DBMS: Transaction Processing

Akhilesh Arya

Transaction

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

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

X account

```
Open_Account(X)
Old_Balance = X.balance
New_Balance = Old_Balance - 800
X.balance = New_Balance
Close_Account(X)
```

Y account

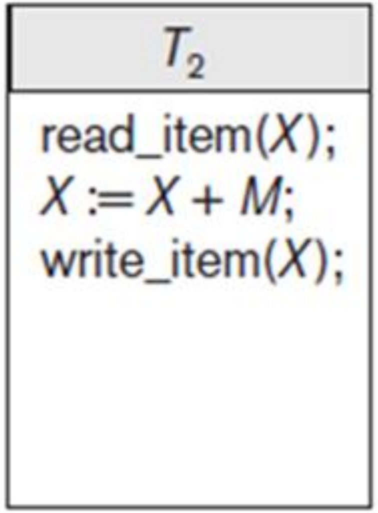
```
Open_Account(Y)
Old_Balance = Y.balance
New_Balance = Old_Balance + 800
Y.balance = New_Balance
Close_Account(Y)
```

Operations of Transaction

- read_item(X)
  - Reads a database item named X into a program variable named X
  - Process includes finding the address of the disk block, and copying to a variable in a memory buffer
- write_item(X)
  - Writes the value of program variable X into the database item named X
  - Process includes finding the address of the disk block, copying to and from a memory buffer, and storing the updated disk block back to disk

Read and Write operations

- **Example:** in case of the complete transaction in a DBMS we can consider it as a combination of read and write operation



T_2

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

Cont..

- Let's assume the value of X before starting of the transaction is 4000.
  - The first operation reads X's value from database and stores it in a buffer.
  - The second operation will increase the value of X by M (500). So buffer will contain 4500.
  - The third operation will write the buffer's value to the database. So X's final value will be 4500.

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

Cont..

- **For example:** If in the above transaction, the credit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the customer
- To solve this problem, we have two important operations:
  - **Commit:** It is used to save the work done permanently.
  - **Rollback:** It is used to undo the work done.

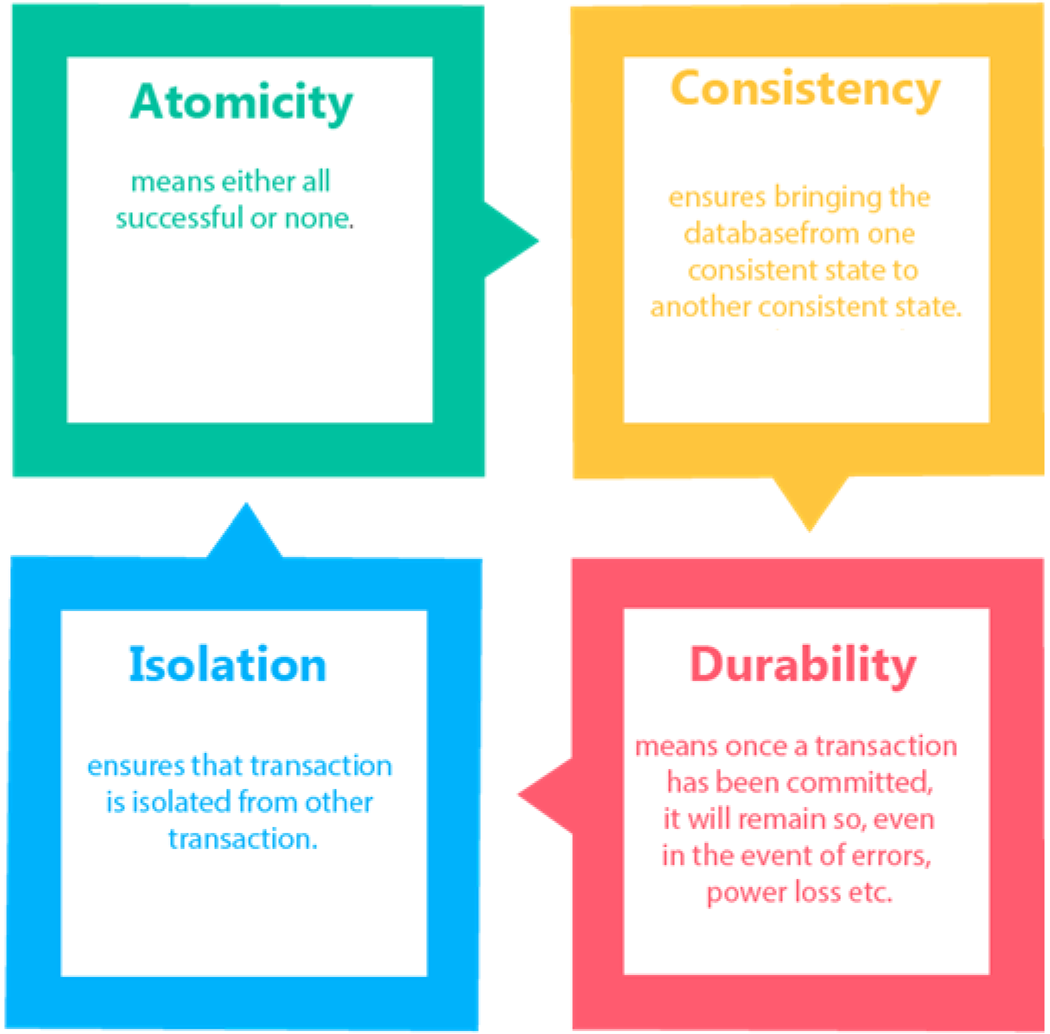
Buffers

- DBMS will maintain several main memory data buffers in the database cache
- When buffers are occupied, a buffer replacement policy is used to choose which buffer will be replaced
  - Example policy: **least recently used**

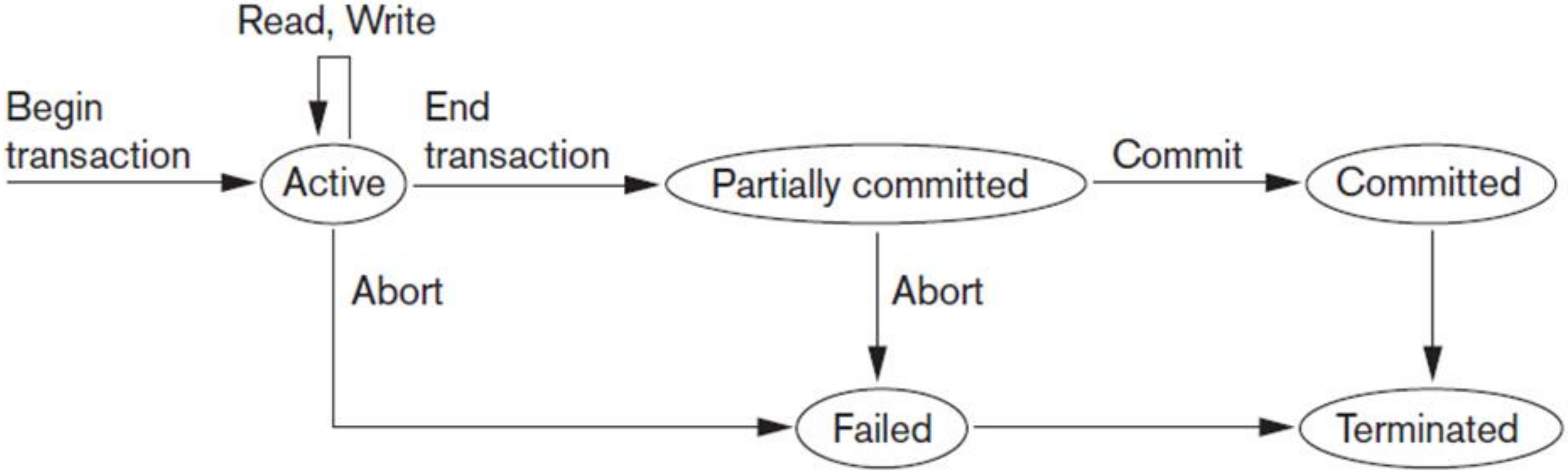
Transaction property

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

  The **ACID** properties


Atomicity
means either all successful or none.
Consistency
ensures bringing the databasefrom one consistent state to another consistent state.
Durability
means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.
Isolation
ensures that transaction is isolated from other transaction.

Transaction States



- **Active state**
  - The active state is the first state of every transaction. In this state, the transaction is being executed
  - For example: Insertion or deletion or updating a record is done here. But all the records are still not saved to the database
- **Partially committed**
  - In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database
  - In the total mark calculation example, a final display of the total marks step is executed in this state

- **Committed**
  - A transaction is said to be in a committed state if it executes all its operations successfully
  - In this state, all the effects are now permanently saved on the database system and the transaction is **terminated**
- **Failed state**
  - If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
  - In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute

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

Dirty Read Problem

- If in this case transaction T1 will roll back at any point after transaction T2 commits then the end result of the T1 and T2 will be different and the database will lead to inconsistent state

T1	T2
Read(a)	
Write(a)	
	Write(a)
	Transactions Commit
Commit	

Unrepeatable read problem

- Two read operations of transaction T2 on ‘a’ will get 2 different values of ‘a’ on different time stamps as the transaction T1 makes some changes on ‘a’ before second read of T2

T1	T2
Read(a)	
	Read(a)
Write(a)	
	Read(a)

Phantom read problem

- Transaction T2 will not get the value of ‘a’ on the second read as transaction T1 deleted ‘a’ before second read of T2

T1	T2
Read(a)	
	Read(a)
Delete(a)	
	Read(a)

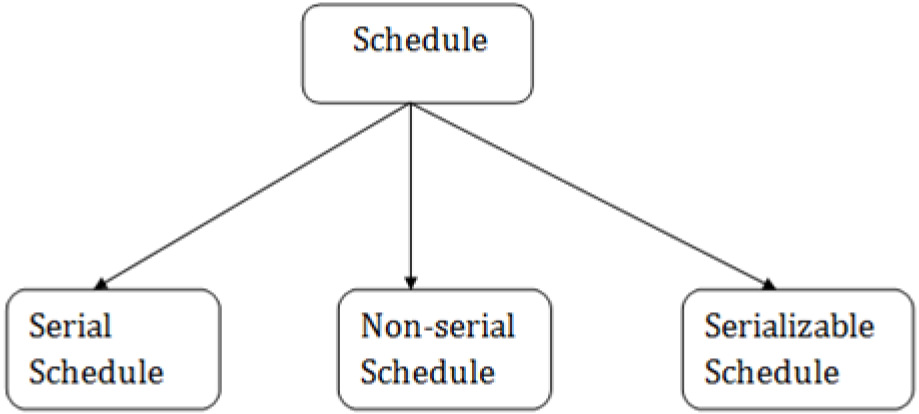
Lost Update problem

- Transaction T1 will commit the value of 'a' updated by T2 instead of its own value because before T1's commit operation T2 updated the value of 'a'
- This is also called at *write-write conflict* problem

T1	T2	
Read(a)		
Write(a)		
	Write(a)	*Blind write*
	Commit	
Commit		

Schedule

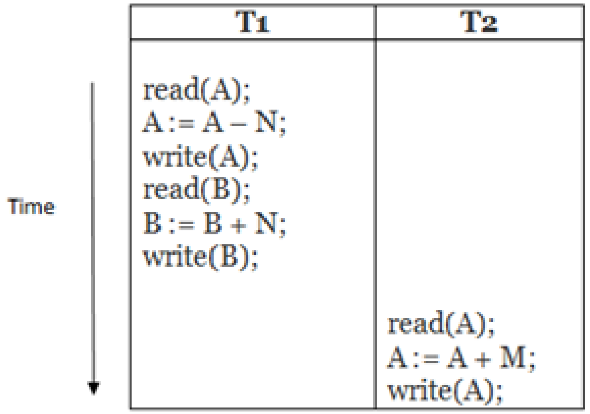
- A series of operation from one transaction to another transaction is known as schedule
- It is used to preserve the order of the operation in each of the individual transaction



1. Serial Schedule

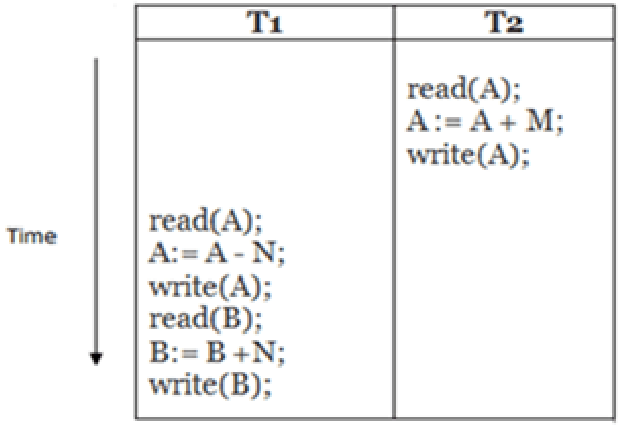
- The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction
- **For example:** Suppose there are two transactions T1 and T2
  - Execute all the operations of T1 which was followed by all the operations of T2
  - Execute all the operations of T2 which was followed by all the operations of T1

Cont..



T1	T2
read(A);	
A := A – N;	
write(A);	
read(B);	
B := B + N;	
write(B);	
	read(A);
	A := A + M;
	write(A);

Schedule A



T1	T2
	read(A);
	A := A + M;
	write(A);
read(A);	
A:= A - N;	
write(A);	
read(B);	
B:= B +N;	
write(B);	

Schedule B

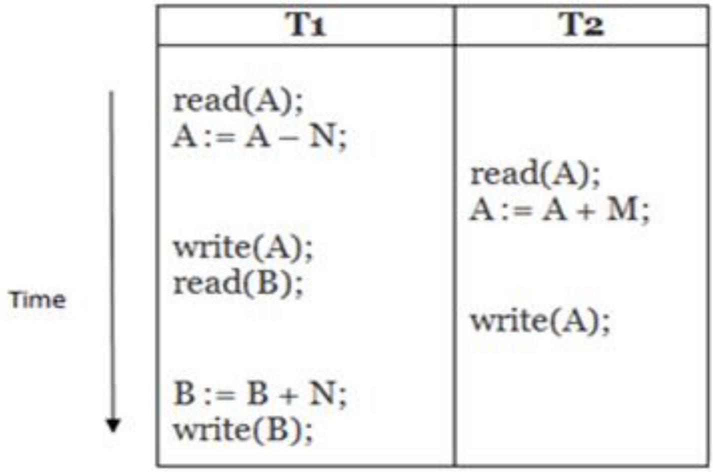
Cont..

- Problem with serial schedules
  - **Limit concurrency** by prohibiting interleaving of operations
  - Unacceptable in practice
  - **Solution**: determine which schedules are equivalent to a serial schedule and allow those to occur concurrently

Non-serial Schedule

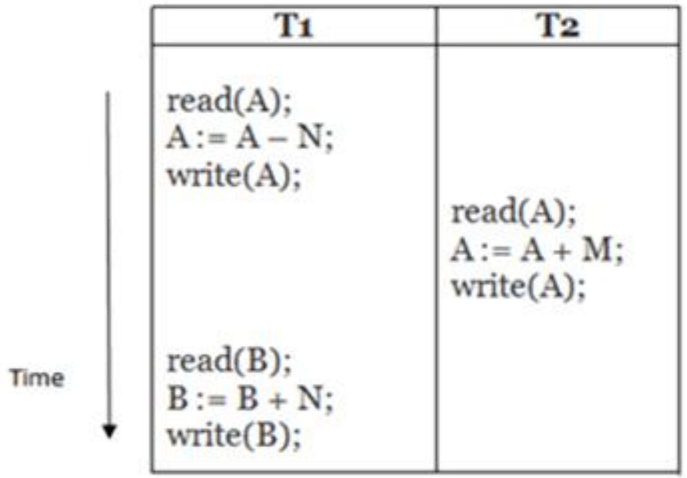
- If interleaving of operations is allowed, then there will be non-serial schedule
- It contains many possible orders in which the system can execute the individual operations of the transactions

Cont..



T1	T2
read(A);	
A := A – N;	
	read(A);
	A := A + M;
write(A);	
read(B);	
	write(A);
B := B + N;	
write(B);	

Schedule C



T1	T2
read(A);	
A := A – N;	
write(A);	
	read(A);
	A := A + M;
	write(A);
read(B);	
B := B + N;	
write(B);	

Schedule D

Serializable schedule

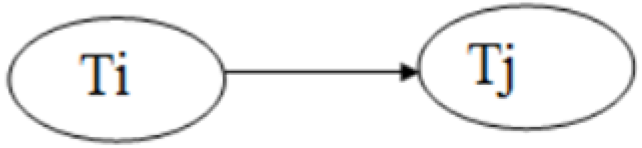
- The serializability of schedules is used to find non-serial schedules that allow the transaction to execute concurrently without interfering with one another
- It identifies which schedules are correct when executions of the transaction have interleaving of their operations
- A non-serial schedule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

Testing of Serializability

- **Precedence graph** is used to test the Serializability of a schedule
- 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:

Cont..

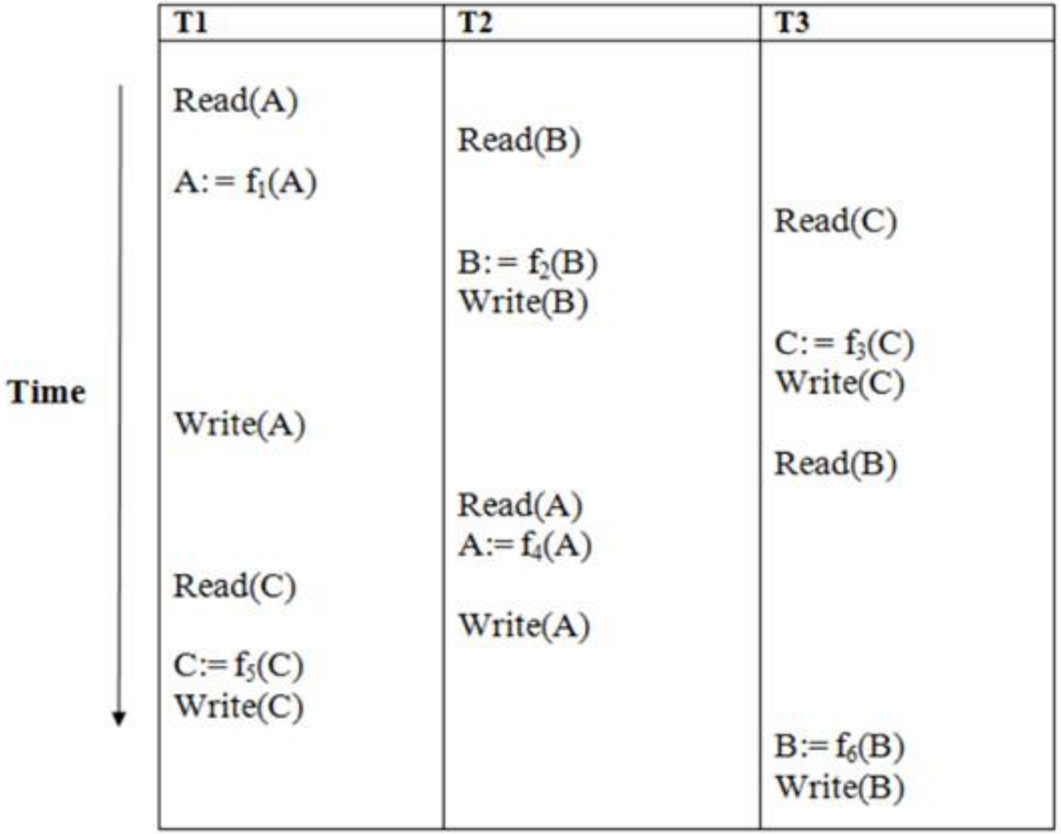
- Create a node Ti → Tj if Ti executes write (Q) before Tj executes read (Q)
- Create a node Ti → Tj if Ti executes read (Q) before Tj executes write (Q)
- Create a node Ti → Tj if Ti executes write (Q) before Tj executes write (Q)



- 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:

- Create a precedence graph of following schedule:



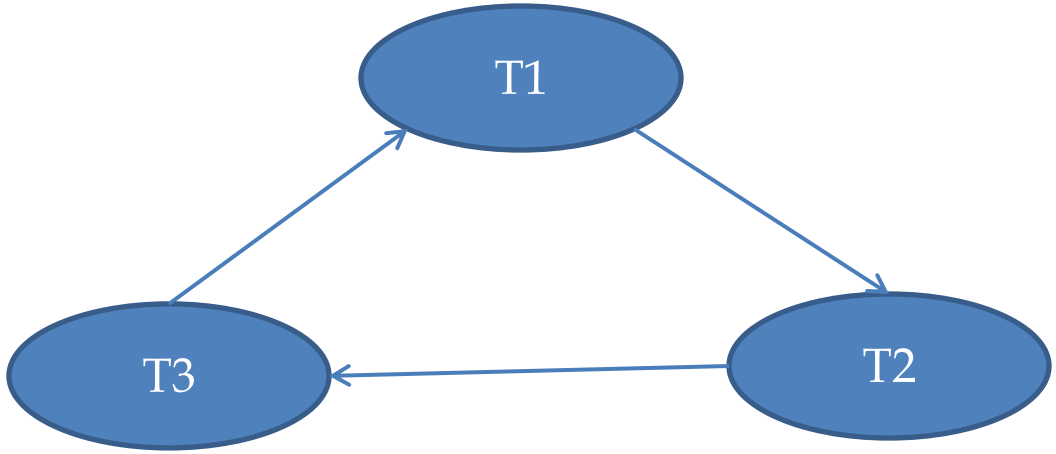
T1	T2	T3
Read(A)		
	Read(B)	
A:= f_1(A)		
		Read(C)
	B:= f_2(B)	
	Write(B)	
		C:= f_3(C)
		Write(C)
Write(A)		
		Read(B)
	Read(A)	
	A:= f_4(A)	
Read(C)		
	Write(A)	
C:= f_5(C)		
Write(C)		
		B:= f_6(B)
		Write(B)

Schedule S1

Explanation:

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

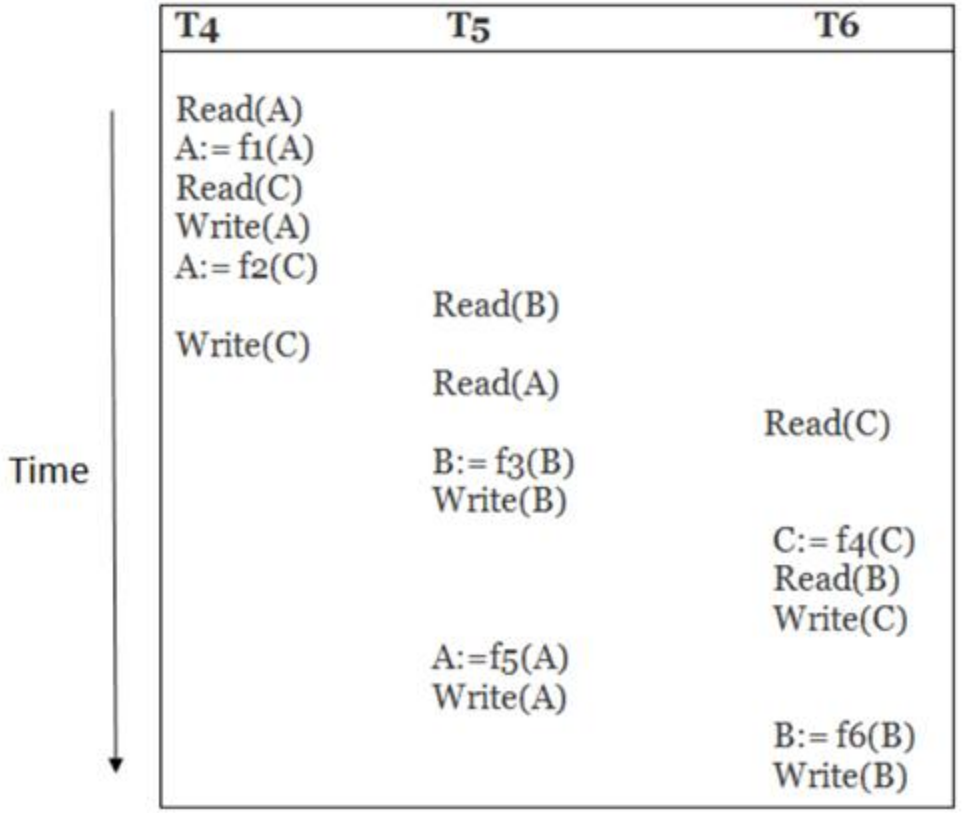
- Precedence graph of schedule S1 is:



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

Example:

- Create a precedence graph of following schedule:



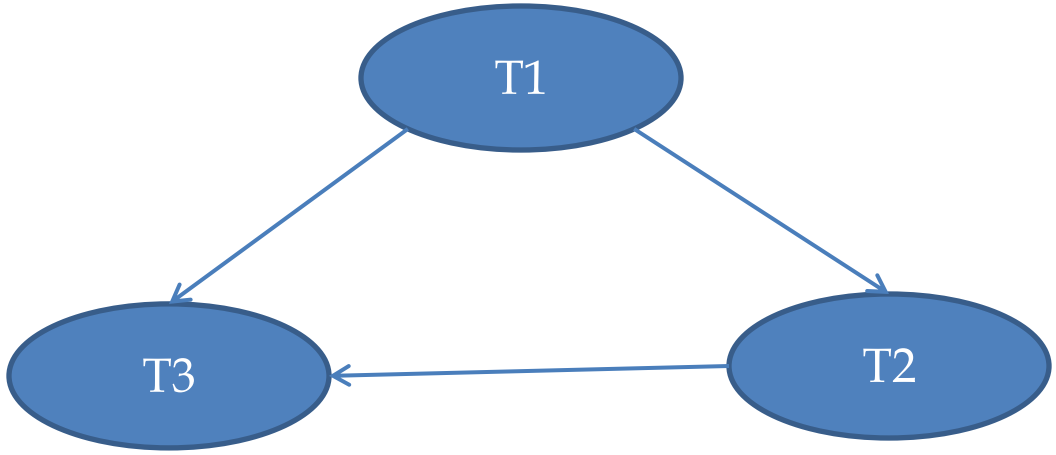
T4	T5	T6
Read(A)		
A:= f1(A)		
Read(C)		
Write(A)		
A:= f2(C)		
	Read(B)	
Write(C)		
	Read(A)	
		Read(C)
	B:= f3(B)	
	Write(B)	
		C:= f4(C)
		Read(B)
		Write(C)
	A:=f5(A)	
	Write(A)	
		B:= f6(B)
		Write(B)

Schedule S2

Explanation:

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

- Precedence graph of schedule S2 is:



The precedence graph for schedule S1 contains no cycle that's why Schedule S2 is ***serializable***

Conflict serializable schedule

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

Conflicting Operations

- The two operations become conflicting if all conditions satisfy:
  - Both belong to separate transactions
  - They have the same data item
  - They contain at least one write operation

Example:

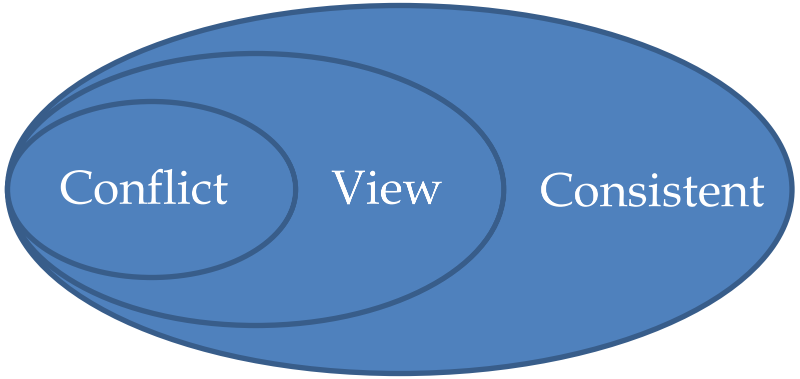
- Create a precedence graph and find the loops in it, if no loop exist then the schedule will be called as conflict serializable

T1	T2	T3
Read(x)		
		Read(y)
		Read(x)
	Read(y)	
	Read(z)	
		Write(y)
	Write(z)	
Read(z)		
Write(x)		
Write(z)		

View Serializablity

- If a non serial schedule is **view equivalent** of a serial schedule then that schedule will be called as **view serializable** schedule

View Serializablity



1. Initial read must be same in serial and non serial schedule
2. Final write must be same in serial and non serial schedule
3. Intermediate read must be same in serial and non serial schedule

Draw precedence graph for the following schedule and find out if the schedule is serializable or not?

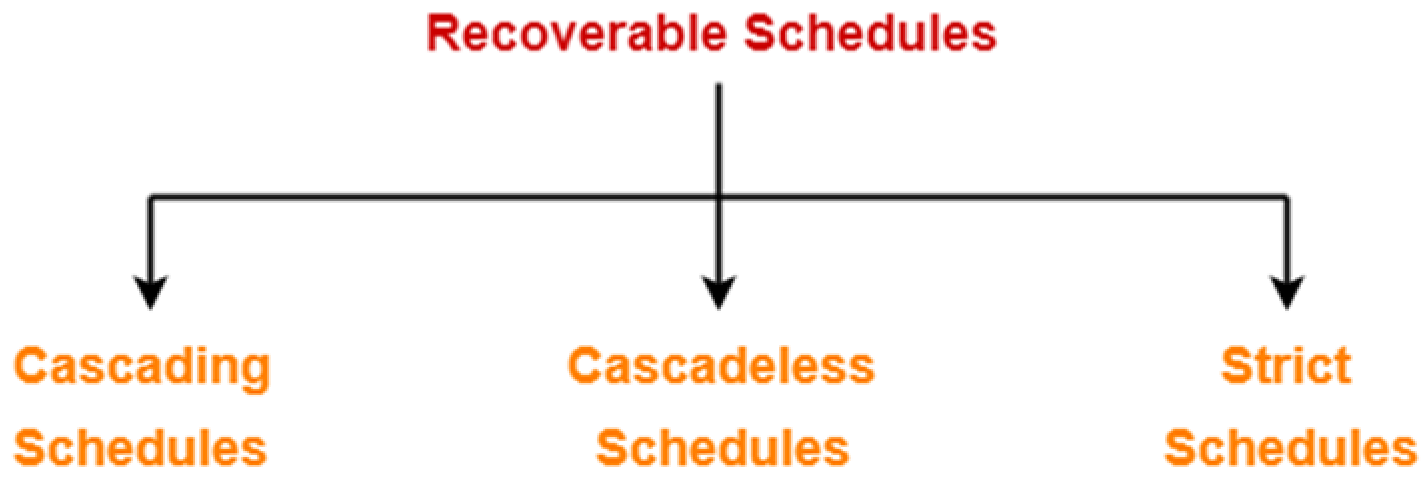
T1	T2	T3
Read(a)		
	Write(a)	
Write(a)		
		Write(a)

Recoverable VS Irrecoverable Schedule

- Find dirty read operations in the transactions
  - If exist then check for recoverability
  - Else it is recoverable

T1	**T2**
Read(a)	
a=a+10	
Write(a)	
	Read(a)
	a=a-10
	Write(a)
	Commit
Commit	

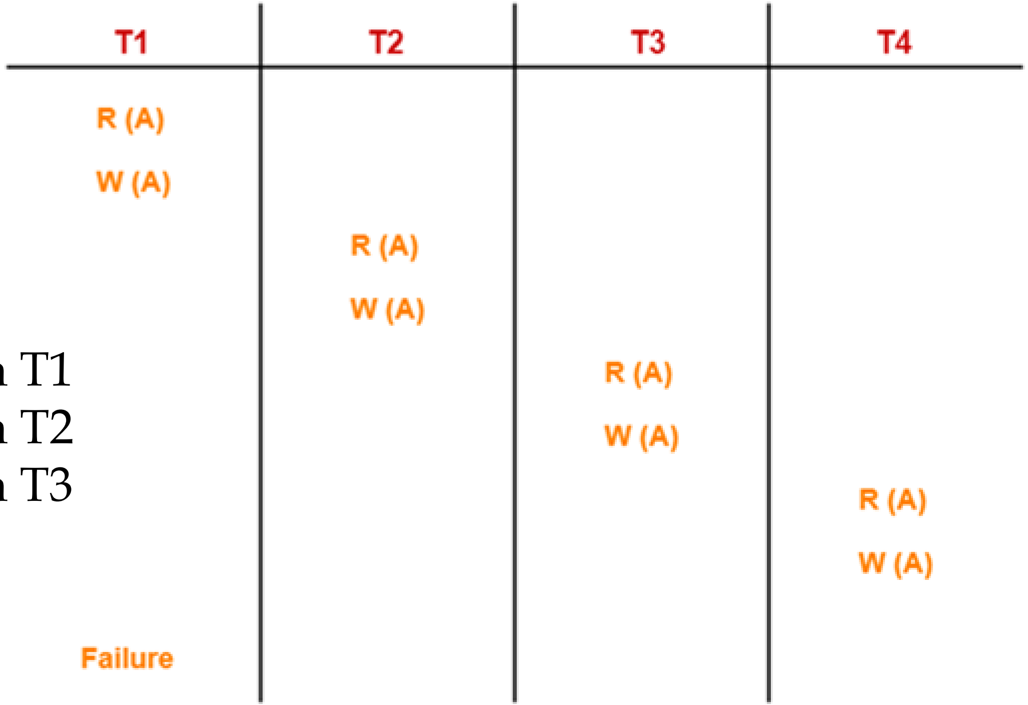
Types of recoverable schedules



Cascading Schedule

- If in a schedule, 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

Example:



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

Cascading Recoverable Schedule

Here,

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

Note: If the transactions T2, T3 and T4 would have **committed before** the failure of transaction T1, then the schedule would have **been irrecoverable**

Cascadeless schedule

- **If in a schedule, a transaction is not allowed to read a data item until the last transaction that has written it is committed or aborted,** then such a schedule is called as a Cascadeless Schedule
  - In other words, Cascadeless schedule allows only committed read operations
  - Therefore, it avoids cascading roll back and thus saves CPU time

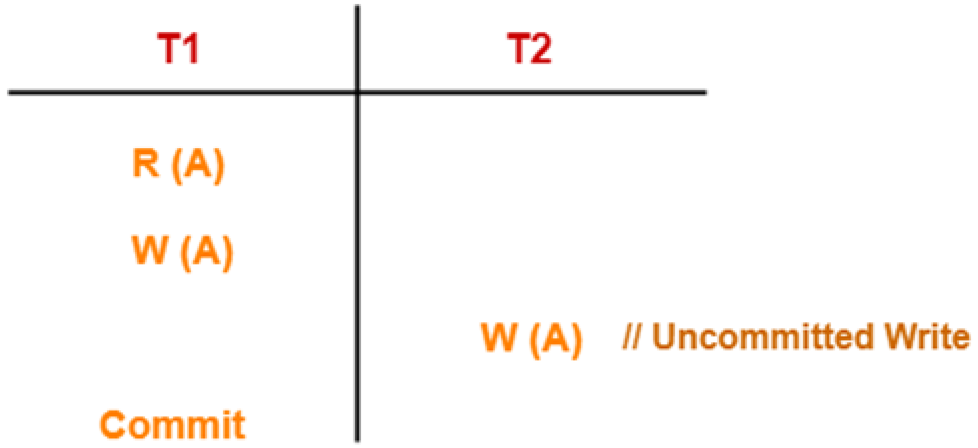
Example:

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

Cascadeless Schedule

- Cascadeless schedule allows only committed read operations.
- However, it allows uncommitted write (blind write) operations.

Example:



T1	T2
R (A)	
W (A)	
	W (A) // Uncommitted Write
Commit	

Cascadeless Schedule

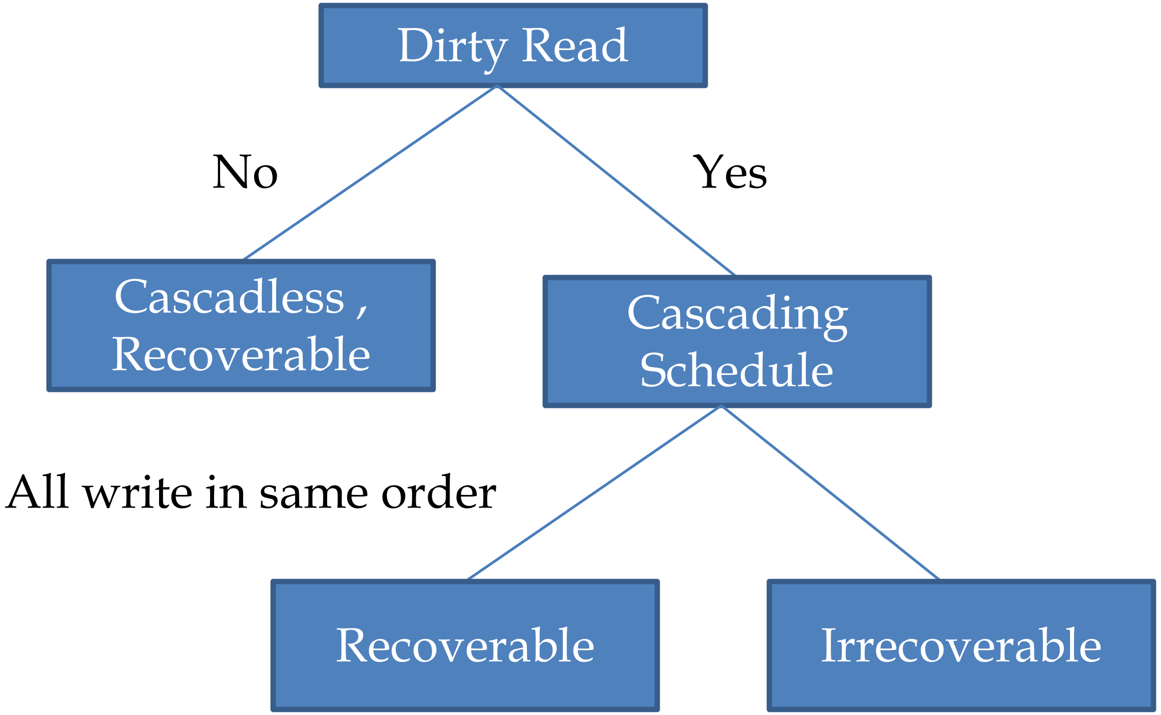
Strict Schedules

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

Example:

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

Strict Schedule


Dirty Read
No
Yes
Cascadless , Recoverable
Cascading Schedule
All write in same order
Recoverable
Irrecoverable