## Transaction

A transaction is a unit of program execution that accesses and possibly updates various data items. It is written in high level languages C, C++, Java with embedded database accesses.

## Purpose of transaction

It is considered as a task, consist of multiple subtasks. for eg Transfering balance from one account to another

$A := A - 50$ , $B = B + 50$

The pattern of transaction is

- Begin Transaction
- ---
- End Transaction

(i) Begin the transaction
(ii) Execute DML
(iii) If no, error commit
(iv) If error occur, then rollback.

## Properties of Transaction ACID

(i) **Atomicity**: Either all operations of the transactions are reflected properly in the DB or none are.

(ii) **Consistency**: Execution of a transaction in isolation preserves the consistency of the DB. It ensure DB remain in valid state, mean it will not violate Business rule

(iii) **Isolation**: It guarantees that for every pair of transaction $T_i$ & $T_j$ it appear to $T_i$ that either $T_j$ finished execution before $T_i$ started or $T_j$ started execution after $T_i$ finish.

Durability : The changes made to the DB persist even, if there are system failures

Ex:

Atomicity: Suppose there are two account A & B with balance 1000 & 2000.

```
T1  read(A);
    A: A - 50;
    write(A);
    Read(B);
    B: = B + 50;
    write(B);
```

read(X): transfer data item X from DB to a variable

write(X): transfer the value in the variable X to the DB

Now suppose a failure happened after write(A), but before write(B). Now A = 950, B = 2000. The sum of A+B is 950 + 2000 = 2950 which is no longer preserved. The state is called inconsistent state.

Consistency : It require that the sum of A & B be unchanged by the execution of the transaction. It is the responsibility of programme to achieve the consistency.

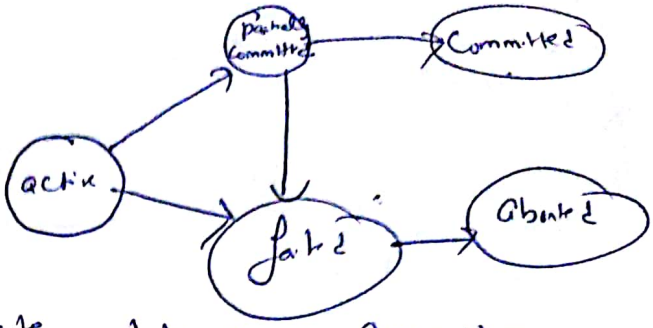
Durability : Updates carried out by the transaction have been written to disk before transaction completes. Log of transaction written to disk are sufficient to reconstruct the updates.

Isolation If another transaction reads A & B intermediately then it will obtain an inconsistent state of DB.

## Storage Structure

(i) Volatile Storage
(ii) Nonvolatile Storage
(iii) Stable Storage

## Transaction States




**Active**, 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.

**Aborted**, after the transaction has been rolled back & the DB has been restored to its state prior to the start of transaction.

**Committed**: after successful completion.

## SYSTEM LOG

To be able to recover from failures that affect transactions the system maintains a log to keep track of all transactions operations that affect the value of DB item.

It is sequential, append only file that is kept on the disk

[Start_transaction, T] Tid

[write_item, T, X, old_val, new_val]

[read_item, T, X]

[commit, T]

[abort, T]

A transaction enter the failed state after the System determine that the transaction can no longer proceed with its normal execution for eg h/w error, logical error) It must be rolled back. then it enter the aborted state. Now it has 2 option — (i) Restart transaction.

(ii) Kill transaction (Internal logic. Person)

Observable External writes for eg sending Email, money to user.

Transaction of dispensing cash at ATM, system fails just before the cash is actually dispensed. Now compensating transaction, such as depositing the cash back into user's account needs to be execute

## Concurrency

It is the process of managing/controlling simultaneous operations on the DB.

## Advantages of concurrency

(i) Reduced waiting time
(ii) Reduced ~~Big problem~~ Response time
(iii) Resource Utilization
(iv) Efficiency.

If concurrent transaction are execute w/o any concurrency control algorithm then following problem may arise :-

### (i) Dirty Read Problem (write – Read)

It occurs when one transaction out of multiple transaction is allowed to see the intermediate result of another transaction before it is committed.

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | C |
| Failure | |

### (ii) UnRepeatable Read problem (Read – Write)

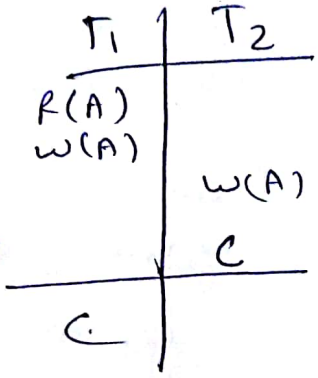
$T_2$ thinks, it is isolated, But it is getting two different value of X.

X = 10

| | $T_1$ | $T_2$ | |
|---|---|---|---|
| 10 | R(X) | | |
| | : | R(X) | 10 |
| 15 | W(X) | | |
| | | R(X) | 15 |

(iii) Lost update Problem (Write - Write)

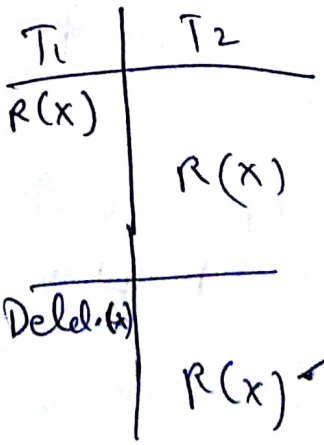
It occurs when update operation by one user can be overridden by another user. When 2 or more transaction select the same row and then update the row based on the value originally selected. The last update overwrites the updates made by the other transaction.




A = 10

It is also called Blind write (W/o Read)

(iv) Phantom Read



## Scheduling of Transactions

A schedule is a sequence of the operations by the set of concurrent transactions that preserves the order of the operations in each of the individual transactions. Suppose there are 2 transactions.

arranged in chronological order

| $T_1$ | $T_2$ |
|---|---|
| Read(A)<br>A = A - 50<br>Write(A)<br>Read(B)<br>B = B + 50<br>Write(B) | Read(A);<br>temp = A × 0.1;<br>A = A - temp;<br>Write(A);<br>Read(B);<br>B = B + temp;<br>Write(B); |

When transaction are running concurrently then their operation may conflict with each other if they satisfy following conditions :

→ The operation must belong to different transaction

→ operation affect the same data item.

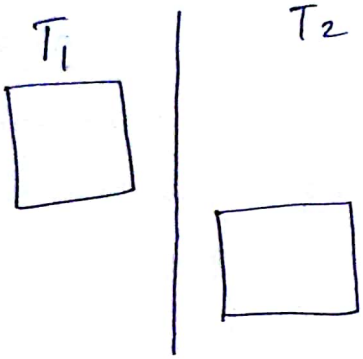
→ Atleast one of the operation must be write

## Type of Schedule

(i) Serial : Transaction are executed non interleaved
$T_1$ execute Before $T_2$ or
$T_2$ " " $T_1$.

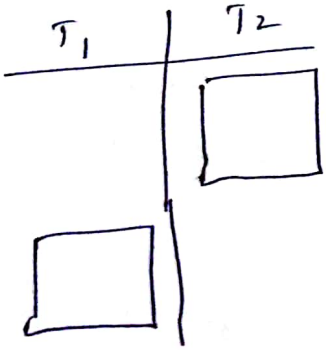
(ii) Non Serial

When operation of transaction are interleaved & interfering with each other.




Schedule 1

$T_1$ followed by $T_2$




Schedule 2

$T_2$ followed by $T_1$

Possible Schedule are $n!$

| | $T_1$ | $T_2$ |
|---|---|---|
| | 1000 R(A) | |
| A = 1000 | 950 A = A - 50 | |
| B = 2000 | 950 W(A) | |
| | | R(A) 950 |
| | | temp = A * 0.1 |
| | | A = A - temp 855 |
| | | W(A) |
| | 2000 R(B) | |
| | 2050 B = B + 50 | |
| | 2050 W(B) | |
| | 2050 Commite | |
| | | R(B) 2050 |
| | | B = B + temp 2050 + 95 |
| | | W(B) = 2145 |
| | | commit |

Non Serial Schedu-3

Sum A + B = in consistent

not all the Non-Serial Schedule are resulting into correct state. for eg

A = 1000
B = 2000

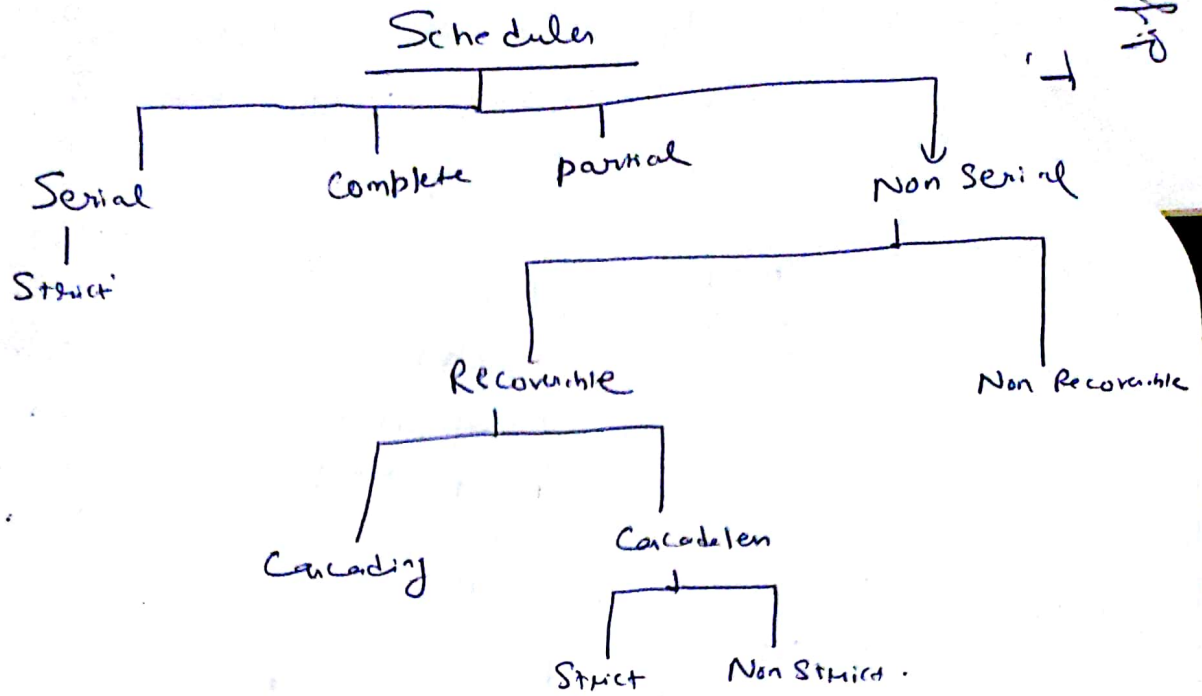
| $T_1$ | $T_2$ |
|---|---|
| 1000 R(A) | |
| 950 A = A - 50 | |
| | R(A) 1000 |
| | temp = A * 0.1 |
| | A = A - temp 900 |
| | W(A) 900 |
| | R(B) 2000 |
| 950 W(A) | |
| 2000 R(B) | |
| 2050 B = B + 50 | |
| 2050 W(B) | |
| commit | |
| | B = B + temp 2000 + 100 |
| | W(B) 2100 |
| | commit |

Concurrent Schedule lead to inconsistent State.

final value of A & B after committing both transaction are 950 and 2100 & sum A + B is not consistent.

## Serializability

A concurrency control protocol is used to schedule transaction in such a way that it used to prevent inconsistency. We can ensure consistency of the database by making sure that any schedules that is executed has the same effect as a schedule that could have occurred without any concurrent execution. That is schedule is equivalent to a serial schedule.




## Type of Serializable Schedule

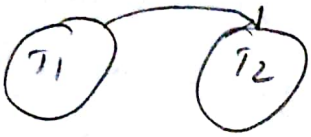
(i) Equivalent Schedule
(ii) Conflict serializability
(iii) View "

Consider Two transaction $T_1$ and $T_2$ & containing instr. I and J in Schedule 'S' corresponding to $T_1$ & $T_2$. If I & J refer to different data item then order doesn't matter

1. I = Read(Q) J = R(Q) : Order doesn't matter

2. I = R(Q), J = W(Q) : Order matter
3. I = W(Q), J = R(Q) : Order matter
4. I = W(Q), J = W(Q) : Result of only the latter instruction is preserved.

I and J conflict if they are operation by different Transactions on the same data item and atleast one of them is write operation.




| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| R(B) | |
| W(B) | |
| | R(B) |
| | W(B) |

W(A) of $T_1$ is in conflict with R(A) of $T_2$

W(A) of $T_2$ doesn't conflict with read(B)

If I & J don't conflict then we can swap the order of I & J to produce new schedule S'. S is equivalent to S'. So the equivalent schedule of above schedule is obtained by interchanging the non-conflicting operation.

Conflict Equivalent

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| R(B) | |
| | W(A) |
| W(B) | |
| | R(B) |
| | W(B) |

Non conflict, it can be swap

Non conflicting, it can be swap

If a schedule, S can be transformed into a schedule S' by a series of swaps of non-conflicting instruction, we say that S and S' conflict equivalent.

## Conflict Serializable

A Schedule is conflict serializable if it is conflict equivalent to a serial schedule.

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| R(B) | |
| W(B) | |
| | R(A) |
| | W(A) |
| | R(B) |
| | W(B) |

It is obtained by performing continuous swaping of non-conflicting operations & leads to serial schedule.

| $T_3$ | $T_4$ |
|---|---|
| R(Q) | |
| | W(Q) |
| W(Q) | |

Non serializable.

S: $r_1(a)\ w_1(a)\ r_2(a)\ w_2(a)\ r_1(b)\ w_1(b)\ r_2(b)\ w_2(b)$

Q1 S: $T_1 T_2$ = $r_1(a)\ w_1(a)\ r_1(b)\ w_1(b)\ r_2(a)\ w_2(a)\ r_2(b)\ w_2(b)$

Find it is conflict serializable or not

Also make serial schedule of above schedule

Q2 S: $r_1(a)\ w_2(a)\ w_1(a)$

$T_1 T_2$ : Can't Swap

$T_2 T_1$ : Can't Swap Non conflict serializable.

# Precedence Graph



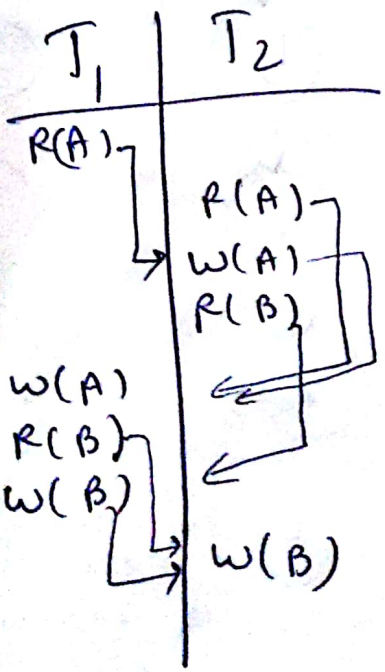
Draw precedence graph $G=(V, E)$ corresponding to schedule, S. $T_i \rightarrow T_j$ Edge consist of all edges

(1) $T_i$ execute $w(Q)$ before $T_j$ execute $R(Q)$

(2) $T_i$ " $R(Q)$ " $T_j$ " $w(Q)$

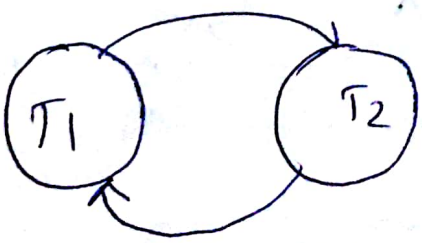
(3) $T_i$ " $w(Q)$ " " " $w(Q)$


Or


→ Look for conflict operation

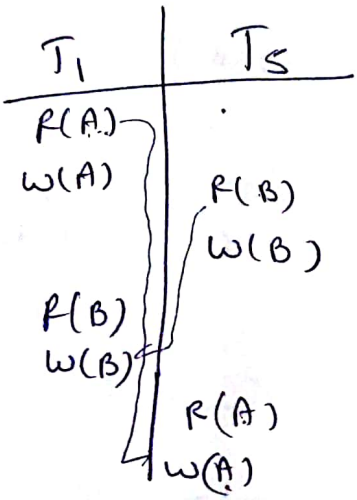
→

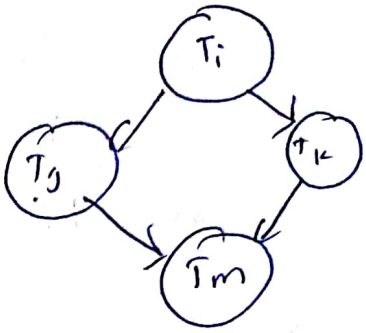


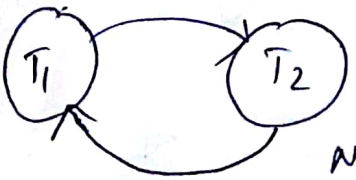

If cycle exist, then it is not conflict Serializable

## Serializability order

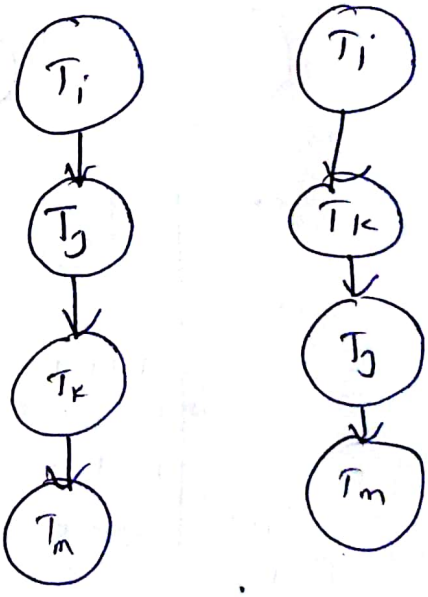
It can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called topological sorting.



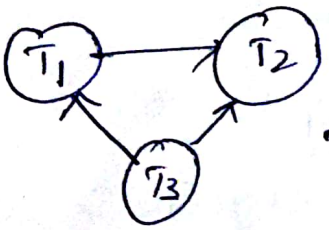




Non conf... Serial...



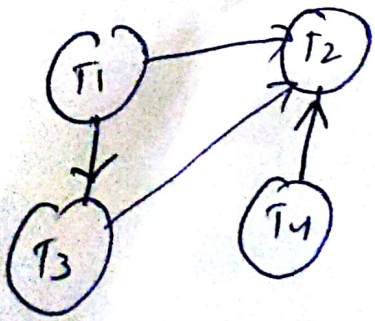

Q2




$T_3\ T_1\ T_2$

Look for the node with minimum no of incoming edges & put at top then in increasing order of incoming edge

Q3




$T_4\ T_1\ T_3\ T_2$

$T_1\ T_4\ T_3\ T_2$

$T_1\ T_3\ T_4\ T_2$

Swap Non-conflicting Instruction

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| w(A) | |
| | R(A) |
| | w(A) |
| R(B) | |
| w(B) | |
| | R(B) |
| | w(B) |

Serial ⟹

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| w(A) | |
| R(B) | |
| w(B) | |
| | R(A) |
| | w(A) |
| | R(B) |
| | w(B) |

Convert it into Serial Schedule

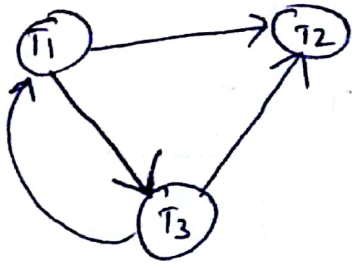
Now it is conflict serializable

Q2

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| R(X) | | R(Z) |
| | | w(Z) |
| | R(Y) | |
| R(Y) | | |
| | w(Y) | |
| | | w(X) |
| | w(Z) | |
| w(X) | | |

Now there 3! way to organize into serial schedule. But this could be very length.

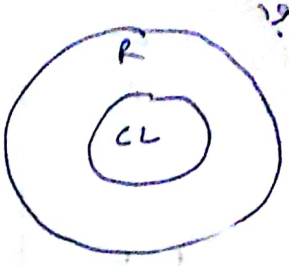
Make Topological Graph




Stop analysing if all the vertex are connected & make cycle. Now check for cycle. If cycle exist then it is non serial schedule & not Conflict serializable

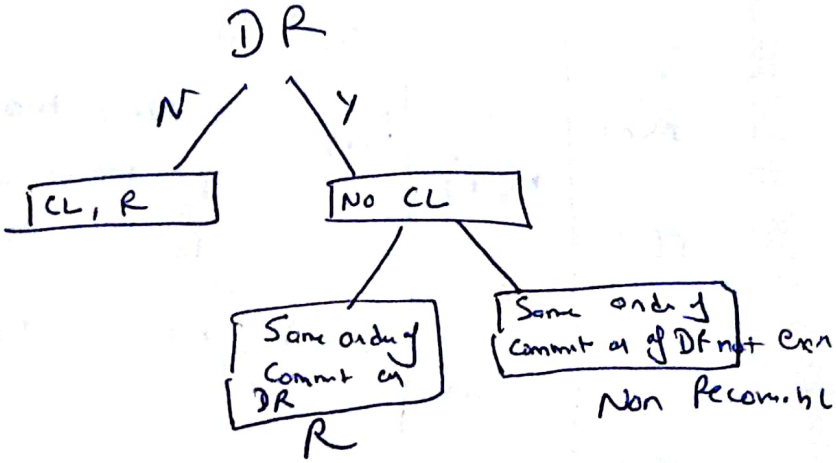
## Cascadeless Schedule (Strict)

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| R(A) | . | |
| W(A) | | |
| c | | |
| | R(A) | |
| | W(A) | |
| | c | |
| | | R(A) |
| | | c |




If dirty read exist then it is not cascadeless.

If schedule is CL then it is implicitly Recoverable.




Q

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| R(A) | | |
| W(A) | | |
| | | DR |
| | R(A) | |
| | W(A) | DR |
| | | R(A) |
| | C | |
| C | | |
| | | C |

find DR : If uncommitted value is read by transaction that is called DR

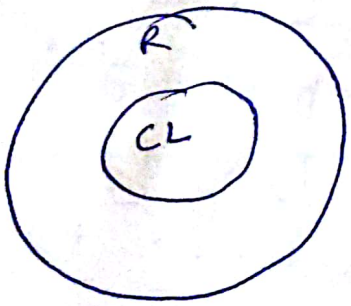
## Recoverable Schedule

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| R(A) | | |
| W(A) | | |
| Failure | R(A) | |
| | W(A) | |
| | | R(A) |
| C | C | C |

Now it is recoverable

Now it is cascading rollback

A Schedule is cascadeless if it doesn't contain cascading rollback. Problem is created by Dirty Read. Now Put commit just after the final statement of transaction Reverse to above termination.




| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| R(A) | | |
| W(A) | | |
| C | | |
| | R(A) | |
| | W(A) | |
| | C | |
| | | R(A) |
| | | C |

Now No dirty read $T_2$ is ready value from DB.

## Recoverable Schedule

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| R(A) | | |
| W(A) | | |
| | R(A) | |
| | W(A) | |
| | | R(A) |
| C | C | C |

This Schedule is recoverable but Not cascadeless.

## Recoverable Schedule

| $T_6$ | $T_7$ |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | commit |
| R(B) | |

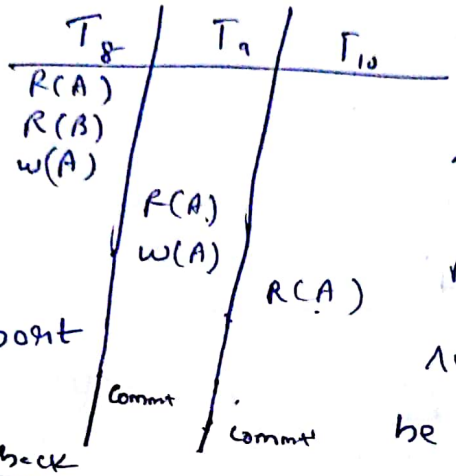
This is partial schedule because, no commit/abort operation for $T_6$ is included.
$T_7$ commits immediately after R(A), Thus $T_7$ commits before $T_6$ is still in the active state.
Now suppose $T_6$ fails & aborted. Now $T_7$ is dependent on $T_6$ which implies $T_7$ must be aborted to ensure atomicity. However $T_7$ is already committed ∴ it can't be rollback

**Defn** A recoverable schedule is one where, for each pair of transactions $T_i$ and $T_j$ such that $T_j$ reads data item previously written by $T_i$, the commit operation of $T_i$ appears before the commit operation of $T_j$. If above schedule is to be recoverable, $T_7$ would have to delay committing until $T_6$ commits

## Cascadeless Schedule



Even if schedule is recoverable, to recover correctly from the failure, we may have to rollback several transactions. It is caused by Dirty Read.




Consider this partial schedule, $T_9$ reads A written by $T_8$, $T_{10}$ reads A written by $T_9$. Now suppose $T_8$ fails & must be rollback. Since $T_9$, $T_{10}$ are dependent, must be rollback

Ex. Cascading Rollback with Recovery

This phenomena is called cascading Rollback. It is undesirable. It is desirable to restrict the Schedules to those where cascading rollbacks cannot occur.

**Defn** A cascadeless schedule is one where, for each pair of transaction $T_i$ and $T_j$ such that $T_j$ reads a data item previously written by $T_i$, the commit operation of $T_i$ appear before the read operation of $T_j$. Every cascadeless schedule is recoverable.

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| R(A) | | |
| W(A) | | |
| C | | |
| | R(A) | |
| | W(A) | |
| | C | |
| | | R(A) |
| | | C |

Eg. Cascadeless

Q1

| $T_1$ | $T_2$ |
|---|---|
| A = 10 | |
| R(A) | |
| A = A + 10 | |
| 20 W(A) | |
| | R(A) : 20 (Dirty Read) |
| | A = A - 5 : 15 |
| ↑ Rollback | W(A) |
| failure ~~~~ | C |
| C | |

$T_2$ is irrecoverable

Irrecoverable Schedule

after rollback, value of A is 10 again but $T_2$ will not be able to rollback.

Solution in $T_2$ must commit after the commit of $T_1$.

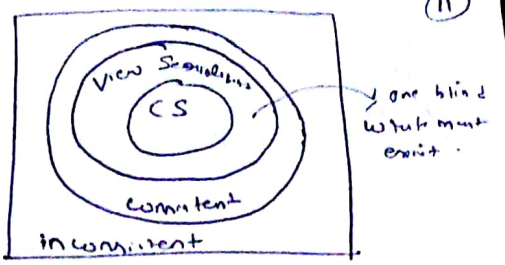
How to find Recoverable / irrecoverable

(i) Check DR

(ii) If DR is not present then it is recoverable.

(iii) If DR is present, Schedule is recoverable only if order of DR is same as order of commit.

## View Serializability




Less restrictive definition of equivalence of schedule is called view equivalence.
This leads to another definition as view serializability.

Two Schedules S and S' are said to be view equivalent if the following 3 condition holds.

(i) Set of transaction and operation must be same in S & S'.

(ii) for any operation $r_i(X)$ of $T_i$ in S, if the value of X read by the operation has been written by an operation $w_j(X)$ of $T_j$. the same condition must hold for the value of X read by operation $r_i(X)$ of $T_i$ in S'.

(iii) $W_k(Y)$ of $T_k$ is the last operation to write Y in S the $W_k(Y)$ of $T_k$ must also be the last operation to write Y in S'.

NOTE : If Blind write is there, then it could view serializable
If " " not there " it is not view serializable.

## Prove Schedule is view Serializable

find all possible serial schedules

$N.S \longrightarrow VS$

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 1 | 3 | 2 |
| 2 | 1 | 3 |
| 2 | 3 | 1 |
| 3 | 1 | 2 |
| 3 | 2 | 1 |

Serial Schedule

A Schedule is view equivalent, if each read operation of transaction reads the result of the same write operation in both schedules.
final write operation on each data must be same in both schedules

| Initial Read S | |
|---|---|
| A | $T_1$ |
| B | $T_1$ |

| Initial Read S' | |
|---|---|
| A | $T_1$ |
| B | $T_1$ |

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| R(B) | |
| W(B) | |
| | R(B) |
| | W(B) |

| $T_1$ |
|---|
| R(A) |
| W(A) |
| R(B) |
| W(B) |
| R(A) |
| W(A) |
| R(B) |
| W(B) |

(i) Initial Read should be same in both Schedules

Now look for final write

| S | |
|---|---|
| A | $T_2$ |
| B | $T_2$ |

| S' | |
|---|---|
| A | $T_2$ |
| B | $T_2$ |

(ii) Final write should be same.

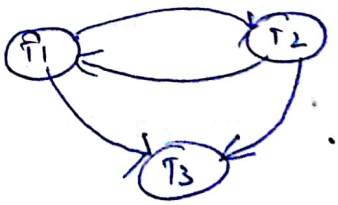
(iii) Intermediate write should be same. i.e all the intermediate Reads in transaction must be reading the value write : W-R conflict the be same.

Q

S

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| R(A) | | |
| | W(A) | |
| W(A) | | |
| | | W(A) |

(i) Check for conflict Serializable




∴ it is not CS

(ii) Since there are blind write ∴ it could be view Serial

S'

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| R(A) | | |
| W(A) | | |
| | W(A) | |
| | | W(A) |

(iii) Initial Read $T_1$ in both S, S'
Final write of A $T_3$ in both S, S'
Intermediate Read is not present.
∴ it is view equivalent. ∴ view serial

Strict Schedule is schedule which doesn't allow any transaction to read/write till any existing transaction already working on it. i.e Previous transaction must commit before other transaction R/W. It is more Strict than Cascadeless Schedule.

$S_1$

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| | W(A) |
| C | |
| | R(A) |
| | C |

Not Strict

$S_2$

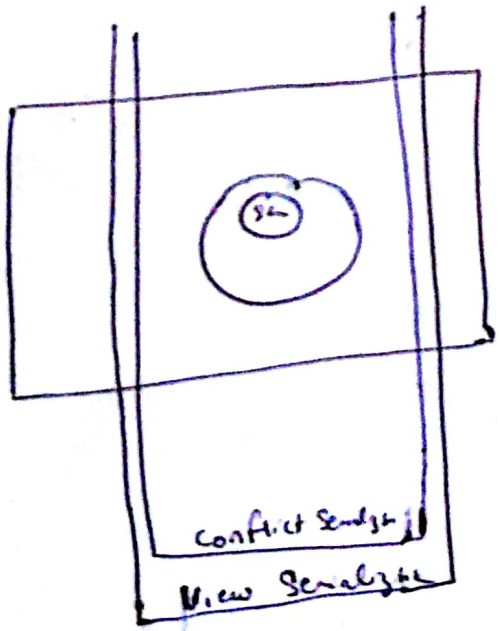
| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| C | |
| | W(A) |
| | R(A) |
| | C |

Strict

$S_3$

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| | R(B) |
| W(A) | |
| C | W(B) |
| | R(A) |
| | C |

Strict



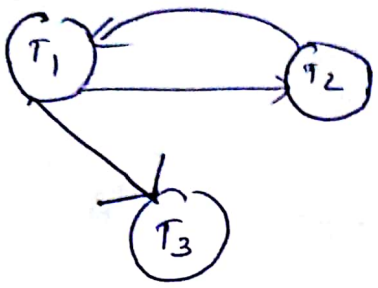

**Partial Schedule** : A schedule without commit or Abort is known as partial schedule

**Complete Schedule** : A schedule of transaction operations with commit / abort at the last is known as complete schedule.

Q Check whether schedule is View Serializable or not

S: $r_1(a)\ w_2(a)\ w_1(a),\ w_3(a)$

$T_1\ T_2\ T_3$ : $r_1(a)\ w_1(a)\ w_2(a),\ w_3(a)$




It is not conflict serializable

Check for VS

→ Ifirst Read(a) by $T_1$ in both schedule

→ last write(a) by $T_3$ in both schedule

→ Intermediate Rw is not present ∴ it is View serializable