Database Transactions

Dr. Seema Gupta Bhol

Database Transaction

Database Transaction is An executing program (process) that includes one or more database access operations

- Read operations (database retrieval, such as SQL SELECT)
- Write operations (modify database, such as SQL INSERT, UPDATE, DELETE)

Transaction: A logical unit of database processing

- Example: Bank balance transfer of $50 dollars from a checking account to a saving account in a BANK database

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Example of Fund Transfer

- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

- **Consistency requirement** in above example: The sum of A and B is unchanged by the execution of the transaction
  - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	T2
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A), read(B), print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B	

- Isolation can be ensured trivially by running transactions **serially ,**That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

ACID properties of Transactions

Atomicity, Consistency, Isolation, Durability:

- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency preservation**: A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation**: Even though transactions are executing concurrently, they should appear to be executed in isolation – that is, their final effect should be as if each transaction was executed in isolation from start to finish.
- **Durability or permanency**: Once a transaction is committed, its changes (writes) applied to the database must never be lost because of subsequent failure.

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. 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.
  - That is, 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.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Commit Point of a Transaction

A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log file (on disk).

The transaction is then said to be **committed**.

When Program execute COMMIT;

- SQL command to finish the transaction successfully
- The next SQL statement will automatically start a new transaction

ROLLBACK

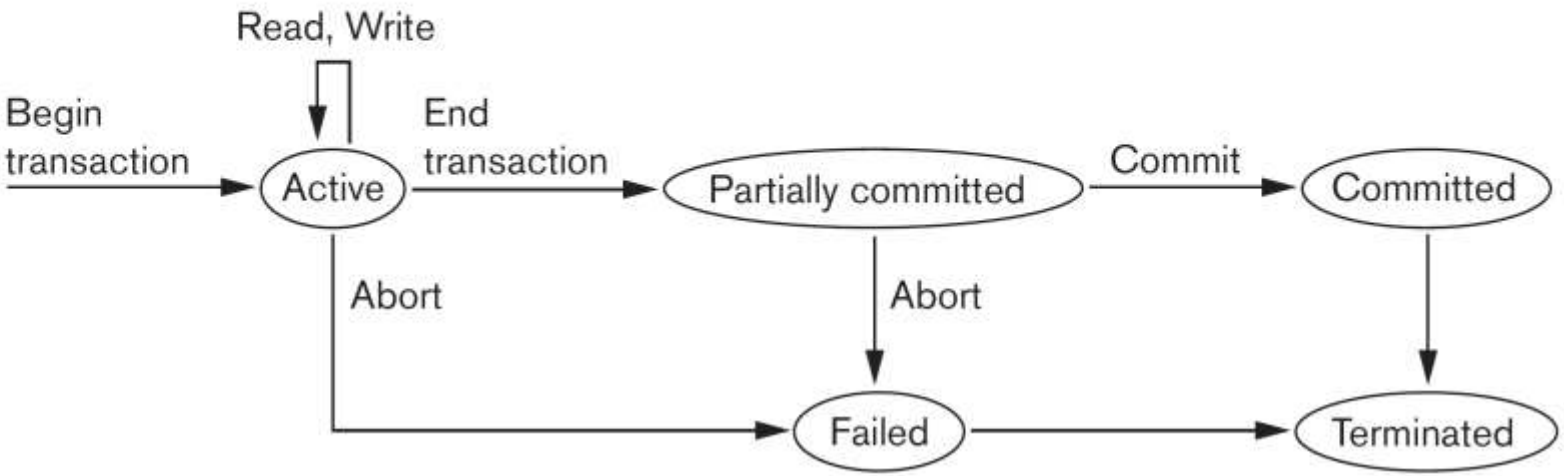
- If the program reaches to a place where it can’t complete the transaction successfully, it can execute ROLLBACK
- This causes the system to “abort” the transaction
  - The database returns to the state without any of the previous changes made by activity of the transaction

Transaction States

- **Active** – the initial state; the transaction stays in this state while it is executing(executing read, write operations)
- **Partially committed** – after the final statement has been executed (ended but waiting for system checks to determine success or failure).
- **Failed --** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
  - Kill the transaction
- **Committed** – after successful completion(transaction succeeded)

.

Transaction States

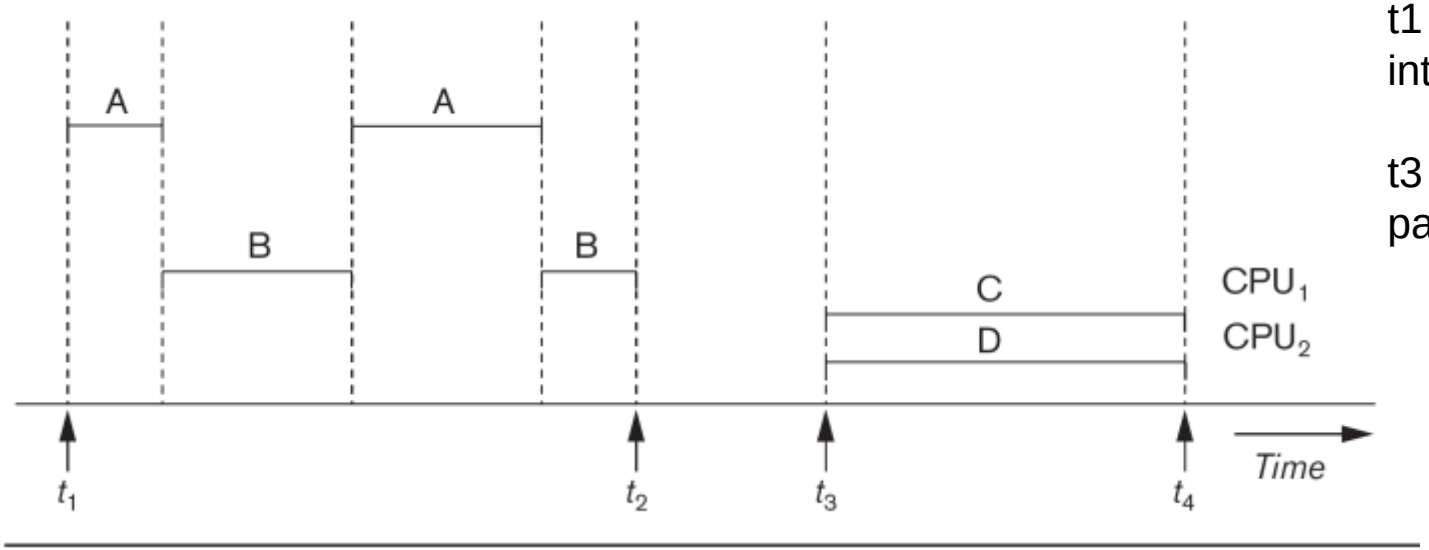


Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – Mechanisms to achieve isolation i.e. to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

Two Modes of Concurrency

- **Interleaved processing**: concurrent execution of processes is interleaved in a single CPU
- **Parallel processing**: processes are concurrently executed in multiple CPUs .
- Basic transaction processing theory assumes interleaved concurrency



t1 and t2 are interleaved.

t3 and t4 are parallel processing

Interleaved Processing and Parallel Processing

- **Interleaved Processing –**
  The concurrent execution of processes is interleaved in a single CPU. The transactions are interleaved, meaning the second transaction is started before the primary one could finish. And execution can switch between the transactions. It can also switch between multiple transactions. This causes inconsistency in the system.
- **Parallel Processing –**
  It is defined as the processing in which a large task into various smaller tasks and smaller task also executes concurrently on several nodes. In this, the processes are concurrently executed in multiple CPUs.

Schedules of Transactions

- When transactions are executing concurrently in an interleaved fashion, the *order of execution* of operations from the various transactions forms what is known as a **transaction schedule**.

- Formal definition of a **schedule** (or **history**) S of n transactions T1, T2, ..., Tn :

  An ordering of all the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of Ti in S must appear *in the same order* in which they occur in Ti.

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement .

Schedule 1

- Let T_1 transfer $50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2 :

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

Schedule 2

- A serial schedule where T_2 is followed by T_1

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

Schedule 3

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

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)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

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

Schedule 4

- The following concurrent schedule does not preserve the value of (*A* + *B*).

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)	
commit	
	$B := B + temp$
	write (B)
	commit

Serial schedule

- A schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively (without interleaving of operations from other transactions) in the schedule. Otherwise, the schedule is called **nonserial.**
- Serial schedules are *not feasible* for performance reasons:
  - No interleaving of operations
  - Long transactions force other transactions to wait
  - System cannot switch to other transaction when a transaction is waiting for disk I/O or any other event
  - Need to allow concurrency with interleaving without sacrificing correctness

Equivalence of Schedules

- A schedule S is **serializable** if it is **equivalent** to some serial schedule of the same n transactions.
- There are (n)! serial schedules for n transactions
- A serializable schedule can be equivalent to *any of the serial schedules*
- Two schedules are called equivalent if they produce the same final state of the database.
- Difficult to determine without *analyzing the internal operations of the transactions*, which is not feasible in general.

Serializability

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

  1. **Conflict serializability**
  2. **View serializability**

Conflict equivalent

- Two schedules are conflict equivalent if the relative order of *any two conflicting operations* is the same in both schedules.
- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item X accessed by both l_i and l_j, and at least one of these instructions wrote X.
- Two operations are **conflicting** if:
    - They access the same data item X
    - They are from two different transactions
    - At least one is a write operation
- Read-Write conflict example: r1(X) and w2(X)
- Write-write conflict example: w1(Y) and w2(Y)

Conflicting Instructions (example)

1. l_i = **read**(X), l_j = **read**(X). l_i and l_j don't conflict.
2. l_i = **read**(X), l_j = **write**(X). They conflict
3. l_i = **write**(X), l_j = **read**(X). They conflict
4. l_i = **write**(X), l_j = **write**(X). They conflict

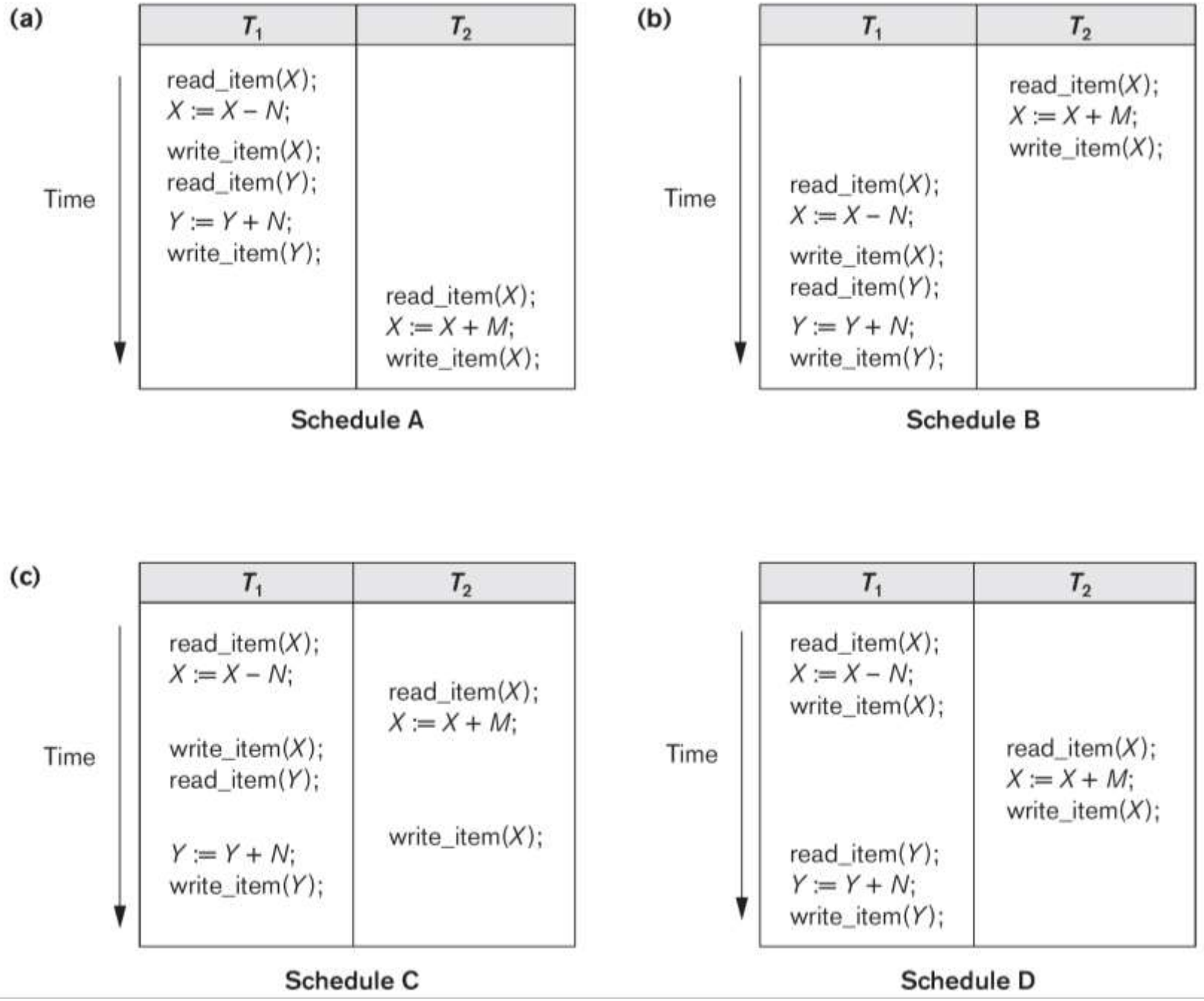
- Thus, a conflict between l_i and l_j forces a temporal order between them.
- If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

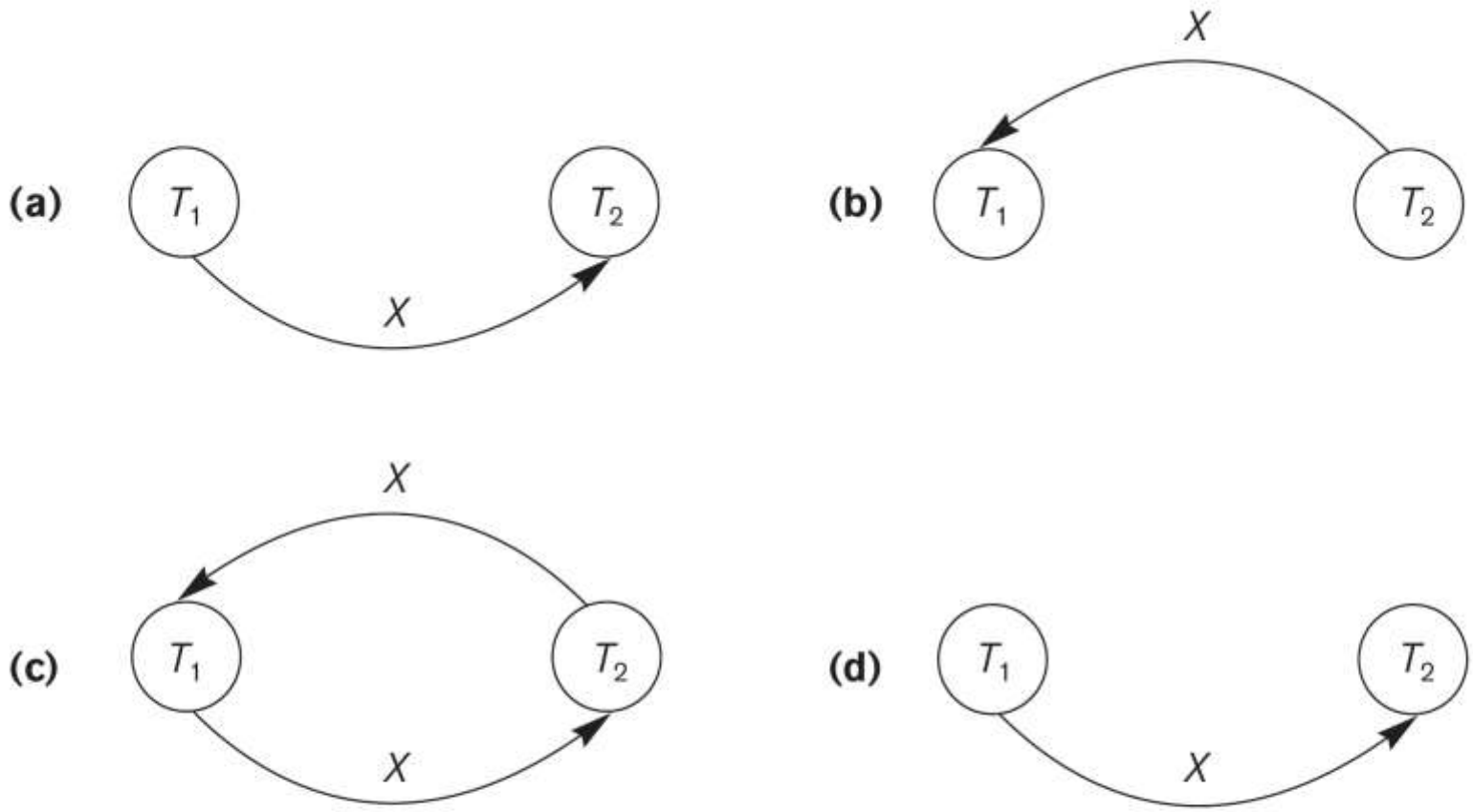
Testing for conflict serializability

A schedule S is said to be **serializable** if it is conflict equivalent to some serial schedule S'.

Algorithm

- Looks at only r(X) and w(X) operations in a schedule
- Constructs a precedence graph (serialization graph) – **one node for each transaction**, plus directed edges
- An **edge is created** from T_i to T_j if one of the operations in T_i appears before a conflicting operation in T_j
- The schedule is serializable if and only if the precedence graph **has no cycles**.


(a)
T1
T2
read_item(X);
X := X − N;
write_item(X);
read_item(Y);
Y := Y + N;
write_item(Y);
read_item(X);
X := X + M;
write_item(X);
Time
Schedule A
(b)
T1
T2
read_item(X);
X := X − N;
write_item(X);
read_item(Y);
Y := Y + N;
write_item(Y);
read_item(X);
X := X + M;
write_item(X);
Time
Schedule B
(c)
T1
T2
read_item(X);
X := X − N;
write_item(X);
read_item(Y);
Y := Y + N;
write_item(Y);
read_item(X);
X := X + M;
write_item(X);
Time
Schedule C
(d)
T1
T2
read_item(X);
X := X − N;
write_item(X);
read_item(Y);
Y := Y + N;
write_item(Y);
read_item(X);
X := X + M;
write_item(X);
Time
Schedule D


(a)
T_1
T_2
X
(b)
X
T_1
T_2
(c)
X
T_1
T_2
X
(d)
T_1
T_2
X

View serializability

- **View equivalence**: A less restrictive definition of equivalence of schedules than conflict serializability *when blind writes are allowed*

- **View serializability:** definition of serializability based on view equivalence. A schedule is *view serializable* if it is *view equivalent* to a serial schedule.

Blind write

- Blind write: Performing the Writing operation (updation), without reading operation, a such write operation is known as a blind write.
- If no blind write exists, then the schedule must be a non-View-Serializable schedule.
- If there exists any blind write, then, in that case, the schedule may or may not be view serializable.
- Then, draw a precedence graph using those dependencies. If no cycle/loop exists in the graph, then the schedule would be a View-Serializable otherwise not.

View Serializability (Cont.)

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- Every view serializable schedule that is not conflict serializable has **blind writes.**

View Equivalent

The premise behind view equivalence:

- Each read operation of a transaction reads the result of *the same write operation* in both schedules.
- **"The view"**: the read operations are said to see the *same view* in both schedules.
- The final write operation on each item is the same on both schedules resulting in the same final database state in case of blind writes

View Serializability

- Let *S* and *S'* be two schedules with the same set of transactions. *S* and *S'* are **view equivalent** if the following three conditions are met, for each data item *Q*,
  1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q*.
  2. If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'*.
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

Problem: Prove whether the given schedule is View-Serializable or not

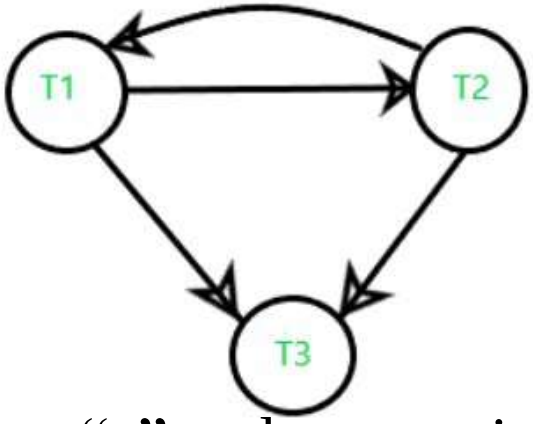
S' : read1(A), write2(A), read3(A), write1(A), write3(A)

First of all we'll make a table for a better understanding of given transactions of **schedule S'**

T_1	T_2	T_3
read(a)		
	write(a)	
		read(a)
write(a)		
		write(a)

- First, we check whether it is Conflict-Serializable or not, because if it is Conflict-Serializable so it will also be View-Serializable, so we will make a precedence graph for the **schedule S'.**
- Here we will check whether the Schedule s' contains any blind write. We found that the schedule s' contains a blind-write write2(a) in transaction T2. Hence schedule S' may or may not be View-Serializable. So we will look at another method. Because, if it does not contain any Blind-write, we can surely state that the schedule would not be View-Serializable.
- Now, we will draw a dependency graph that is different from the precedence graph.

- Its Dependency graph will be :



- Transaction $\mathbf{T_1}$ first reads data_item "a" and transaction $\mathbf{T_2}$ first updates(write) **"a".**
- So, the transaction $\mathbf{T_1}$ must execute before $\mathbf{T_2}$**.**
- In that way, we get the dependency $\mathbf{(T_1 \rightarrow T_2)}$ in the graph.
- And, the final update(write) on **"a"** is made by transaction $\mathbf{T_3}$**.**
- So, transaction $\mathbf{T_3}$ must execute after all the other **transactions($\mathbf{T_1}$, $\mathbf{T_2}$).**
- Thus, we get the dependency $\mathbf{(T_1, T_2)} \rightarrow \mathbf{T_3}$ in the graph.
- As there is a cycle/loop in the precedence graph, the schedule s' is not in View-Serializable.

View Equivalent

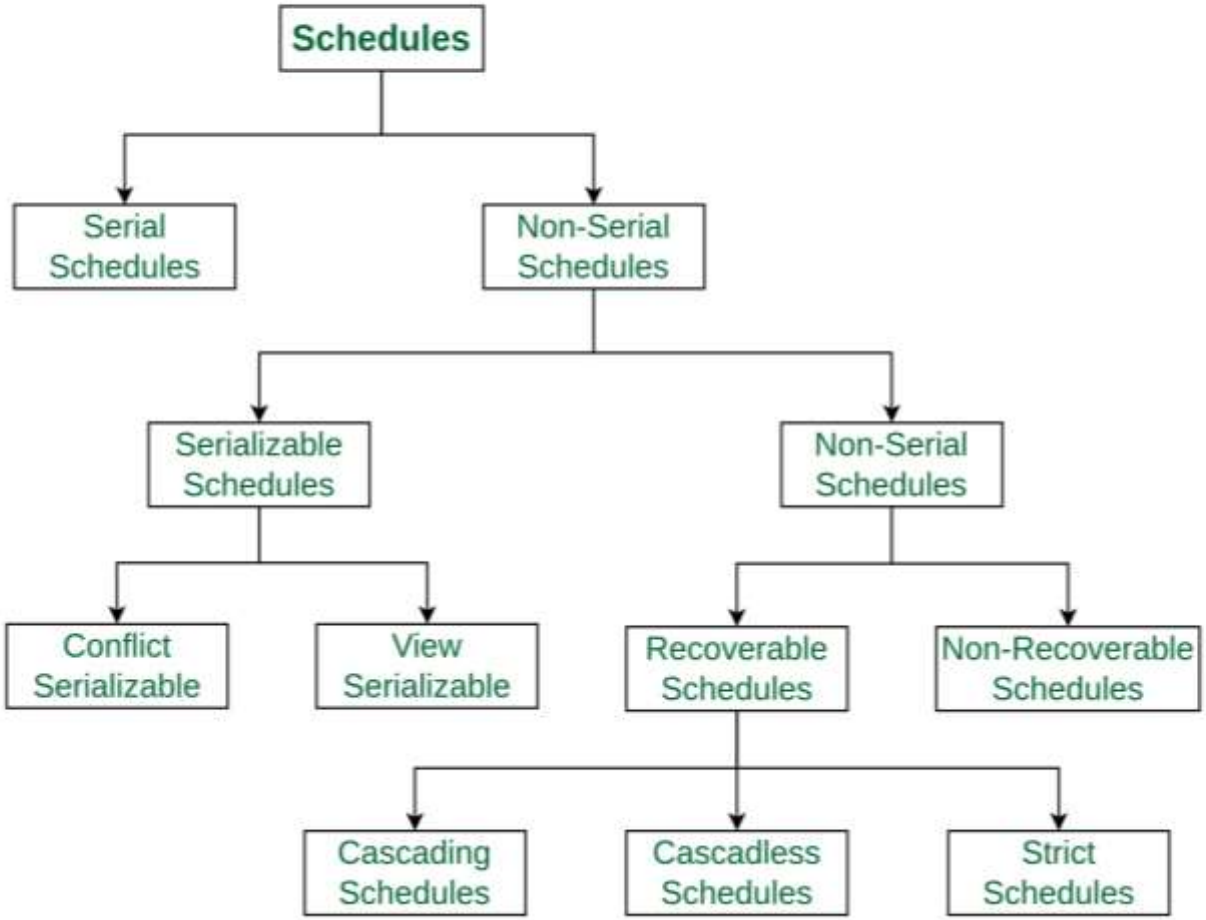
Relationship between view and conflict equivalence:

- The two are same under **constrained write assumption** (no blind writes allowed)
- Conflict serializability is **stricter** than view serializability when **blind writes occur** (a schedule that is view serializable is not necessarily conflict serialiable.
- Any conflict serializable schedule is also view serializable, but not vice versa.

Difference between Serial Schedule and Serializable Schedule

Serial Schedule	Serializable Schedule
In Serial schedule, transactions will be executed one after other.	In Serializable schedule transaction are executed concurrently.
Serial schedule are less efficient.	Serializable schedule are more efficient.
In serial schedule only one transaction executed at a time.	In Serializable schedule multiple transactions can be executed at a time.
Serial schedule takes more time for execution.	In Serializable schedule execution is fast.

Types of schedules in DBMS



Characterizing Schedules based on Recoverability

Need to address the effect of transaction failures on concurrently running transactions.

Schedules classified into two main classes:

- **Recoverable schedule:** One where no *committed* transaction needs to be rolled back (aborted).

  A schedule S is **recoverable** if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

- **Non-recoverable schedule:** A schedule where a committed transaction may have to be rolled back during recovery.

  This violates **Durability** from ACID properties (a committed transaction cannot be rolled back) and so non-recoverable schedules *should not be allowed*.

Recoverable Schedule

- A schedule is recoverable if it allows for the recovery of the database to a consistent state after a transaction failure.
- In a recoverable schedule, a transaction that has updated the database must commit before any other transaction reads or writes the same data.
- If a transaction fails before committing, its updates must be rolled back, and any transactions that have read its uncommitted data must also be rolled back.

Recoverable Schedules

- If a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule is not recoverable

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- If T_8 should abort, T_9 would have read an inconsistent database state. Hence, database must ensure that schedules are recoverable.

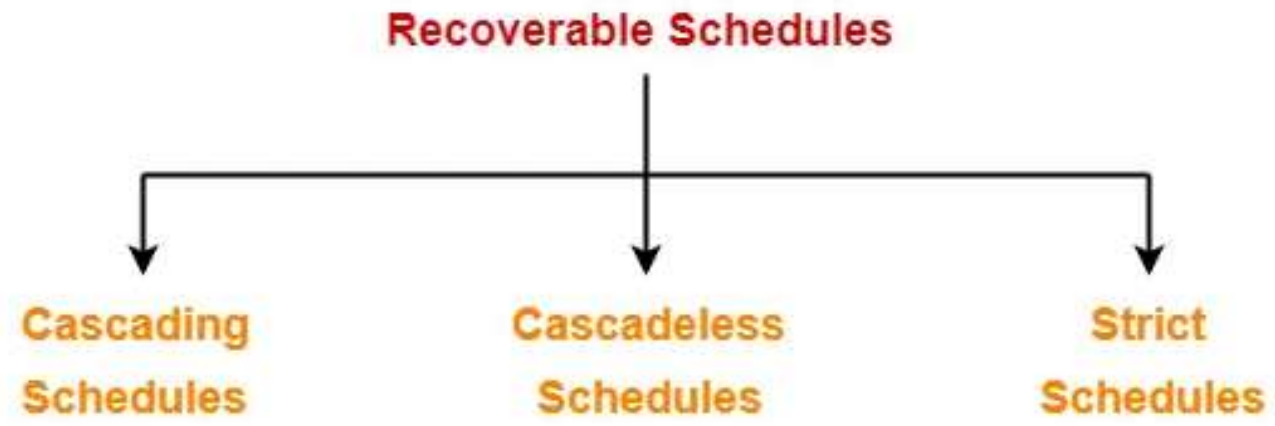
Types of Schedules based Recoverability

There are three types of Schedules based on the Recoverability

- **Recoverable Schedule:** A schedule is recoverable if it allows for the recovery of the database to a consistent state after a transaction failure
- **Cascadeless Schedule:** Recoverable schedules can be further refined as Cascadeless Schedule .A schedule is cascaded less if it does not result in a cascading rollback of transactions after a failure.
- **Strict Schedule:** Cascadeless schedules can be further refined Strict Schedule A schedule is strict if it is both recoverable and cascades.

Types of Recoverable Schedules

A recoverable schedule may be any one of these kinds-



1. Cascading Schedule
2. Cascadeless Schedule
3. Strict Schedule

Cascading Schedules

- 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.

Cascading Schedules

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

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

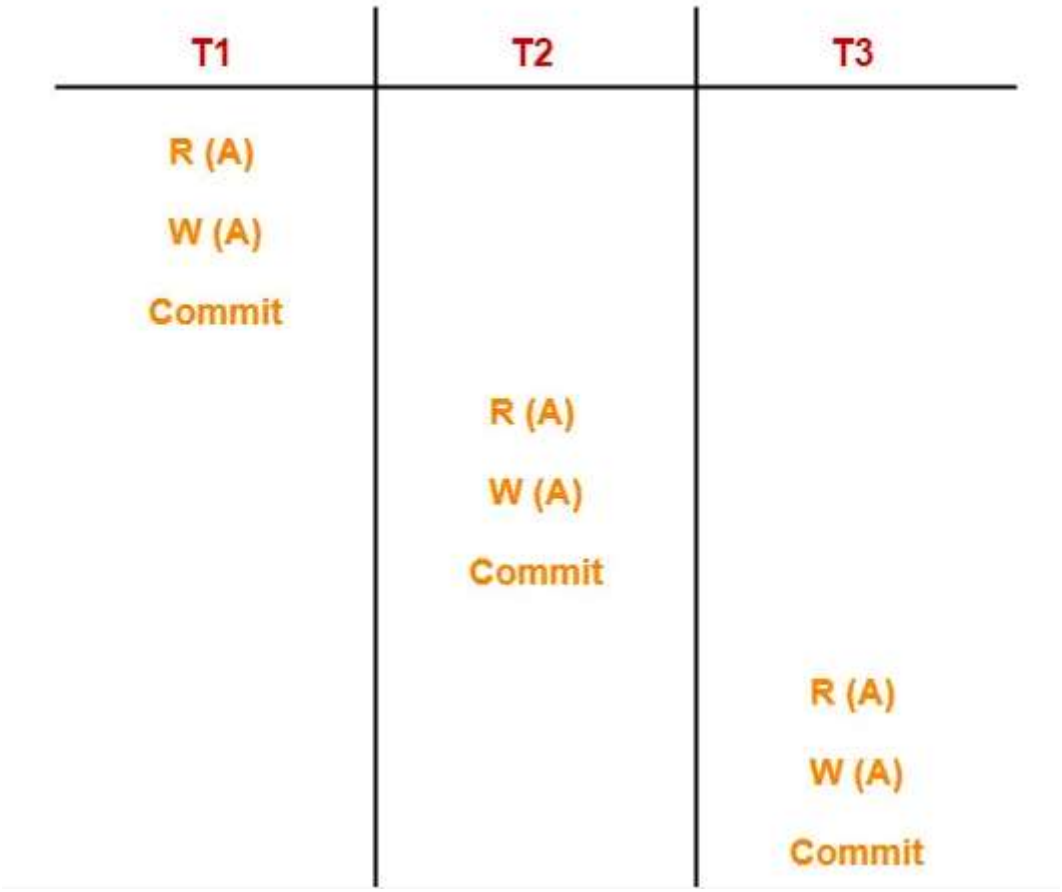
If T_{10} fails, T_{11} and T_{12} must also be rolled back. This is Known as Cascading Rollbacks

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

Cascade-less Schedule

- A schedule is cascaded less if it does not result in a cascading rollback of transactions after a failure.
- 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.
- A schedule is cascaded less if it does not result in a cascading rollback of transactions after a failure.
- If a transaction fails before committing, its updates must be rolled back, but any transactions that have read its uncommitted data need not be rolled back.
- Every Cascadeless schedule is also recoverable

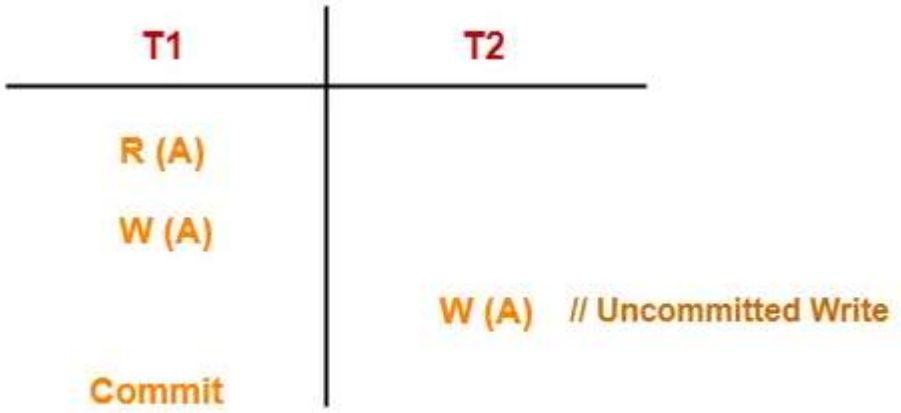
Cascade-less Schedule



Cascade-less Schedule

- 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.
- In a cascade-less schedule, a transaction that has read uncommitted data from another transaction cannot commit before that transaction commits.
- In other words:
  - Cascadeless schedule allows only committed read operations.
  - However, it allows uncommitted write operations.

Cascade-less Schedule

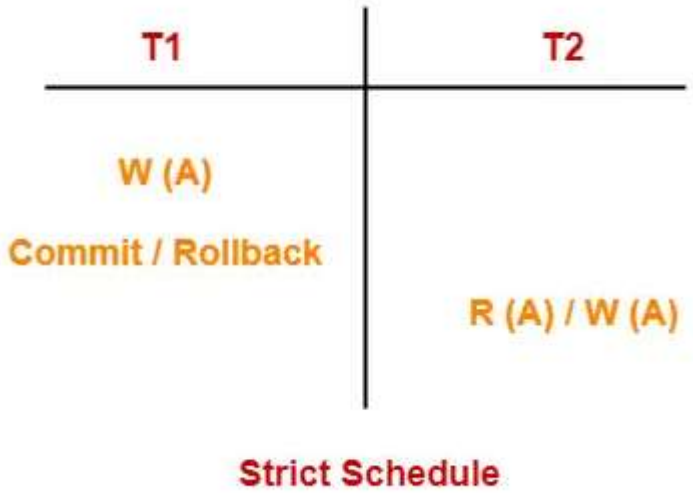


Cascadeless Schedule

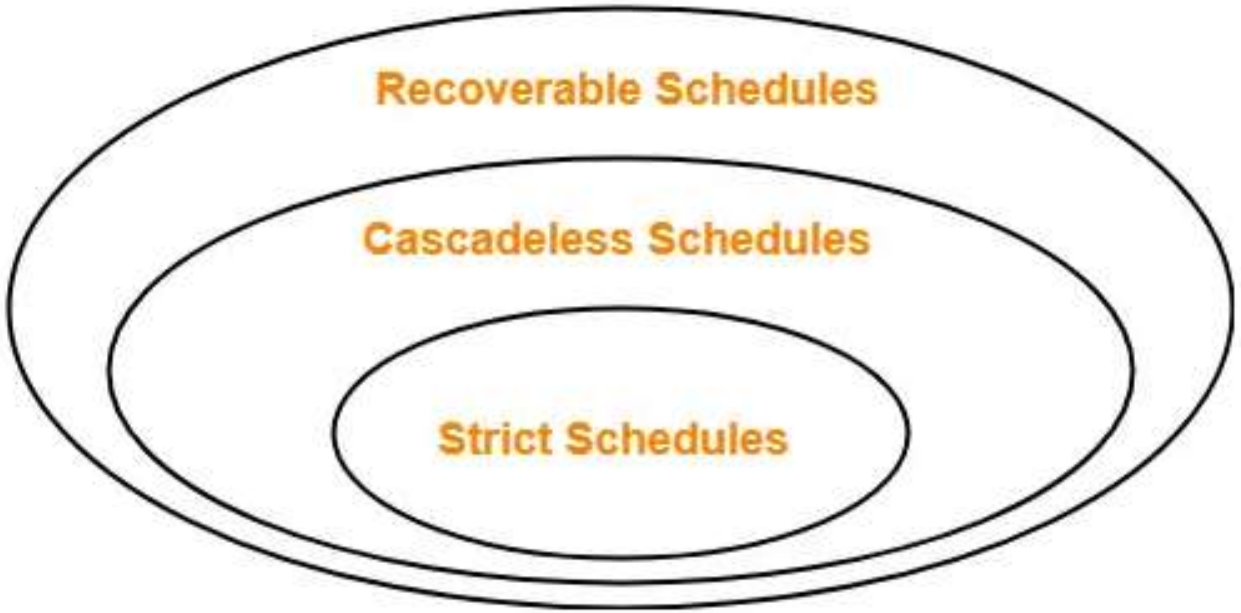
Strict Schedule

- 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.
- A schedule in which a transaction T2 can neither read *nor write* an item X until the transaction T1 that last wrote X has committed.
- In other words, Strict schedule allows only committed read and write operations.
- Clearly, strict schedule implements more restrictions than cascadeless schedule.

Strict Schedule



- Strict schedules are more strict than cascadeless schedules.
- All strict schedules are cascadeless schedules.
- All cascadeless schedules are not strict schedules.



Different recoverable schedules

Cascading Schedule	Casecadless Schedule	Strict Schedule
Cascading allows READ or WRITE operation for dependent Transaction (T2) before to T1 committed or abort.	Casecadless Don't allow READ to dependent Transaction (T2) untill T1 committed or abort. But it allows WRITE operation for dependent Transaction (T2) before to T1 committed.	Strict don't allow READ or WRITE operation to dependent Transaction (T2) until T1 committed or abort.
Note: Rollback of one Transaction leads to rollback of all other dependent Transactions Note: This case is problematic.	Note: This case is also problematic.	Note: This case if 100% safe.