10/04/2020

# TRANSACTION
## (Part I)

- Transaction can take place as:
  read (x), modify w(x-1)
  let 'x' be the no. of seats to be booked.

★ Concurrent Database Access:
- Data present in database can be accessed by DBMS.
- Other softwares are present to control transactions concurrently.
- Multiple clients can access database in concurrent manner.
- Consistent = accurate.

1). Attribute level inconsistency:
If both clients are trying to access the data concurrently (at a same time) Thus, either client 1 comes before client 2 or client 2 comes before client 1. But, result remains same. This is called concurrent execution.
↳ (Handled)

Scanned by CamScanner

2). Tuple - level inconsistency :

- C1 < C2 (C1 before C2) } concurrently
- C2 < C1 (C2 before C1) } (one after other)

∴ consistency is maintained.

3). Table - level inconsistency :

C1 // trying to update 'Emp' table
C2 // trying to update 'Project'

- C1 < C2 } consistency is maintained
- C2 < C1 } in both methods.

Goal :

# A transaction executes a set of operations running in isolated manner.

* Resilince To System Failure :-

- When there is a data transfer happening, then either there should be complete transfer or no transfer.
- Any data is 1st written into cache then to storage device.

Scanned by CamScanner

- Software (Oracle, etc)
  ↓
  DBMS
  ↓
  Data

- If any kind of failure occurs, the transaction should ensure that if data is partially lost then it is completely lost. → Resilince to System Failure.
- Once the transaction is ~~committed~~ completed, it should be committed.

Goal: all - or - nothing execution

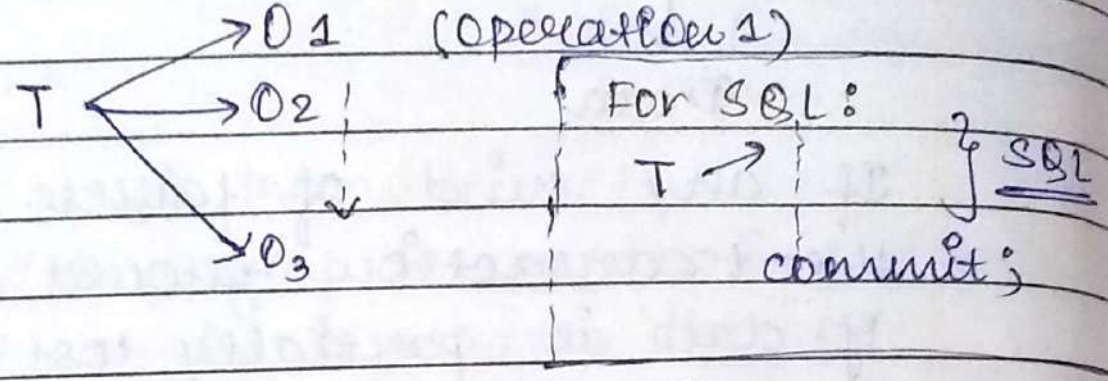
Solution for both concurrency & failures is TRANSACTION.

★ TRANSACTION is a sequence of one or more SQL operations treated as a unit:

- Transaction appear to run in isolation. (Concurrency Goal)
- If the system fails, each transaction's

Scanned by CamScanner

changes are reflected either entirely or not at all (System-Failure Goal).




- Once trans. is committed then anything after that is a new transaction.
- If trans. is not committed then it does "roll back".

## ★ TRANSACTION PROPERTIES :-

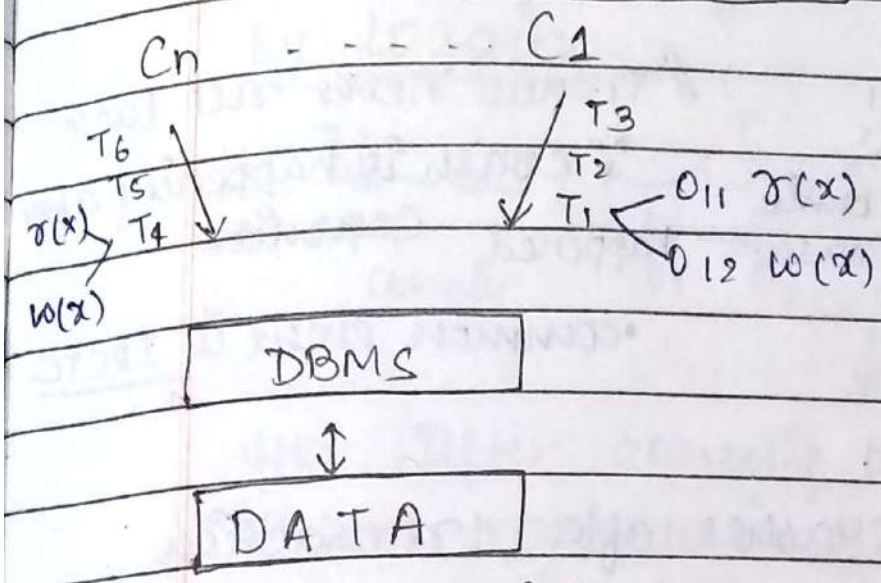
(III) A Atomicity
(IV) C Consistency
(I) I Isolation
(II) D Durability

- Every Trans. 'T' will have Operation 'O' in it. If commit / abort is not present then it is partial transaction

Scanned by CamScanner

- Complete Transaction : 'commit' / 'abort' present at end.

① ISOLATION




Both T1 & T4 can read simultaneously. But T1 & T4 both can't write simult. This is controlled by Scheduler.

* Serializability :-

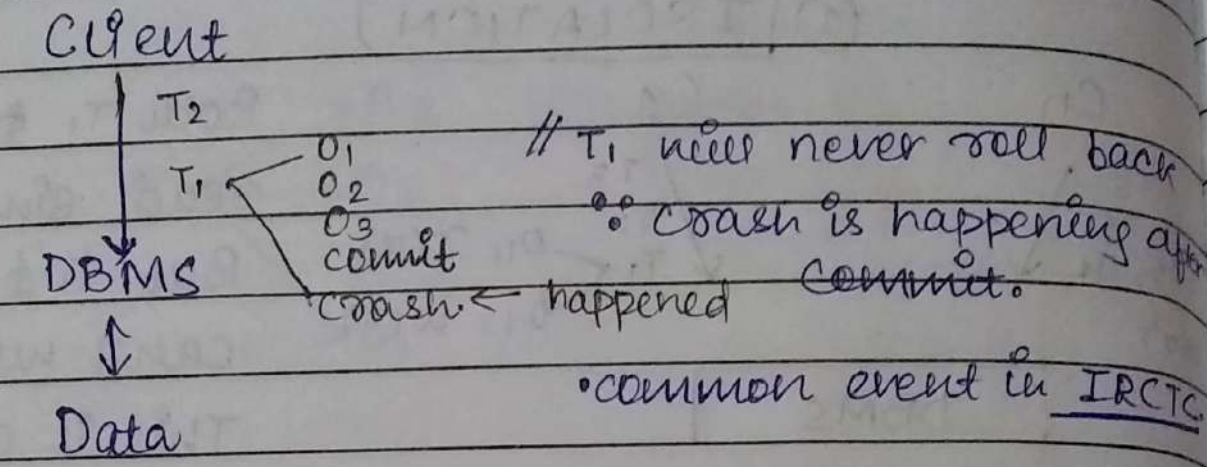
- Any tran. whose operations may be interleaved but exec. must be equivalent to some serial order. ~~of all~~

- System guarantees Serializability by LOCKING.

| ★ Debit Transaction<br>↳ 03 operations | Credit Transaction<br>↳ 03 operations |
|---|---|
| ID1 • Read (A/c no, bal) | Ic1 • Read (A/c no, bal) |
| ID2 • bal = bal - Amt | Ic2 • bal = bal + Amt |
| ID3 • Write (A/c no, bal) | Ic3 • Write (A/c no, bal) |

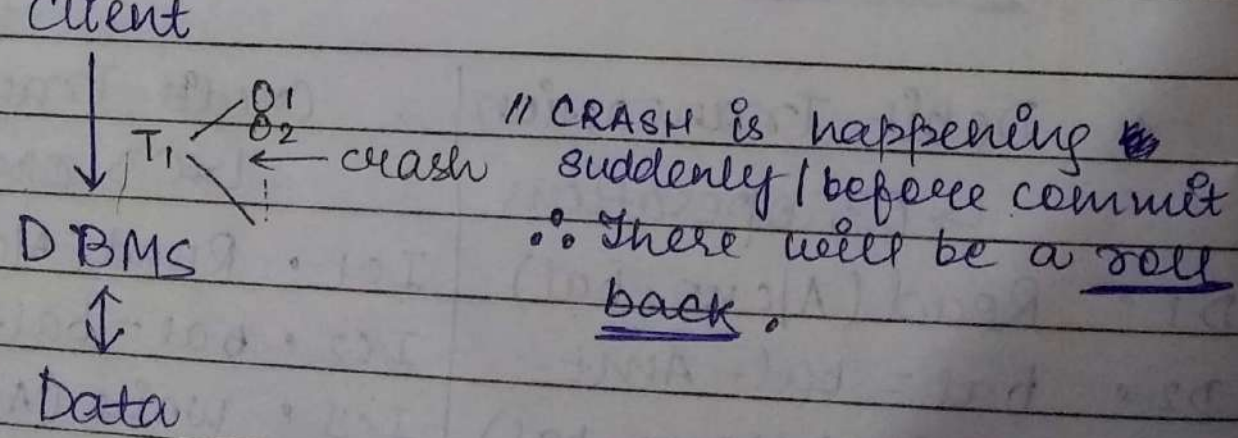
Scanned by CamScanner

## ② DURABILITY :



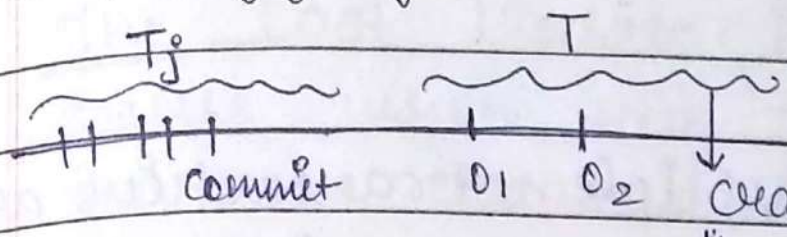

- If system crashes after transaction commits, all effects of transaction remain in database.
- The system guarantee Durability by Logging.

## ③ ATOMICITY :




Scanned by CamScanner

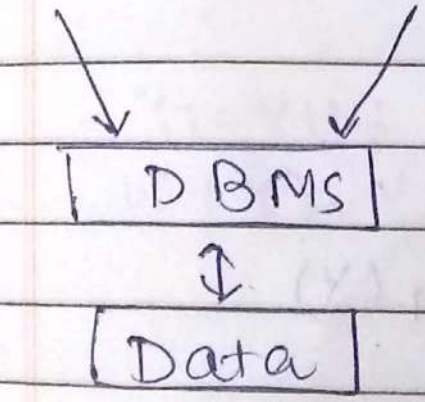
- Each transaction is "all-or-nothing" never left half done.
- The system guarantee Atomicity by logging.




↓ When crash occurs then, after coming back it goes to log it has maintained for previous transaction (Tj).

- Logging happens for every operation of a transaction.

## (4) CONSISTENCY:




Each client, each trans:
- can assume all constraint hold when trans. begins
- must guarantee all constraints hold when transaction ends.

* Holding → releasing → commit
the const. (lock) the const. (unlock)

* Consistency is maintained.

Scanned by CamScanner

Serializibility ⇒ constraints always hold.

- Lock the resource → Unlock the resource after Committing.
- The system guarantees consistency by
- when multiple transactions are accessing same data simultaneously, the access is protected through concurrency control mechanism.
  ↳ Consistency.

E.g

| $T_1$ | $T_2$ |
|---|---|
| read_item(X); | read_item(X); |
| X := X-N; | X := |
| write_item(X); | write_item(X); |
| read_item(Y); | |
| Y := Y+N; | |
| write_item(Y); | |

$T_1$: $r_1(x)$ $w_1(x)$ $r_1(y)$ $w_1(y)$

$T_2$: $r_2(x)$ $w_2(x)$

Scanned by CamScanner

* Advantages of concurrency control:
  - Increased processor & disk utilization.
  - Reduced average response time.

* The Lost Update Problem:-
Occurs when two trans. that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

E.g:

| $T_1$ | $T_2$ |
|---|---|
| r(X); | |
| X := X-N; | |
| | r(X); |
| | X := X+M; |
| w(X); | |
| r(Y); | |
| | w(X); ← Item X has an incorrect value because its update by $T_1$ is "lost" (overwritten). |
| Y := Y+N; | |
| w(Y); | |

Scanned by CamScanner

## * Temporary Update (Dirty Read) Problem:

Occurs when one trans. updates a database item & then the trans. fails for some reason.

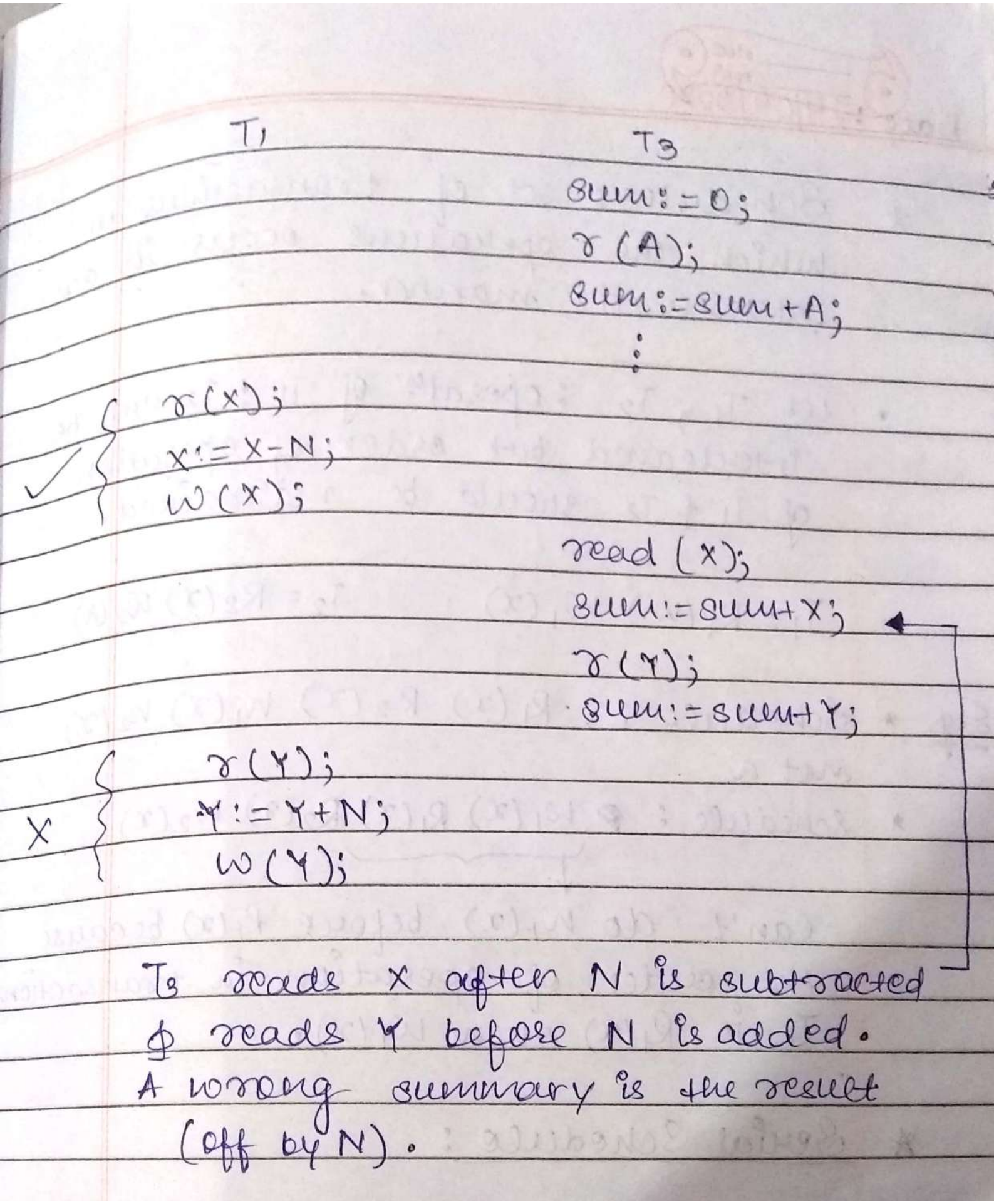
| $T_1$ | $T_2$ |
|---|---|
| r(x); | |
| x := x-N; | |
| w(x); | |
| | r(x); |
| | x := x+M; |
| | w(x); |
| r(Y); | |

crash → // Transaction $T_1$ fails & must change the value of X back to its old value; meanwhile $T_2$ has read the "temporary" incorrect value of X.

## * Incorrect Summary Problem:

↳ Correct summary is achieved when updation takes place after reading the item (data).

Scanned by CamScanner




Scanned by CamScanner

Date : 14/04/2020

* Schedules: set of transactions in which the operations occur in an interleaved manner.

- Let $T_1$, $T_2$ : operations of $T_1$ & $T_2$ can be interleaved but order of operations of $T_1$ & $T_2$ should be maintained.

- $T_1 = R_1(x)\ W_1(x)$ $\quad\quad$ $T_2 = R_2(x)\ W_2(x)$

Eg * Schedule : $R_1(x)\ R_2(x)\ W_1(x)\ W_2(x)$ not a

* Schedule : $W_1(x)\ R_1(x)\ R_2(x)\ W_2(x)$

Can't do $W_1(x)$ before $R_1(x)$ because the order of operation in transaction $T_1$ is $R_1(x)$ then $W_1(x)$.

* Serial Schedule :

- Serial Schedule ( $T_1 < T_2$ ) : $R_1(x)\ W_1(x)\ R_2(x)\ W_2$

- ($T_2 < T_1$) : Serial Schedule : $R_2(x)\ W_2(x)\ R_1(x)\ W_1$

Scanned by CamScanner

For 2 transactions → 2 serial schedules
3 → 6 " "

∴ For 'n' transactions n! serial schedules are possible.

E.g $R_1(x)\ R_2(x)\ W_1(x)\ W_2(x)$ : Non-Serial Schedule

∵ Interleaving is happening but not consecutively.

E.g:

| $T_1$ | $T_2$ |
|---|---|
| $R_1(X)$ | |
| $X := X-N;$ | |
| | $R_2(X);$ |
| | $X := X+M;$ |
| $W_1(X)$ | |
| $R_1(Y)$ | |
| | $W_2(X)$ ← Item X has incorrect value because its update by $T_1$ is lost |
| $Y := Y+N;$ | |
| $W_1(Y);$ | |

Sa: $r_1(x)\ r_2(x)\ w_1(x)\ R_1(y)\ w_2(x)\ w_1(y)$

Scanned by CamScanner

Eg

| $T_1$ | $T_2$ |
|---|---|
| $R_1(X)$; | |
| $X := X - N$; | |
| $W_1(X)$ | |
| | $R_2(X)$; |
| | $X := X + M$; |
| | $W_2(X)$; |
| $R_1(Y)$; | |

Transact$^n$: $T_1$ fails & must change the value of X back to its old value & $T_2$ has read incorrect value of X.

Schedule is : $R_1(X)\ W_1(X)\ R_2(X)\ W_2(X)\ R_1(Y)$ ... ($S_b$)

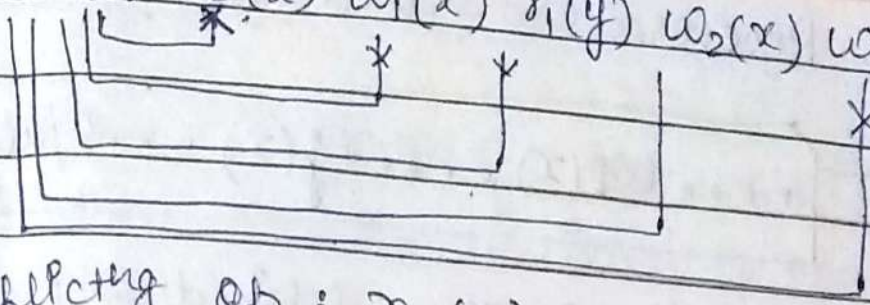
## Conflicting Operations :

Two operations in a schedule are conflicting if :-

- They belong to different transact$^n$
- They access the same item X
- At least one of them is a Write(X) operat$^n$.

Scanned by CamScanner

Eg $S_a$ : $r_1(x)\ r_2(x)\ w_1(x)\ r_1(y)\ w_2(x)\ w_1(y)$

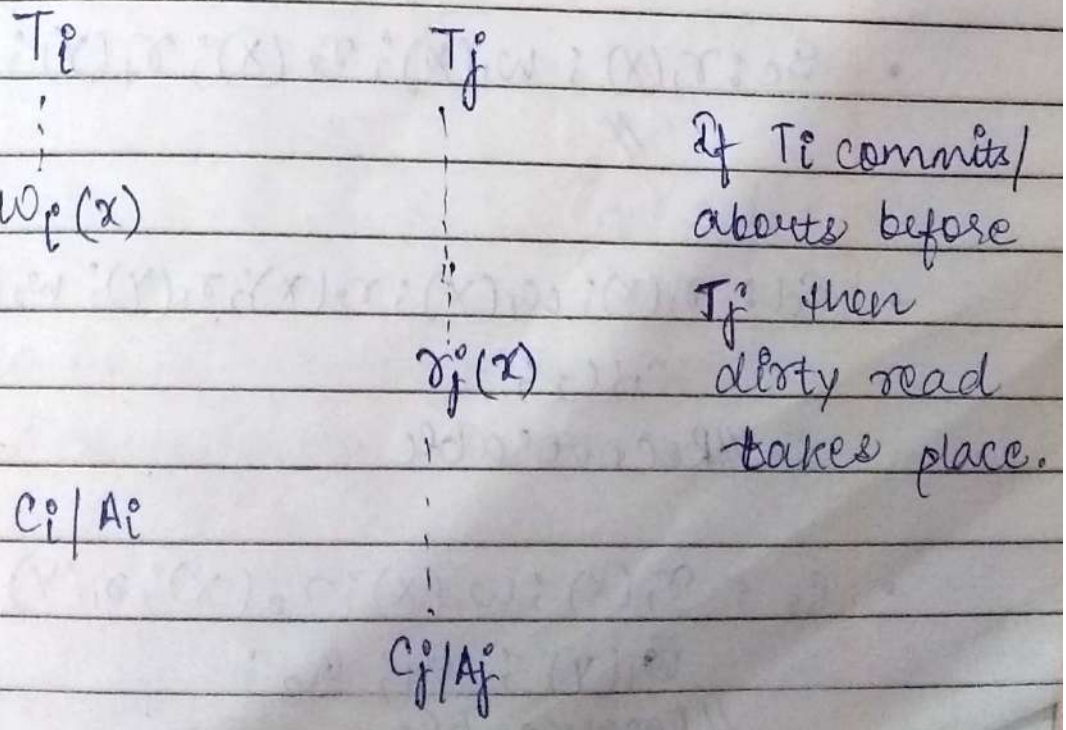


∴ Conflicting op : $r_1(x)\ w_2(x)$
$r_2(x)\ w_1(x)$
$w_1(x)\ w_2(x)$

- $r()\ r()$ can never be conflicting operations.

## * TYPES OF SCHEDULES :-

* **Recoverable Schedule** : If each transaction aborts/commits only after each trans. from which it has read has committed.




Scanned by CamScanner

→ If these properties are not present then it is recoverable.

## Recoverable Schedule:

.... $w_i(x)$ .... $r_j(x)$ .. c/a .... c/a$_j$

→ Previously defined on previous page

Eg: $S_a$: recoverable

- $S_b$: $r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$
  ∵ $a_2$ will happen by default.
  ∴ Recoverable.

- $S_a'$: $r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1;$
  ∴ recoverable.

- $S_c$: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$
  //

- $S_d$: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2;$
  //Recoverable

- $S_e$: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2;$
  //Recoverable

Scanned by CamScanner

② : Cascading Rollback Schedule:

## Avoiding Cascading Rollback: (ACR)

* A schedule is ACR if transactⁿ may read only values written by committed transactions.

* Note: One where every transactⁿ reads only the items that are written by committed transactions.

③ * Strict Schedule: A schedule in which a trans. can neither read/write an item x until the last trans. that wrote(x) has comit/aborted.

| Ti | Tj |
|---|---|
| W(x) | |
| | r( ) / w( ) |
| Ci(x) | |
| | Cj / aj. |

Sf: $W_1(X,5); W_2(X,8); a_1;$ // ACR
// cascadeless ; // Not strict.
(ACR)

Scanned by CamScanner

I

* Relationship among different schedules
  - Every ACR is Recoverable.
  - Every strict schedule is ACR.
  - Every strict schedule is serializable.

Date : 17/04/2020

* Serializibility : A schedule S of 'n' transactions is serializible if it is equivalent to some serial schedule of the same 'n' transactions.

- Let $S_1$ (non-serial) | $S_2$ (serial)

On executing, we get → res1() | res2()

If res1() = res2() then $S_1$ (non-serial) schedule is serializable.

- Any serial schedule is serilizible always i.e. they give correct result.
- If non-serial schedule is strict then it is serializible.
- If non-serial schedule → ACR then may or may not be serializible.

Scanned by CamScanner

i). Conflict equivalence
ii). View Equivalence

1]. Conflict Equivalence :-

$T_1 = R_1(X)\ W_1(X)\ R_1(Y)\ W_1(Y)$

$T_2 = R_2(X)\ W_2(X)$

Conflicting operations

Non-serial : $S_1 : R_1(X)\ W_1(X)\ R_2(X)\ W_2(X)\ R_1(Y)\ W_1(Y)$

Serial : $S_2 : R_1(X)\ W_1(X)\ R_1(Y)\ W_1(Y)\ R_2(X)\ W_2(X)$

- Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

- Conflicting operat$^{n}$ of

$S_1 : R_1(X)\ W_2(X)$
$W_1(X)\ R_2(X)$
$W_1(X)\ W_2(X)$

$\therefore S_1$ & $S_2$ are equivalent.

$S_2 : R_1(X)\ W_2(X)$
$W_1(X)\ R_2(X)$
$W_1(X)\ W_2(X)$

$T_1 < T_2$

$T_1 \rightarrow T_2$

Scanned by CamScanner

Ques: Check if it is conflict serializable or not?

E.g:

| $T_3$ | $T_4$ |
|---|---|
| read(x) | |
| | write(x) |
| write(x) | |

Schedule: $r_3(x); w_4(x); w_3(x)$

↳ check if serializable or not?

- Then, find an equivalent schedule.
- ∵ We have 2 transactions then we can have 2! schedules.

a) $T_3 < T_4$

∴ Serial schedule: $r_3(x)\ w_3(x)\ w_4(x)$

Conflicting operation: $r_3(x)\ w_4(x)$

$w_3(x)\ w_4(x)$

∵ The conflicting operations order is not same as the conf. op of the schedule given.

∵ It is not an ~~equivalent~~ conflicting serializible schedule.

Let's check 2nd one.

Scanned by CamScanner

b) $T_4 < T_3$

∴ Serial Schedule : $w_4(x)\ r_3(x)\ w_3(x)$

∴ Conflicting op : $w_4(x)\ r_3(x)$

$w_4(x)\ w_3(x)$

Soln: Again the order doesn't match.

∴ The given schedule is not conflicting serializable schedule.

NOTE: If the no. of transactions increases then we will have more no. of serial schedule. So, for checking the conflicting serializibility, we go via Precedence_Graph() Algorithm.

## * Precedence_Graph() Algorithm :

Step 1: Draw nodes corresponding to each of transaction in S.

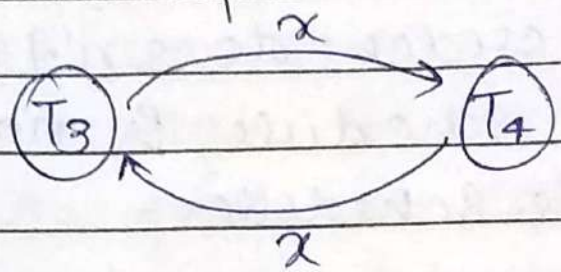
Step 2: Draw an edge from $T_i$ to $T_j$, if $T_i$ preceedes & conflicts with $T_j$. Complete the precedence graph.

Scanned by CamScanner

(conflict) serializibility.

Step3: If precedence graph is acyclic:
then "S is conflict serializable".
else "S is non-conflict serializable".

previous example:




∵ conflict ops are
$r_3(x)$ $w_4(x)$
$w_4(x)$ $w_3(x)$

∵ precedence graph is cyclic
∴ It is non-conflict serializability.

Ques:

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| $R_1(x)$; | | |
| $w_1(x)$; | | |
| $R_1(Y)$; | | |
| $w_1(Y)$; | | |

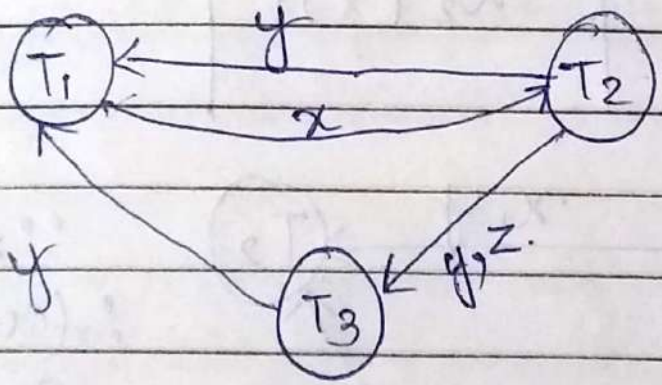
Scanned by CamScanner

Schedule given:-

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| | $R_2(Z)$; | |
| | $R_2(Y)$; | |
| | $W_2(Y)$; | |
| | | $R_3(Y)$; |
| | | $R_3(Z)$; |
| $R_1(X)$ | | |
| $W_1(X)$ | | |
| | | $W_3(Y)$; |
| | | $W_3(Z)$; |
| | $R_2(X)$ | |
| $R_1(Y)$; | | |
| $W_1(Y)$ | | |
| | $W_2(X)$ | |

∵ data items in both are same, we take recent one.

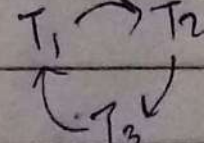
Sol^n:




∵ There exists a cycle $T_1 \rightleftarrows T_2$ in the precedence graph

∴ It is non-conflict Serializable.

or non-Serializable.

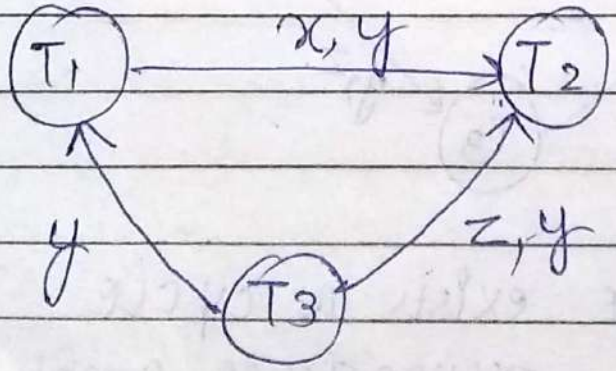



Scanned by CamScanner

Ques:

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| | | $R_3(Y)$; |
| | | $R_3(Z)$; |
| $R_1(X)$; | | |
| $W_1(X)$; | | |
| | | $W_3(Y)$; |
| | | $W_3(Z)$; |
| | $R_2(Z)$; | |
| $R_1(Y)$; | | |
| $W_1(Y)$; | | |
| | $R_2(Y)$; | |
| | $W_2(Y)$; | |
| | $R_2(X)$; | |
| | $W_2(X)$; | |

Soln:




∵ Acyclic
∴ (conflict) serializable

∴ Equivalent serial schedule: $T_3 \rightarrow T_1 \rightarrow T_2$

Scanned by CamScanner

## 2] View Serializibility:

* $S_1$ & $S_2$ are view equivalent if following cond$^{n}$'s are met:-

a) If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2.

b) If Tj reads value of A written by Ti in S1, then Tj also reads value of A written by Ti in S2.

c) If Ti writes final value of A in S1, then Ti also writes final value of A in S2.

i.e.

| | $S_1$ | $S_2$ (Serial) |
|---|---|---|
| 1). | $r_i(A)$.... | $r_i(A)$.... |
| 2). | ....$w_q(A)$... $r_j(A)$... | → |
| 3). | ..... $w_i(A)$ | →.... : $w_i(A)$ |

Scanned by CamScanner

| Eg. | $T_1$ | $T_2$ | $T_3$ |
|---|---|---|---|
| let $S_1$ → | read(x) | | |
| | | write(x) | |
| | write(x) | | |
| | | | write(x) |

Sol$^{n}$: let a serial schedule be $S_2$ : $T_1 < T_2 < T_3$

$S_2$ be : $r_1(x)\ w_1(x)\ w_2(x)\ w_3(x)$

- 1st cond$^{n}$ is satisfied [$\because R_1(x)$ is the starting value in both $S_1$ & $S_2$.]
- 2nd cond$^{n}$ doesn't come into picture [$\because \ldots W_i(x) \ldots r_j(x)$ doesn't occur anywhere. $\therefore$ need not check].
- $W_3(x)$ is the last value in both $S_1$ & $S_2$.

$\therefore$ $S_1$ is equivalent to $S_2$.
Thus, the given schedule is View Serializable.

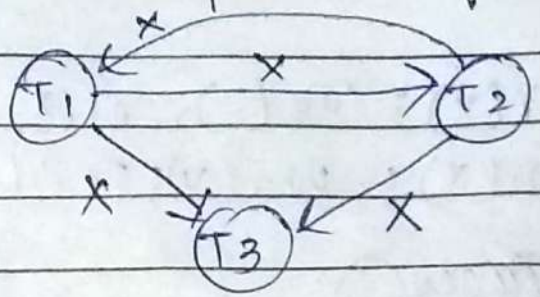
$S_1 \equiv S_2$

Scanned by CamScanner

// To check if a schedule is view serializable or not, check if all n! schedules.

this example only

* let's check for conflict serializable also :-




cycle occurs $T_1 \rightleftarrows T_2$

∴ Non-conflict serializable.

NOTE:

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

may or may not be conflict seri.

* Find other schedules which are view serializable with S1 :-

1) $T_1 < T_2 < T_3$ or $T_1 \rightarrow T_2 \rightarrow T_3$

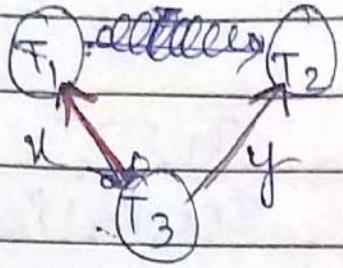
2) ~~$T_1 < T_3 < T_2$~~ → Only this is possible.

∵ we need to have $T_1$ at first for getting $R_1(X)$ at first & $T_3$ at last to get $W_3(X)$ at last.

Scanned by CamScanner

# EXERCISES:

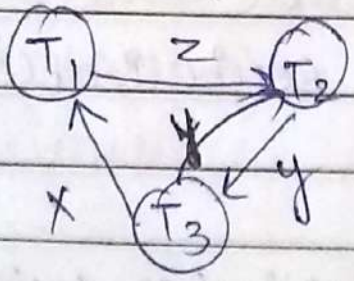
LP Q2:- * $S_1$: $r_1(x)$; $r_2(z)$; $r_1(x)$; $r_3(x)$; $r_3(y)$; $w_1(x)$; $w_3(y)$; $r_2(y)$; $w_2(z)$; $w_2(y)$




∴ serializable.

Equivalent serial schedule: $T_3 \rightarrow T_1 \rightarrow T_2$ or $T_3 \rightarrow T_2 \rightarrow T_1$

* $S_2$: $r_1(x)$; $r_2(z)$; $r_3(x)$; $r_1(z)$; $r_2(y)$; $r_3(y)$; $w_1(x)$; $w_2(z)$; $w_3(y)$; $w_2(y)$;




∴ non-serializable

LP Q4:- $T_1$ : $R_1(x)$ $R_1(y)$ $W_1(x)$

$T_2$ : $R_2(x)$ $R_2(y)$ $W_2(x)$ $W_2(y)$

i) write-read conflict : // non-serial

S: $R_2(x)$ $R_2(y)$ $W_2(x)$ $R_1(x)$ $R_1(y)$ $W_2(y)$ $W_1(x)$

// multiple answers possible.

Scanned by CamScanner

S: $R_2(x)\ R_2(y)\ W_2(x)\ R_1(x)\ W_1(x)\ W_2(y)\ R_1(y)$

Though, write-read conflict is there but the order of operations should be maintained in the transaction.

∴ The schedule is wrong.

∴ Not correct.

ii) read-write conflict:

- $R_1(x)\ R_1(y)\ W_2(x)\ R_2(x)\ R_2(y)\ W_2(x)\ W_2(y)$

or

- $R_2(x)\ R_1(x)\ R_1(y)\ R_2(y)\ W_1(x)\ W_2(x)\ W_2(y)$

iii) write-write conflict:

$R_1(x)\ R_1(y)\ R_2(x)\ R_2(y)\ W_1(x)\ W_2(x)\ W_2(y)$

LP Q5: PTO

Scanned by CamScanner

Rj(x)
(C)

LP

(5):

$S_3$: $r_1(x)$; $r_2(z)$; $r_1(z)$; $r_3(x)$; $r_3(y)$; $w_1(x)$; C1; $w_3(y)$; C3; $r_2(y)$; $w_2(z)$; $w_2(y)$; C2

↓ ACR (not bad)

∴
- Recoverable
- ACR
- Strict

$S_4$: $r_1(x)$; $r_2(z)$; $r_1(z)$; $r_3(x)$; $r_3(y)$; $w_1(x)$; $w_3(y)$; $r_2(y)$; $w_2(z)$; $w_2(y)$; C1; C2; C3;

not
- Recoverable
- not ACR
- not strict

$S_5$: $r_1(x)$; $r_2(z)$; $r_3(x)$; $r_1(z)$; $r_2(y)$; $r_3(y)$; $w_1(x)$; $w_2(z)$; $w_3(y)$; $c_{o2}(y)$; C3; C2;

↳ $C_3$ (not strict)

- Recoverable
- ACR
- not strict

} // ∵ There is no write-read conflicting op

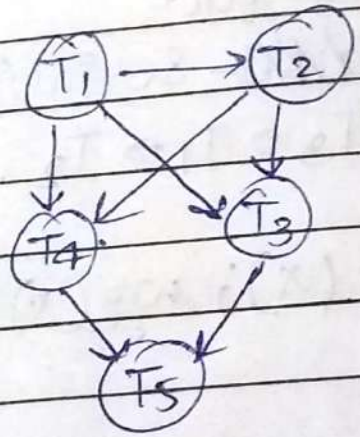
Scanned by CamScanner

• ACR is better but strict is best.
• Recoverable is minimum req.

| | Recoverable | ACR | Strict |
|---|---|---|---|
| S3 | ✓ | ✓ | ✓ |
| S4 | ✗ | ✗ | ✗ |
| S5 | ✓ | ✓ | ✗ |

∴ S3 is best.
S5 is good enough.
S4 is bad.

UP (7):




Topological Sort:

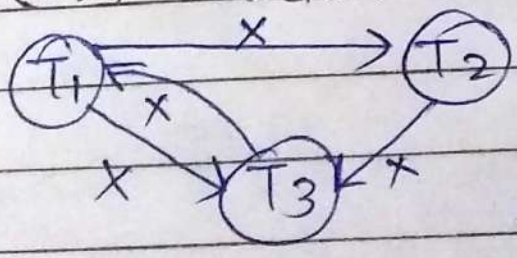
$T_1 \rightarrow T_2 \rightarrow T_3$

$T_4 \rightarrow T_5$

$T_1, T_2, T_3, T_4, T_5$

Ques: whether conflict serializable or not?

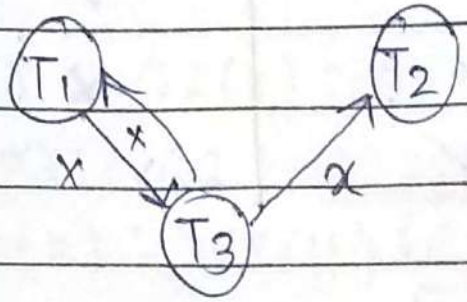
a) $r1(x); r3(x); w1(x); r2(x); w3(x);$




∴ non-serializable.

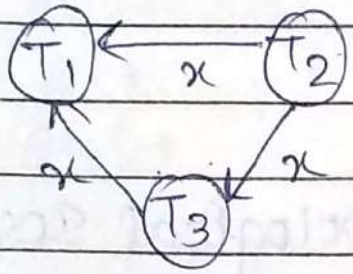
Scanned by CamScanner

b). $r_1(x)$; $r_3(x)$; $w_3(x)$; $w_1(x)$; $r_2(x)$




∴ non-serializable.

c). $r_3(x)$; $r_2(x)$; $w_3(x)$; $r_1(x)$; $w_1(x)$

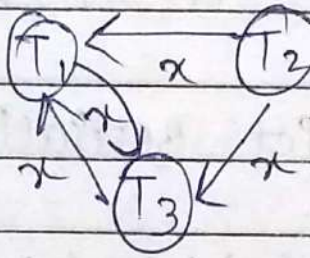



conflict

∴ ~~non~~ serializable.

$T_3 \rightarrow T_1 \rightarrow T_2$.

d). $r_3(x)$; $r_2(x)$; $r_1(x)$; $w_3(x)$; $w_1(x)$.




∴ non-serializable.

21/04/2020

GATE question

(B3) $r_1(A)$; $r_2(B)$; $w_2(B)$; $r_1(C)$; $r_2(D)$; $w_1(D)$; $w_2(C)$; $w_1(B)$; $C_1$; $C_2$;

Scanned by CamScanner

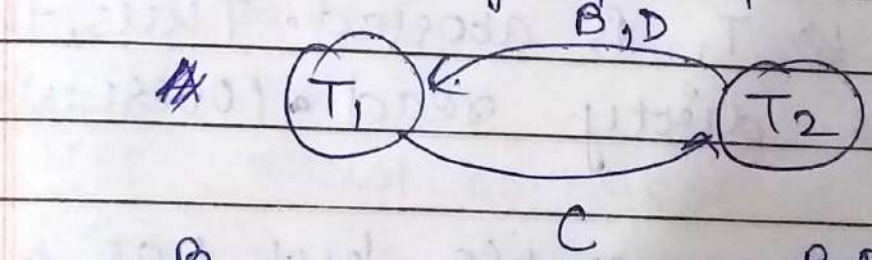
a). $r_1(A); r_1(C); w_1(D); r_2(B); w_2(B); r_2(D); w_1(B); w_2(C); C_1; C_2;$

b). $r_2(B); w_2(B); r_2(D); w_2(C); r_1(A); r(C); w_1(D); w_1(B); C_1; C_2;$

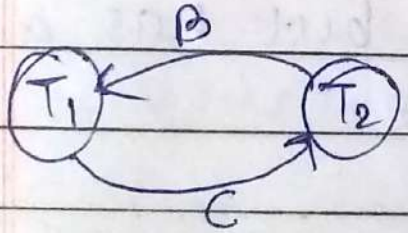
c). $r_2(B); w_2(B); r_2(D); r_1(A); r_1(C); w_1(D); w_1(B); w_2(C); C_1; C_2;$

d). $r_1(A); r_1(C); w_1(D); w_1(B); r_2(B); w_2(B); r_2(D); w_2(C); C_1; C_2;$

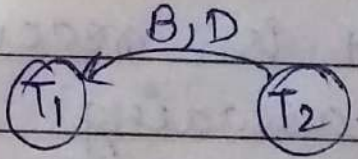
Sol$^{n}$: Precedence graph for question :-



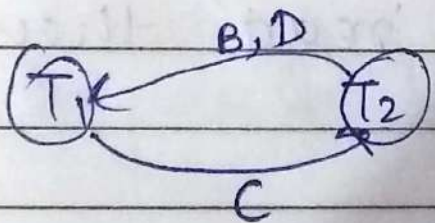

a).



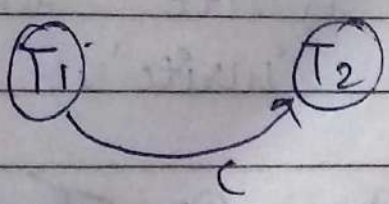

b).




c).




d).




∴ question graph matches with 'c'.

∴ It is conflict serializable with 'c'.

Scanned by CamScanner

GATE:

Ques: $S = r_2(X); r_1(X); r_2(Y); w_1(X); r_1(Y); w_2(X); a_1; a_2$

Which is TRUE?

a). S is non-recoverable.

b). S is recoverable, but has a cascading abort.

c). S does not have a cascading abort.

d). S is strict.

Soln: ... $w_1(X); r_1(Y); w_2(X); a_1; a_2$

After $w_1(X)$ is writing X, then for $w_2(X)$ i.e. $T_2$ writes X (reads X & then write X). But $T_1$ is aborted. Thus, there exists dirty read. (Overwriting).

∴ It is recoverable but has a cascading abort.

// 'Write' means 'read' then 'write'.

If $T_1$ is aborted then $T_2$ is also aborted here.

Scanned by CamScanner

Gate 2015

89.

| | $T_1$ | $T_2$ |
|---|---|---|
| 1 | $r(A)$ | |
| 2 | $w(A)$ | |
| 3 | | |
| 4 | | $r(C)$ |
| 5 | | $w(C)$ |
| 6 | | $r(B)$ |
| 7 | | $w(B)$ |
| 8 | | $r(A)$ |
| 9 | $r(B)$ | commit |

Transaction T1 fails immediately after time instance 9. which is correct?

a). T2 must be aborted & then both T1 & T2 must be restarted to ensure trans. atomicity.

b). Schedule S is non-recoverable & can't ensure trans. atomicity.

Soln: T2 can't be aborted ∵ It is commited. & 'S' is recoverable.

Atomicity → ✗ not ensured (all or nothing)

Scanned by CamScanner

NOTE: If T2 is not committed at time instance. & then option 'A' would be correct.

T2 should be aborted & T1 & T should re-start.

Scanned by CamScanner

# DBMS : Chapter 05

## TRANSACTION PROCESSING

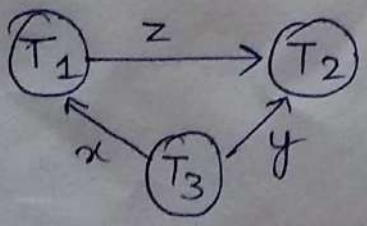
2). A user must guarantee that the transaction does not corrupt data or insert irrelevant data into the database.

E.g: In a bank database, a user must guarantee that a cash that a cash withdrawing transaction must accurately models with the amount a person removes from his/her bank account.

A DBMS must guarantee that transactions are executed fully and independently of all other transactions. An essential property of a DBMS is that the transaction should execute in isolation and DBMS must guarantee isolation with respect to concurrent execution of several transactions and database consistency. DBMS can guarantee Isolation through locking.

↓[quest$^{n}$ no. in LP (erron)]

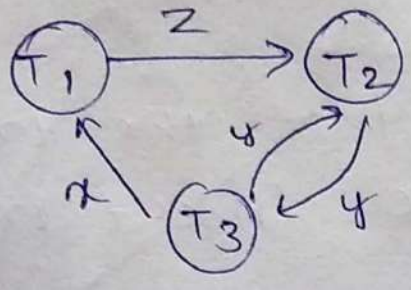
2). $S_1$ : $r_1(x)$ ; $r_2(z)$ ; $r_1(z)$ ; $r_3(x)$ ; $r_3(y)$ ; $w_1(x)$ ; $w_3(y)$ ; $r_2(y)$ ; $w_2(z)$ ; $w_2(y)$ ;




∵ Graph is acyclic.
∴ It is conflict-serializable.
∴ Equivalent schedule is :
$T_3 \rightarrow T_1 \rightarrow T_2$

Scanned by CamScanner

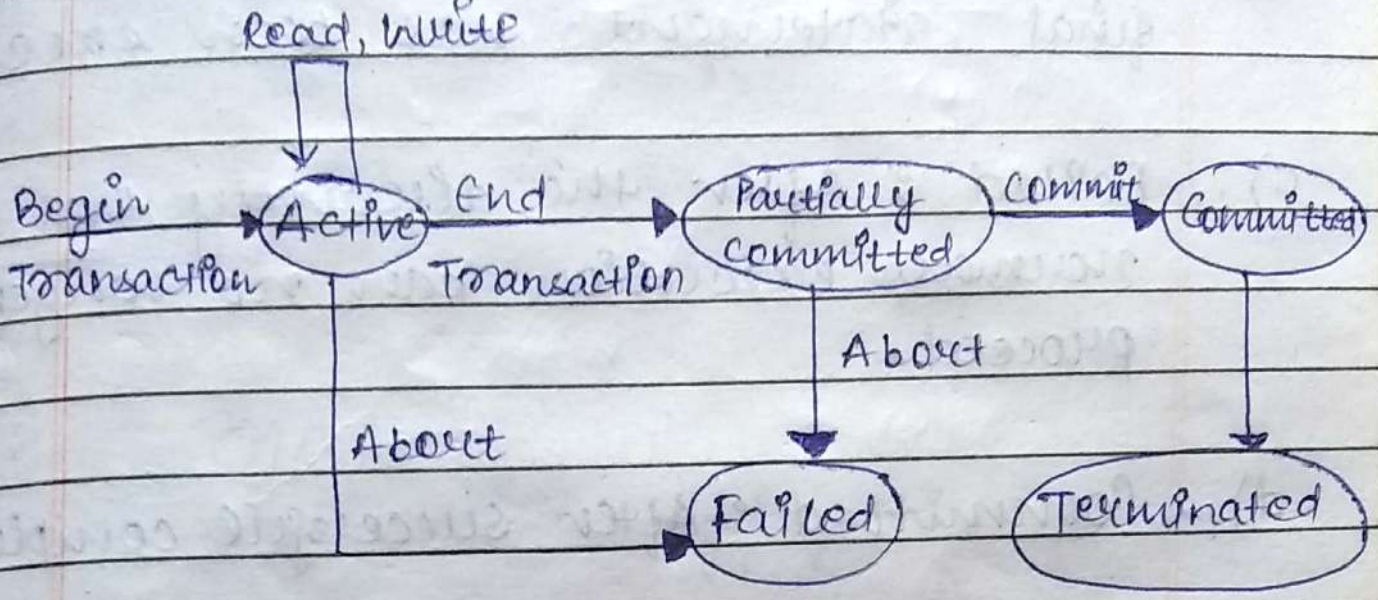
$S_2$ : $r_1(x)$ ; $r_2(z)$ ; $r_3(x)$ ; $r_1(z)$ ; $r_2(y)$ ; $r_3(y)$ ; $w_1(x)$ ;
$w_2(z)$ ; $w_3(y)$ ; $w_2(y)$ ;




∵ Graph is cyclic.
∴ It is non-conflict serializable.

Scanned by CamScanner

(3)* TRANSACTION STATE:




A transaction is an atomic unit of work that is either completed entirely or not done at all.

For recovery purposes, the system needs to keep track of when the transaction starts, terminates and commits or aborts.

A transaction must be in one of the following states:

a). Active: the initial state. Transaction stays in it while it is executing.

Scanned by CamScanner

b). Partially committed : After the final statement has been executed.

c). Failed : After the discovery that normal execution can no longer proceed.

d). Committed : After successful completion.

e). Aborted : After transaction has been rolled back and the database is restored to its prior state.
Two options are there once it is aborted :

- KILL the transaction
- restart the transaction (only if no integral logical error)

Scanned by CamScanner

4).

<i> Write - Read conflict:

$r_2(X)$ $r_2(Y)$ $w_2(X)$ $r_1(X)$ $w_2(Y)$ $w_1(X)$

<ii> Read - Read conflict:

$r_1(X)$ $r_1(Y)$ $r_2(X)$ $r_2(Y)$ $w_1(X)$ $w_2(X)$ $w_2(Y)$

<iii> Write - Write conflict:

$r_1(X)$ $r_1(Y)$ $r_2(X)$ $r_2(Y)$ $w_1(X)$ $w_2(X)$ $w_2(Y)$

5). $S_1$: $r_1(X)$; $r_2(Z)$; $r_1(Z)$; $r_3(X)$; $r_3(Y)$; $w_1(X)$; $c_1$; $w_3(Y)$; $c_3$; $r_2(Y)$; $w_2(Z)$; $w_2(Y)$; $c_2$

| Recoverable | ACR | Strict. |
|---|---|---|
| ✓ | ✓ | ✓ |

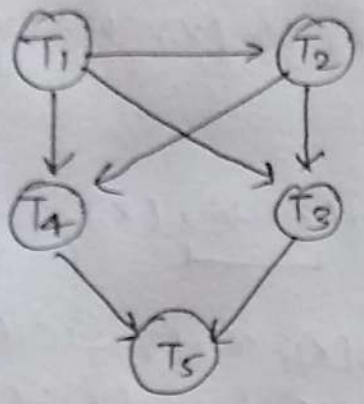
| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| $r_1(X)$ | | |
| | $r_2(Z)$ | |
| $r_1(Z)$ | | |
| | | $r_3(X)$ |
| | | $r_3(Y)$ |
| $w_1(X)$ | | |
| $c_1$ | | |
| | | $w_3(Y)$ |
| | | $c_3$ |
| | $r_2(Y)$ | |
| | $w_2(Z)$ | |
| | $w_2(Y)$ | |
| | $c_2$ | |

Scanned by CamScanner

$S_2$ : $r_1(X)$ ; $r_2(Z)$ ; $r_1(Z)$ ; $r_3(X)$ ; $r_3(Y)$ ; $w_1(X)$ ; $w_3(Y)$ ;
$r_2(Y)$ ; $w_2(Z)$ ; $w_2(Y)$ ; $c_1$ ; $c_2$ ; $c_3$ ;

| Recoverable | ACR | Strict |
|---|---|---|
| × | × | × |

7).




∵ The precedence graph is acyclic. ∴ It is conflict-Serializable & the equivalent Serial Schedule is :-

$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_5$

(Topological Sort)

Scanned by CamScanner

Q.

(1) ISOLATION:

Serializibility - Operations may be interleaved, but execution must be equivalent to some serial order of all transactions.

The System ensures serializibility by locking.

Scanned by CamScanner

- Transaction property Isolation tells that the results of one transaction will not be visible to other transaction till the transaction "commits".

(2) DURABILITY:

- If system crashes after transaction commits, all effects of transaction remain in database.
- It is guaranteed by Logging.
- The effect of committed transactions are not lost after commitment.

(3) Atomicity:

- Each transaction is "all-or-nothing" never left half done.
- The system guarantees atomicity by Logging.

Scanned by CamScanner

- Either all the instructions are executed in full or none.

## (4) CONSISTENCY:

- Each client, each transaction
  - can assume all constraints hold when transaction begins.
  - must guarantee all constraints hold when transaction ends.

- When multiple transactions are accessing the data simultaneously, the access is protected through concurrency control mechanisms.

Scanned by CamScanner

21/04/2020

# Chapter 06

## CONCURRENCY CONTROL TECHNIQUES

- These are the algorithms that are going to be used in order to handle conflicting operations to ensure serializability.
- Solution to issues in concurrency is serialization through Locking.
- Before reading → lock it
  after writing → unlock it

* Optimistic CC Algo:
  Timestap Alg (Basic timestamp)
  Optimistic: Conflicts are extremely low.

* Pessimistic CC Algo:
  locking Based Alg (2 phase locking) 2PL
  Pessimistic: conflicts are extremely high.

Scanned with CamScanner

## ★ Pessimistic CC Algos:

### * Locking Based Alg:

(1) Binary Lock:

LOCK → 1

NO LOCK → 0

disadv: We don't want the lock to be acquired for reading.

- It doesn't take care of read op where 'read' is a kind of shared operation.

(2) Read/Write locks: (Shared/Exclusive lock)

read_lock(x) r(x), w(x)
write_lock(x).

- If we have both op r(x) w(x) to be performed, It is better to acquire write lock initially.

rl(x) r(x) ul(x) wl(x) w(x) ul(x)

ul(x) → unlock x

wl(x) → write lock x

ul(x) → unlock x

OR

Scanned with CamScanner

wl(x) r(x)  w(x) ul(x)
and r(x)

∵ we can perform w(x) op when wl(x) is acquired.
whereas, w(x) can't be performed if we are holding a read lock rl.

24/04/2020

★ Goldilocks Rule: One shouldn't be too optimistic or too pessimistic.
↳ today used even for AI technology

NOTE: Shared / Exclusive locking scheme does not guarantee Serializibility

In order to guarantee Serializibility we go for 2 PL (2 phase locking)

★ 2 Phase Locking:
(schedule is guaranteed to be serializable).

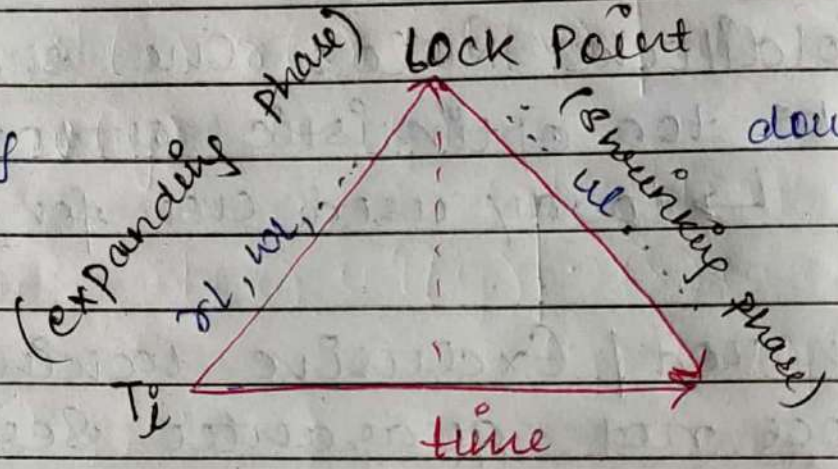
1st phase: growing / expanding phase.
2nd phase: shrinking phase.

P.T.O

Scanned with CamScanner

- rl → read lock
- wl → write lock
- ul → unlock

- In the growing phase, the trans has to acquire all the locks & other op. Once it reaches the lock point, it will start its shrinking phase (down). During shrinking phase, op can be performed and locks can be unlocked




- Upgradation of locks can happen in growing phase.
  - → upgrading rl to wl.

- Downgradation of locks can happen during shrinking phase.
  - → we have wl & we want it to convert to rl.

Scanned with CamScanner

{ guarantees serializability. }
2PL ↔ $T_1$'

{ doesn't guarantee serializability. }
→ Shared / Exclusive lock
$T_1$

E.g

| growing ph. | | |
|---|---|---|
| | rl(Y); | rl(Y); |
| | r(Y); | r(Y); |
| | wl(X); | wl(Y); |
| shrinking phase | ul(Y); | ul(X); |
| | r(X); | r(X); |
| | X := X+Y; | X := X+Y; |
| | w(X); | w(X); |
| | ul(X) | ul(X); |

* The use of locks can cause two disadvantage: problems:
1). deadlock
2). starvation

NOTE

- If every transaction in a schedule follows the 2PL protocol, the schedule is guaranteed to be serializable.

E.g:

| $T_1$ | $T_2$ | |
|---|---|---|
| wl(x) | wl(y) | deadlock |
| wl(y) | wl(x) | |
| ul(x) | ul(y) | starvation arises due to priority. |

Scanned with CamScanner

## ★ Variations of 2PL:

- Basic
- Conservative (deadlock free)
- Strict
- Rigorous

**E.g:** $T_1 : r(x)\ r(y)\ w(y)$

∴ **2PL :** $rl(x)\ r(x)\ wl(y) \mid ul(x)\ r(y)\ w(y)\ ul(y)$

∴ **Conservative 2PL :** $rl(x)\ wl(y) \mid$ lock point $r(x)\ ul(x)\ r(y)\ w(y)\ ul(y)$

★ Conservative 2PL: All the locks are acquired before execution begins of a transaction, then op. & unlocking takes place in shrinking phase.

- Never leads to deadlock.
- If it won't be able to acquire the lock, the transaction won't begin only. Only when all the locks required for trans. execution are acquired, then only trans. starts.

Thus, deadlock free

• If both $r(B)$ & $w(B)$ is there, then put $WL(B)$.

★ **Strict 2PL**: A trans. 'T' does not release any of its exclusive (write) locks until after it commits or aborts.

Strict: $wl_1(x)$ ..... $C_1$ ..... $ul_1(x)$

2PL : $wl_1(x)$ ..... $ul_1(x)$ ... $C_1$

• ~~Not~~ may have deadlock.

★ **Rigorous 2PL**: trans. 'T' does not release any of its locks until after it commits or aborts. → $(r/w)$

• It guarantees strict schedules.

**2PL Example (1)**

| $T_1$ | $T_2$ |
|---|---|
| $r(A)$; | $r(B)$; |
| $r(B)$; | $r(A)$; |
| if $A=0$ then $B=B+1$; | if $B=0$ then $A=A+1$ |
| $w(B)$; | $w(A)$; |

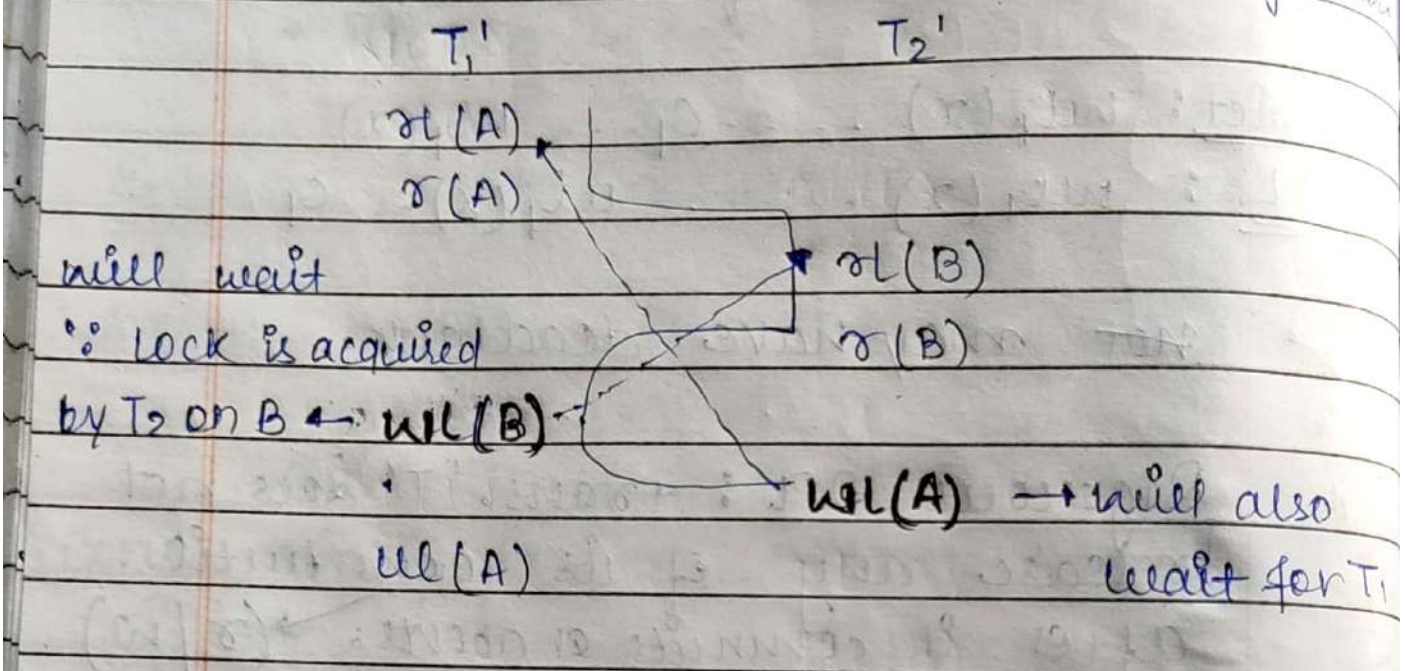
<i> 2PL $T_1'$: $rl(A)\ r(A)\ WL(B)$ | ~~$r(B)$~~ $ul(A)\ r(B)\ w(B)\ ul(B)$

$T_2'$: $rl(B)\ r(B)\ WL(A)$ | $ul(B)\ r(A)\ w(A)\ ul(A)$

Scanned with CamScanner

(ii) Can the execution of these transactions result in a deadlock?

// Take An interleaved schedule of ...

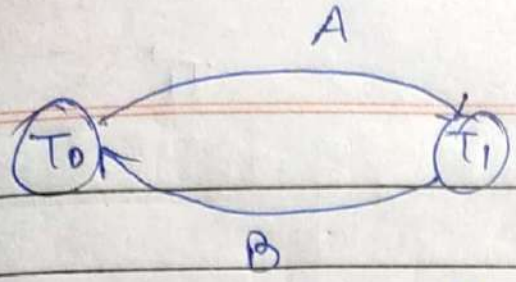



∵ Both are waiting for each other
∴ This is where deadlock is occuring.

## ★ Example (2)

| $T_0$ | $T_1$ |
|---|---|
| $r_0(A)$ | |
| $c_0(A)$ | |
| | $r_1(A)$ |
| | $r_1(B)$ |
| | C |
| $r_0(B)$ | |
| $w_0(B)$ | |
| C | |

Scanned with CamScanner




(i) ∴ Non-conflict serializable

(ii) Put them into 2PL form first. & then check.

| $T_0'$ | $T_1'$ | $T_1'$ |
|---|---|---|
| ~~WL(A)~~ | ~~rL(A)~~ | rL(A) |
| r(A) | ~~rL(A)~~ | r(A) |
| W(A) | ~~r(A)~~ | rL(B) |
| ~~WL(B)~~ | ~~r(B)~~ | UL(A) |
| UL(A) | ~~UL(A)~~ | r(B) |
| r(B) | ~~UL(B)~~ | UL(B) |
| W(B) | | C1 |
| UL(B) | | |
| C0 | | |

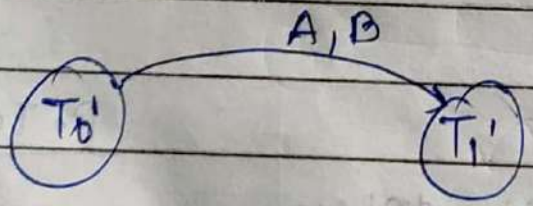
P.T.O

- Find 2PL form
- Make a schedule
- check for serializability

Scanned with CamScanner

(Pi) contd.

| $T_0'$ | $T_1'$ |
|---|---|
| $WL(A)$ | |
| $R(A)$ | |
| $W(A)$ | |
| $WL(B)$ | |
| $UL(A)$ | |
| | $RL(A)$ |
| | $R(A)$ |
| $R(B)$ | |
| $W(B)$ | |
| $UL(B)$ | |
| $C_0$ | |
| | $RL(B)$ |
| | $UL(A)$ |
| | $R(B)$ |
| | $UL(B)$ |
| | $C_1$ |



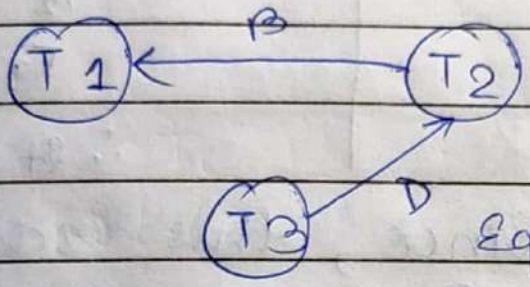

∴ conflict - serializable.

Scanned with CamScanner

* Example (3):

S1 : r3(D) W1(A) r2(B) W3(D) W1(B)
r2(D)

- Serializable :-




∵ It is acyclic
∴ conflict-serializable.
Equivalent serial schedule = $T_3 \rightarrow T_2 \rightarrow T_1$
or $T_3 < T_2 < T_1$

- 2PL :-

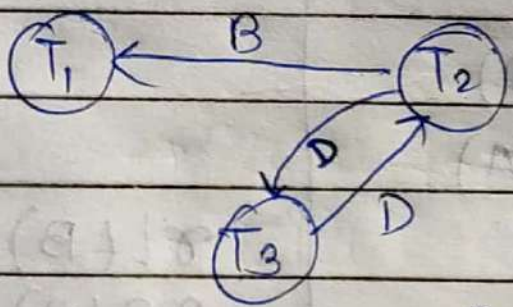
| $T_1$' | $T_2$' | $T_3$' |
|---|---|---|
| | | WL(D) |
| | | r3(D) |
| WL(A) | | |
| W1(A) | | |
| | rL(B) | |
| | r2(B) | |
| | | W3(D) Lock pt |
| | | U(D) |
| | rL(D) Lock Pt. | |
| | U(B) | |
| WL(B) | | |
| W1(B) Lock Point | | |
| U(A) | | |
| | | P.T.O |

Scanned with CamScanner

| $T_1'$ | $T_2'$ | $T_3'$ |
|---|---|---|
| $w(B)$ | | |
| | $r(D)$ | ∴ ~~Xx~~ ~~It~~ ~~is~~ not ~~2PL~~ |
| | $w(D)$ | $T_1 \rightarrow$ 2PL ✓ |
| | | $T_2 \rightarrow$ 2PL ✓ |
| | | $T_3 \rightarrow$ 2PL ✓ |
| | | ∴ It is 2PL. |

Strict 2PL: Since, this schedule require commit / abort & It is not given. ∴ We cannot tell about strict 2PL.

(ii) $S_2$: $w_1(A)\ r_2(B)\ w_3(D)\ w_1(B)\ r_2(D)\ w_3(D)$




∴ Non-conflict serializable.

Scanned with CamScanner

NOTE: every 2PL must be Serializable.

• 2PL :-

| $T_1$' | $T_2$' | $T_3$' |
|---|---|---|
| WL(A) | | |
| W1(A) | | |
| ~~UL(A)~~ | | |
| | rL(B) | |
| | r2(B) | |
| | | WL(D) |
| | deadlock | W3(D) |
| | rL(D) | |
| | UL(B) | |
| | r2(D) | |
| | | ~~UL(D)~~ |
| | W(D) | |

$T_2$' will wait for $T_3$' to release 'D'.
∴ Can't be 2PL.
& $T_3$' can't release because W3(D) is again present at end.

• Strict 2PL :- X Not strict 2PL.

V.V.I

NOTE: If a schedule is non-serializable then it won't be in 2PL. Similarly, it won't be in strict 2PL.
∴ Every 2PL must be serializable.
(But, we need to show in exam).

Scanned with CamScanner

epochconverter → time stamp generator (system)

## ★ TIMESTAMPING:

- A unique identifier created by DBMS that indicates relative starting time of a transaction : TIMESTAMP

## ★ Timestamp Based concurrency Control :-

- Does not use any locks for serializ-ability of schedules. Hence, no deadlocks.

- Each transaction will be assigned a Timestamp Value TS(Tᵢ) & based upon this the transactions are executed.

- Recent one → Youngest One
  Older one → Elder one

Any conflicting read write opⁿ are executed in timestamp Order.

Scanned with CamScanner

- W-TS(X): largest timestamp of any trans. that executed W(X) successfully.
- R-TS(X): " " " " R(X).
- TS(Ti): Timestamp (TS) of trans. Ti.

## * Basic timestamp Ordering Algo:

```
If Ti requests for read(X)
{
    If TS(Ti) < W-TS(X) // Checking for
    {
        rollback; (abort)   Older //
    }                       // came late //
    else
    {
        read(X);
        R-TS(X) = max(R-TS(X), TS(Ti));
    }
}

If Ti requests for write(X)
{
    If (TS(Ti) < R-TS(X)) || (TS(Ti) < W-TS(X))
    {
        rollback; abort   // Oldest //
    }
    else
    {
        write(X);                  // Youngest //
        W-TS(X) = TS(Ti);
    }
}
```

: largest = Youngest

Scanned with CamScanner

## ★ Strict Timestamp ordering:

a). Trans. T issues a W(x) operation:-
If TS(T) > R-TS(X) then delay T until the trans. T' that wrote or read X has terminated (committed or aborted).

b). Trans. T issues a r(x) operation:-
If TS(T) > TS(x) then delay T until trans. T' that wrote or read X has terminated (committed or aborted).

## ★ Thomas's Write Rule:

- If TS(Ti) < TS(Tj) & Ti writes X before TS(Tj) then we can just ignore that write, since it will be overwritten by Tj anyway.

Example a). St1; St2; r1(A); r2(A); W1(B); W2(B)
start trans A; start trans 2; old; young

R-TS(X) = 2 // trans 2 // the one that occurs the latest is the highest.

W1-TS(X) = 1 // whether W2(B) < W1(B) for W2(B)
no, it is not. ∴ It can write. ∴ W-TS(X) = 2

∴ Accepted

b). $S_{t1}$; $S_{t2}$; $r_2(A)$; $C_{02}$; $r_1(A)$; $w_1(A)$

For $r_2(A)$

$R\text{-}TS(A) = 2$

Co

For $r_1(A)$

$R\text{-}TS(A) = 1 \not< 2$

Roll back

For $w_1(A)$

/* Write is requested but already there is a read that has happened */

Same trans done read & now wants to write

∴ Check if $w_1 < r_2$

no, yes.

∴ can't write

∴ write ∴ ~~Accepted~~

c). $S_{t1}$; $S_{t2}$; $S_{t3}$; $r_1(A)$; $w_3(A)$; $C_{03}$; $r_2(B)$; $w_2(A)$

A/q to thomas's write rule, $w_2(A)$ needs to be ignored.

d). $S_{t1}$; $S_{t2}$; $S_{t3}$; $r_1(A)$; $w_1(A)$; $r_2(A)$;

oldest youngest

If $R2(A) < W_1\text{-}TS(A)$ $W_2\text{-}TS(A) = 1$

// $r_2(A)$ must wait (delay) until $w_1(A)$ commits.

// Strict. ∴ delayed

e). $S_{t1}$; $S_{t2}$; $S_{t3}$; $r_1(A)$; $w_2(A)$; $w_3(A)$; $r_2(A)$

∴ rolled back

∵ $r_2(A) < w_3(A)$ // ∴ rolled back

Check for :-

Example (5) · Recoverable · Conflict Serializable
· ACR · View Serializable
· Strict

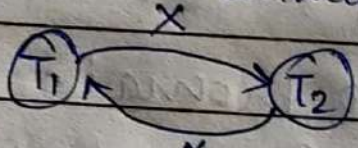
1). $T_1 : R(X)$ $T_2 : R(X)$ $T_1 : W(X)$ $T_2 : W(X)$

∴ Schedule : $S_1 \rightarrow R_1(X)\ R_2(X)\ W_1(X)\ W_2(X)$

| Recoverable | ACR | Strict | Conflict Ser. | View Serializ |
|---|---|---|---|---|
| ✓ | ✓ | X | X | X |

↓ No such pair of conflicting operations found.

↳ $W_1(X)\ W_2(X)$ // these should be commit after $W_1(X)$




non-conflict serializable

$T_1 < T_2$

$S_2 : R_1(X)\ W_1(X)\ R_2(X)\ W_2(X)$

Not View Serializable.

{∵ IInd cond$^n$ fails, thus not equivalent schedule, thus not view serializable}.

$T_2 < T_1$ X

Scanned with CamScanner

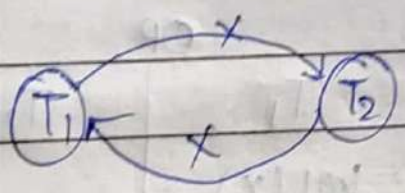
Rec :
ACR : . $w_2$ - - . $c_2$ . . . $r_1$

5). $T_1$ : R(X) $T_2$ : W(X) $T_1$ : W(X) $a_2$ $C_1$.

S : $R_1(X)\ W_2(X)\ W_1(X)\ a_2\ C_1$

| Recoverable | ACR | Strict | con. ser | view ser |
|---|---|---|---|---|
| ✓ | ✓ | X | X | |

no such conf. op.




Let Serial Schedule be :

$T_1 < T_2$ :- $S_2$ : $R_1(X)\ W_1(X)\ C_1\ W_2(X)\ a_2$

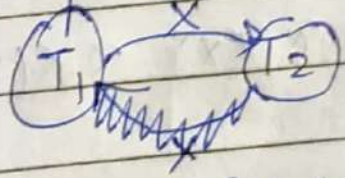
II nd condn doesn't satisfy.

1). $r_1(X)$ . . . → satisfied.

2). . . . $w_2(X)$ . $r_1(X)$ . . . → not satisfied.

8). S : $W_1(X)\ R_2(X)\ W_2(X)\ C_2\ C_1$

| Rec. | ACR | Strict | conflict | view ser |
|---|---|---|---|---|
| X | X | X | Y | X |




View $T_1 < T_2$

- $W_1(X)$ [illegible] $C_1$ $R_2(X)$ [illegible] $C_2$.

Not view serializable.

Example 6 · 2PL
· Strict 2PL
· Conservative 2PL
· Timestamp Ordering
· Thomas Timestamp Ordering

1). $r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)\ C_1\ C_2$ → optional.

2PL: $WL(x)\ r_1(x) \ldots r_2(x)$ but no lock available

OR

| $T_1$ | $T_2$ |
|---|---|
| $WL(x)$ | |
| $r_1(x)$ | |
| | ← waiting for lock |

Strict 2PL: Commit before unlocking

Conservative 2PL: All locks must acquired before transaction starts.

$WL_1(x)\ WL_2(x)$ — cannot do

$r_1(x)\ r_2(x)\ w_1(x)\ a\ W_2(x)$

↑ Should be these for strict 2PL

also

$r_1(x)\ r_2(x)\ C_2\ W_1(x)$ : —

Scanned with CamScanner

- Timestamp ordering

$r_1(x)$ R-TS(x) = 1

$r_2(x)$ R-TS(x) = 2

$r_1(x)$ $r_2(x)$ $w_1(x)$ If TS(TP) < R-TS(x) || ... // true

∴ Rollback

∵ {writing after younger} → '2' is younger

∴ Schedule is not acceptable, ∵ schedule is rolling back. Transaction has read.

- Thomas with Thomas Write Rule:

If there are two writes:-

$w_i \ldots w_j$ where $w_i < w_j$

∴ Ignore $w_i$.

$w_1(x)$ $w_2(x)$ // we can check here But we cannot go till this op. only because it fails at $w_1(x)$ only {same as above}.

| 2PL | Strict 2PL | C-2PL | TO | Thomas |
|---|---|---|---|---|
| X | X | X | X | X |

Scanned with CamScanner

5). $R_1(x)\ W_2(x)\ W_1(x)\ a_2\ C_1$

- **TO**

$r_1(x)$ R_TS(x) = 1

$W_2(x)$ If $TS(T_2) < R\_TS(x)$ || $TS(T_2) < W\_TS(x)$

$2 < 1$

↳ W_TS(x) = 2

$W_1(x)$ $1 < 1$ || $1 < 2$

abort. (Rollback)

Sch. is not acceptable. T.O ✗

- **Thomas** Ignore $W_1(x)$

If $TS(T_2) < W\_TS(x)$ ← check $W_2(x)$ $W_1(x)$

$1 < 2$ Thus, schedule is acceptable

**2PL:** $W_1(x)\ r_1(x)$ ... $r_2(x)$ no lock avl.

**Strict:** 2 PL same as above.

**Cons. 2PL** ↗

| 2PL | Strict | Cons. | TO | TO with Thomas |
|---|---|---|---|---|
| ✗ | ✗ | ✗ | ✗ | ✓ |

Scanned with CamScanner

Exercise 4 :-

b). Conflict Ser : put precedence graph First construct an Interleaved schedule as :-

Schedule will be :

$W_1(C)\ r_1(A)\ W_1(A)\ r_1(B)\ W_1(B)\ C_1$
$r_2(B)\ W_2(B)\ r_2(A)\ W_2(A)\ C_2$

∵ $r_2(B)$ { For $r_2(B)$ we need $L_2(B)$ but $T_1$ has already taken $L_2(B)$, thus we need to wait until $T_1$ releases the lock }. And unlocking is happening at end. Thus, ~~for~~ $T_1$ ~~to unlock~~ will complete first then $T_2$.

∴ Conflict serializable.

∵ All the schedules will point from $T_1$ to $T_2$. $T_1 \rightarrow T_2$.

$T_1 < T_2$

∵ It is conflict Ser. & serial schedule

∴ It will be recoverable, ACR, Strict, serial → Y

No deadlock { ∵ $T_1$ finishes then $T_2$ starts. }

Scanned with CamScanner

पूर्व मध्य रेलवे

# CONCURRENCY CONTROL TECHNIQUES

Nidhi Singh
Roll: 156

## Chapter-06

1). Conservative 2PL helps in preventing deadlock.

- Conservative 2PL is a protocol of locking all the items in a transaction/ schedule before starting the execution of the transaction, by pre-declaring its read & write set.
- In case, that ~~it is no~~ the data needed for locking is not available, then it will not lock any item and the transaction will not even begin.
- In such case, the transaction will wait till the data needed is available.
- Thus, it is a deadlock-free protocol & hence prevents deadlock.

3). Strict 2PL:

- Strict 2PL protocol explains that a transaction 'T' does not release any of its write-locks before it commits or aborts.
- It means that no other transactions can read or write an item that is written by T unless committed.

Scanned by CamScanner

- Thus, leads to strict schedule for recoverability.
- However, Strict 2PL is not a deadlock-free protocol.
- It blocks the writer blocks all the readers until the writer commits or aborts.

* Prove that basic 2PL guarantees conflict serializability of schedules:-

It can be proved that if every transaction in a schedule follows the 2PL protocol, then the schedule is guaranteed to be serializable (conflict serializable).

Eg. Let us consider transaction $T_1$ & $T_2$:-

$T_1$ : $r_1(Y)$ ; $r_1(X)$ ; $X := X+Y$ ; $w_1(X)$

$T_2$ : $r_2(X)$ ; $r_2(Y)$ ; $Y := X+Y$ ; $w_2(X)$

Now, by applying 2PL protocol, transactions become:

| $T_1'$ | $T_2'$ |
|---|---|
| read_lock(Y) | read_lock(X) |
| read(Y) | read(X) |
| write_lock(X) | write_lock(Y) |
| unlock(Y) | unlock(X) |
| read(X) | read(Y); |
| X = X+Y | Y := Y+X |
| write(X) | write(Y) |
| unlock(X) | unlock(Y) |

Here, the transaction satisfies 2PL.

Now, if we check for conflict serializability of schedule $S_1$: $T_1' < T_2'$ with let $X = 20$, $Y = 30$ {initially}

$\therefore S : T_1 < T_2 : X = 50\ Y = 80$

& $S_1 : T_1' < T_2' : X = 50\ Y = 80$

$\therefore$ 2PL guarantees conflict serializability.

Scanned by CamScanner

2. Basic timestamp ordering algorithm to control concurrency :-

- It does not use any locks for the serializability of schedules. Hence, no deadlocks.

- If Ti requests read(X) :-

```
If TS(Ti) < W_TS(X)
{
    rollback;
}
else
{
    read(X);
    R_TS(X) = max{R_TS(X), TS(Ti)};
}
```

- If Ti requests write(X) :-

```
If TS(Ti) < R_TS(X) || TS(Ti) < W_TS(X)
{
    rollback;
}
else
{
    write(X);
    W_TS(X) = TS(Ti);
}
```

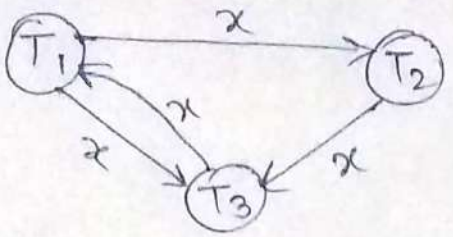
- Notations :- TS(Ti) : Timestamp value of Transact$^{n}$ Ti.

R_TS(X) : largest timestamp of any transact$^{n}$ that executed R(X) successfully.

W_TS(X) : largest timestamp of any transact$^{n}$ that executed W(X) successfully.

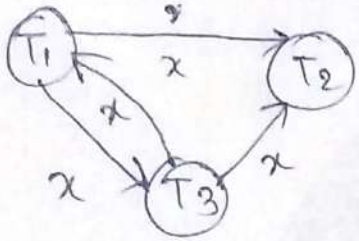
Scanned by CamScanner

6). (a) $r_1(x)$ ; $r_3(x)$ ; $w_1(x)$ ; $r_2(x)$ ; $w_3(x)$ ;



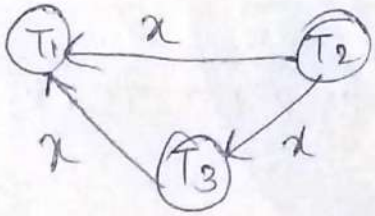

∵ The precedence graph is cyclic in nature.
∴ Non-conflict Serializable.

(b) $r_1(x)$; $r_3(x)$; $w_3(x)$; $w_1(x)$; $r_2(x)$;



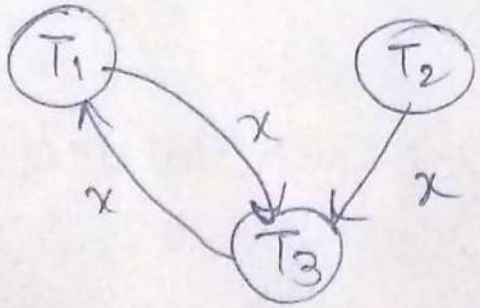

∵ Cyclic in nature.
∴ Non-conflict Serializable.

(c) $r_3(x)$ ; $r_2(x)$ ; $w_3(x)$ ; $r_1(x)$ ; $w_1(x)$ ;




∵ Acyclic
∴ Conflict Serializable.
Equivalent Serial Schedule can be : $T_2 \rightarrow T_3 \rightarrow T_1$

(d) $r_3(x)$ ; $r_2(x)$ ; $r_1(x)$ ; $w_3(x)$ ; $w_1(x)$ ;




∵ Cyclic.
∴ Non conflict Serializa

Scanned by CamScanner

4). $S_1$ : $r_2(z)$ ; $r_2(y)$ ; $w_2(y)$ ; $r_3(y)$ ; $r_3(z)$ ;
$r_1(x)$ ; $w_1(x)$ ; $w_3(y)$ ; $w_3(z)$ ; $r_2(x)$ ;
$r_1(y)$ ; $w_1(y)$ ; $w_2(x)$

| (300) $T_1$ | (100) $T_2$ | (200) $T_3$ ← Timestamp based on Arrival. |
|---|---|---|
| | $r_2(z)$ | |
| | $r_2(y)$ | |
| | $w_2(y)$ | |
| | | $r_3(y)$ |
| | | $r_3(z)$ |
| $r_1(x)$ | | |
| $w_1(x)$ | | |
| | | $w_3(y)$ |
| | | $w_3(z)$ |
| | $r_2(x)$ | ← aborts at this point. |
| $r_1(y)$ | | |
| $w_1(y)$ | | |
| | $w_2(x)$ | |

| | X | Y | Z |
|---|---|---|---|
| RTS | 300 | ~~100~~ 200 | ~~100~~ 200 |
| WTS | 300 | 100 | 0 |

$r_2(z)$ Executed, ∴ R-TS(z) = 100

$r_2(y)$ Executed, ∴ R-TS(y) = 100

$w_2(y)$ executed, ∴ W-TS(y) = 100

$r_3(y)$ Executed, ∴ R-TS(y) = max(100, 200) = 200

$r_3(z)$ Executed, ∴ R-TS(z) = max(100, 200) = 200

$r_1(x)$ Executed, ∴ R-TS(x) = 300

$w_1(x)$ Executed, ∴ W-TS(x) = 300

$r_2(x)$ ∵ 100 < W-TS(x) = 300 ∴ Rollback (abort).

Hence, ~~transac~~ schedule aborts.

Scanned by CamScanner

S2: $r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(Z); r_1(Y); w_1(Y); r_2(Y); w_2(Y); r_2(X); w_2(X)$.

(200) (300) (100) ← Timestamp values.

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| | | $r_3(Y)$ |
| | | $r_3(Z)$ |
| $r_1(X)$ | | |
| $w_1(X)$ | | |
| | | $w_3(Y)$ |
| | | $w_3(Z)$ |
| | $r_2(Z)$ | |
| $r_1(Y)$ | | |
| $w_1(Y)$ | | |
| | $r_2(Y)$ | |
| | $w_2(Y)$ | |
| | $r_2(X)$ | |
| | $w_2(X)$ | |

| | X | Y | Z |
|---|---|---|---|
| RTS | ~~200~~ 300 | ~~100~~ ~~200~~ 300 | ~~100~~ 300 |
| WTS | ~~200~~ 300 | ~~100~~ ~~200~~ 300 | 100 |

Since, all the transactions of the schedule are completed.
∴ The algo. allows execution of this schedule.

$r_3(Y)$ $100 < 0$ X ∴ Executed, ∴ R_TS(Y) = 100

$r_3(Z)$ $100 < 0$ X ∴ Executed, ∴ R_TS(Z) = 100

$r_1(X)$ $200 < 0$ X ∴ Executed, ∴ R_TS(X) = 200

$w_1(X)$ $200 < 200 \;||\; 200 < 0$ X ∴ Executed, ∴ W_TS(X) = 200

$w_3(Y)$ $100 < 100 \;||\; 100 < 0$ X ∴ Executed, ∴ W_TS(Y) = 100

$w_3(Z)$ $100 < 100 \;||\; 100 < 0$ X ∴ Executed, ∴ W_TS(Z) = 100

$r_2(Z)$ $300 < 100$ X ∴ Executed, ∴ R_TS(Z) = max(100, 300) = 300

$r_1(Y)$ $200 < 100$ X ∴ Executed, ∴ R_TS(Y) = 200

$w_1(Y)$ $200 < 100 \;||\; 200 < 100$ X ∴ Executed, ∴ W_TS(Y) = 200

$r_2(Y)$ $300 < 200$ X ∴ Executed, ∴ R_TS(Y) = 300

$w_2(Y)$ $300 < 200 \;||\; 300 < 300$ X ∴ Executed, W_TS(Y) = 300

$r_2(X)$ $300 < 200$ X ∴ Executed, ∴ R_TS(X) = 300

$W_2(X)$ $300 < 200 \;||\; 300 < 300$ X ∴ Executed, W_TS(X) = 300

Scanned by CamScanner