
W (B)	
	W (B)

Checking Whether S is Conflict Serializable Or Not-

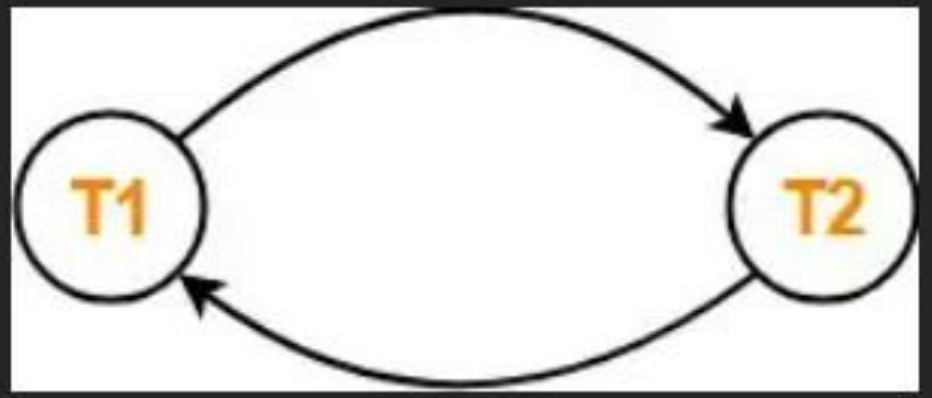
Step-01:

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

- $R_1(A)$, $W_2(A)$ $(T_1 \rightarrow T_2)$
- $R_2(A)$, $W_1(A)$ $(T_2 \rightarrow T_1)$
- $W_1(A)$, $W_2(A)$ $(T_1 \rightarrow T_2)$
- $R_1(B)$, $W_2(B)$ $(T_1 \rightarrow T_2)$
- $R_2(B)$, $W_1(B)$ $(T_2 \rightarrow T_1)$

View Serializability

Draw the precedence graph-



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

View Serializability

Now,

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

Checking for Blind Writes-

- There exists no blind write in the given schedule S.
- Therefore, it is surely not view serializable.

View Serializability

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

T1	T2	T3
R (A)		
	R (A)	
		W (A)
W (A)		

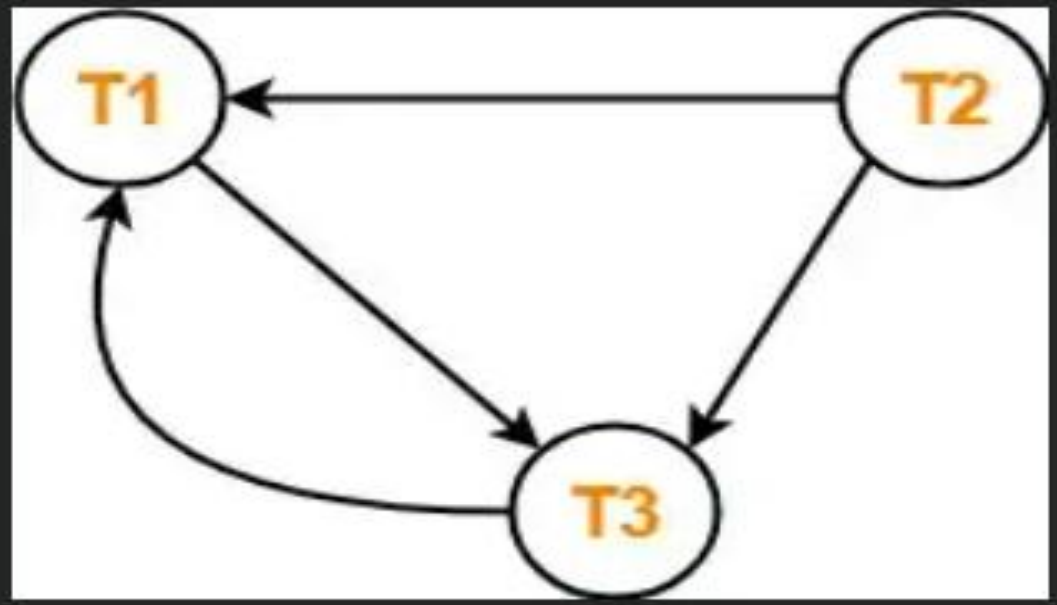
Step-01:

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

- $R_1(A), W_3(A)$ $(T_1 \rightarrow T_3)$
- $R_2(A), W_3(A)$ $(T_2 \rightarrow T_3)$
- $R_2(A), W_1(A)$ $(T_2 \rightarrow T_1)$
- $W_3(A), W_1(A)$ $(T_3 \rightarrow T_1)$

View Serializability

Draw the precedence graph-



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

View Serializability

Checking for Blind Writes-

- There exists a blind write W_3 (A) in the given schedule S.
- Therefore, the given schedule S may or may not be view serializable.

Now,

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

View Serializability

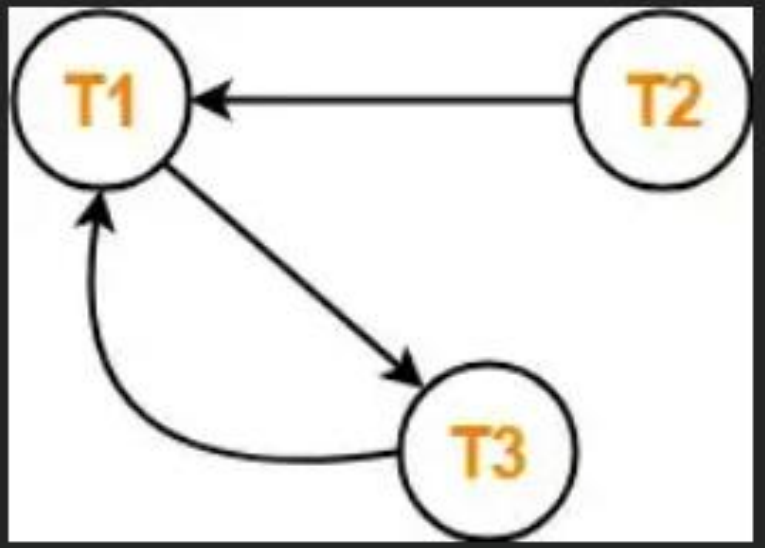
Drawing a Dependency Graph-

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

T1	T2	T3
R (A)		
	R (A)	
		W (A)
W (A)		

View Serializability

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



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

Recoverable Schedules

Non Recoverable Schedule

T1	T2
Read(X)	
X=X+10	
Write(X)	
	Read(X)
	X=X+10
	Write(X)
	Commit
Failure point	
Commit	

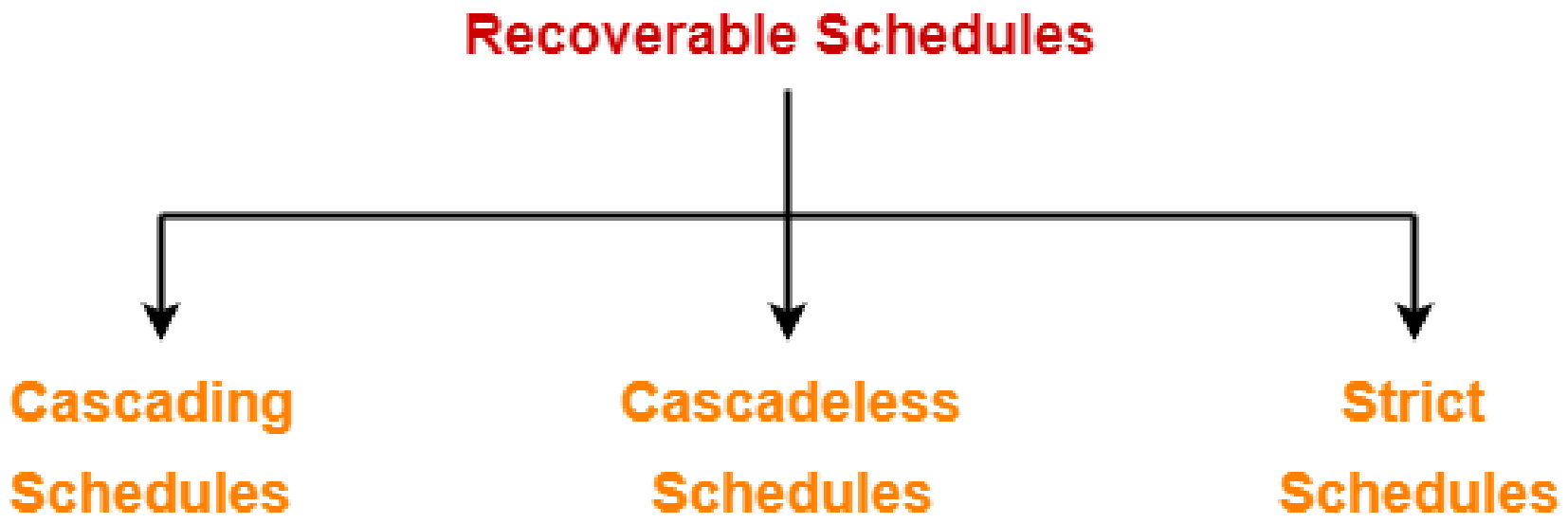
Recoverable Schedule

T1	T2
Read(X)	
X=X+10	
Write(X)	
Commit	
	Read(X)
	X=X+10
	Write(X)
	Commit

- 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 called as a Recoverable Schedule.

Recoverable schedules

Types of Recoverable Schedules



Cascading Schedule

T1	T2	T3	T4
R (A)			
W (A)			
	R (A)		
	W (A)		
		R (A)	
		W (A)	
			R (A)
			W (A)
Failure			

Cascading Recoverable Schedule

- Transaction T2 depends on transaction T1.
- Transaction T3 depends on transaction T2.
- Transaction T4 depends on transaction T3.

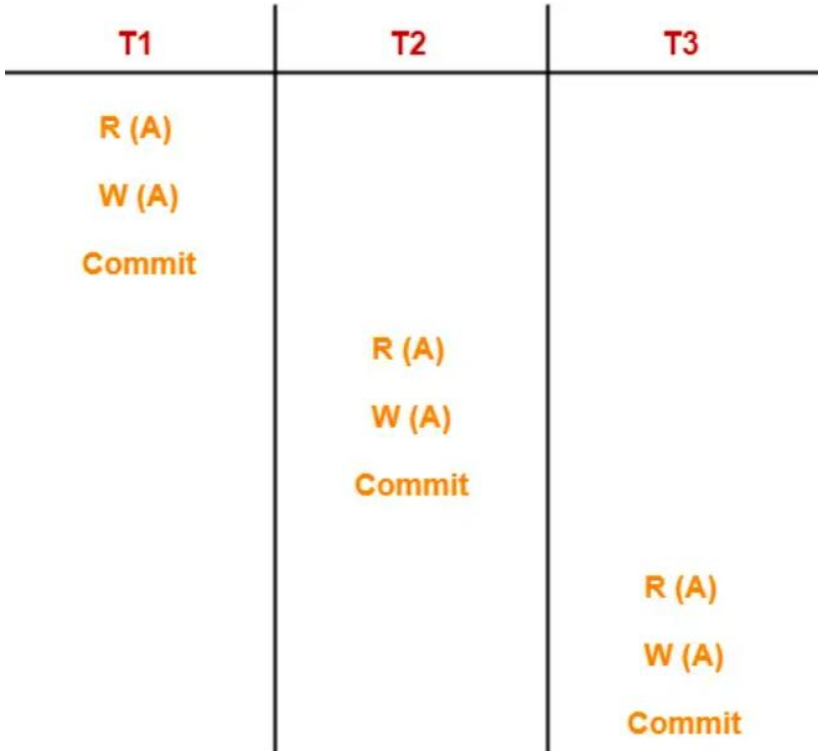
In this schedule,

- The failure of transaction T1 causes the transaction T2 to rollback.
- The rollback of transaction T2 causes the transaction T3 to rollback.
- The rollback of transaction T3 causes the transaction T4 to rollback.

Such a rollback is called as a **Cascading Rollback**

- If in a schedule, the failure of one transaction causes several other dependent transactions to rollback or abort, then such a schedule is called as a Cascading Schedule or Cascading Rollback or Cascading Abort.
- It simply leads to the wastage of CPU time.

Cascadeless schedules



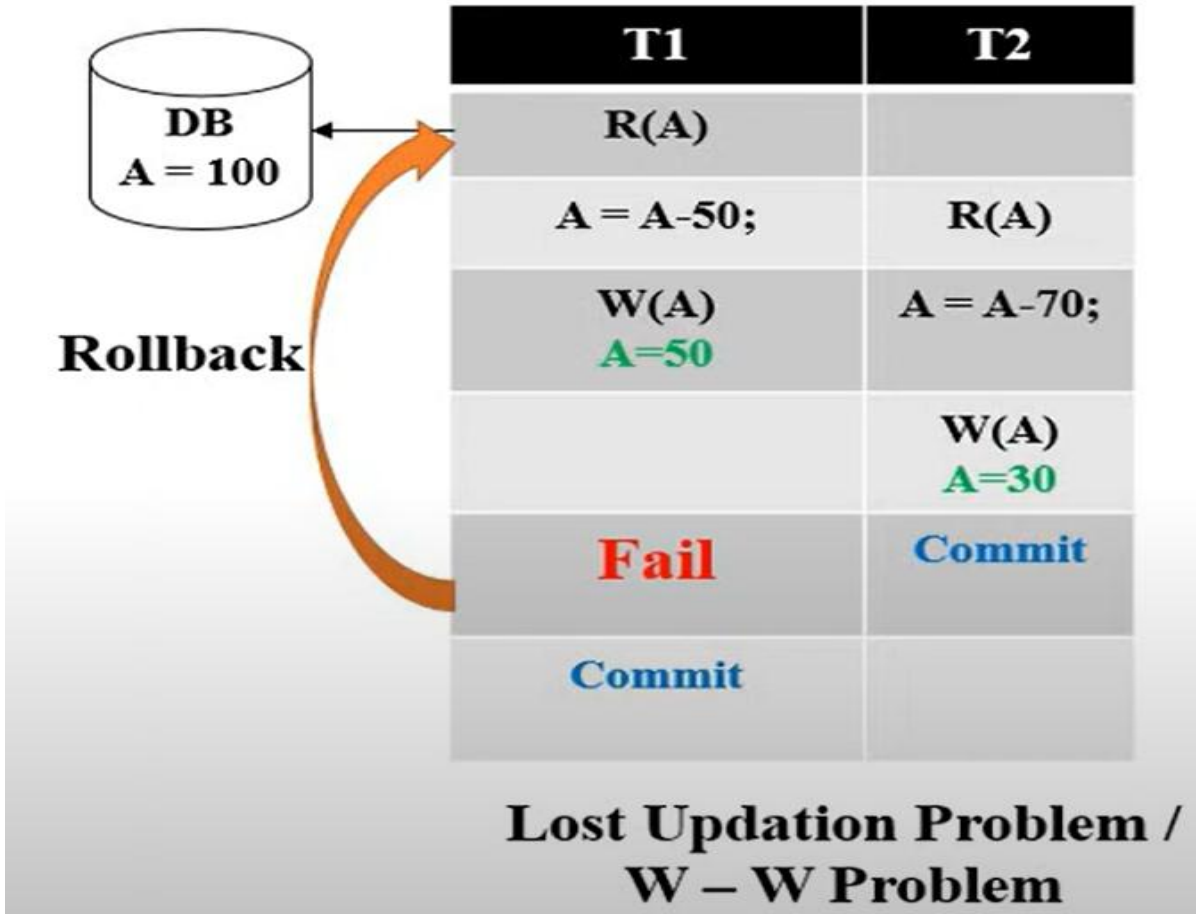
T1	T2	T3
R (A)		
W (A)		
Commit		
	R (A)	
	W (A)	
	Commit	
		R (A)
		W (A)
		Commit

Cascadeless schedule allows only committed read operations.
However, it allows uncommitted write operations.

- A Cascadeless schedule allows only committed read operations.
- Therefore, it avoids cascading roll back and thus saves CPU time.

However, it allows uncommitted write operations.

Cascadeless schedules



- A lost update problem occurs due to the update of the same record by two different transactions at the same time.

Strict Schedule

If in a schedule, a transaction is neither allowed to read nor write a data item until the last transaction that has written it is committed or aborted, then such a schedule is called as a Strict Schedule.

In other words,

- Strict schedule allows only committed read and write operations.
- Clearly, strict schedule implements more restrictions than cascadeless schedule.

T1	T2
R(A)	
W(A)	
Commit	
	W(A)
	R(A)
	Commit

Strict Schedule

Locking

- Data items can lock to maintain isolation.
- Two kind of locks: i) Shared and Exclusive

- Shared lock- read

- Exclusive- write

Timestamps

- Timestamp is a unique identifier created by the DBMS to identify a transaction

- Read Timestamps

- Write Timestamps