Emp(p#, pname, empNo, ename, Job, chg/day, nday)

p# → pname — I

p#, empNo → pname, ename, Job, chg/Day, nday — BCNF

empNo → ename, Job, chg/Day — I

Job → chg/Day — II

p#	pname	empNo	ename	Job	chg/Day	nday
1	I	101	R	Acc	1000	30
1	I	103	A	Ma	4000	15
1	I	105	B	Cl	500	30
1	I	106	C	Cl	500	25
2	H	101	R	Acc	1000	30
2	H	103	A	Ma	4000	10
2	H	107	B	Cl	500	30
2	H	109	S	Cl	500	30
3	E	105	D	Cl	500	30
3	E	103	A	Ma	4000	15
3	E	108	E	Cl	500	30
4	P	104	X	Client	1000	30

p#	pname
1	I
2	H
3	E
4	P

EmpNo	Ename	Job	Chg/Day
101	R	Acc	1000
103	A	Ma	4000
105	B	Cl	500
106	C	Cl	500
107	B	Cl	500
109	S	Cl	500
108	E	Cl	500
104	X	Client	1000

p#	empNo	nday
1	101	30
1	103	15
⋮	⋮	⋮

RA → ✓

IA ✓

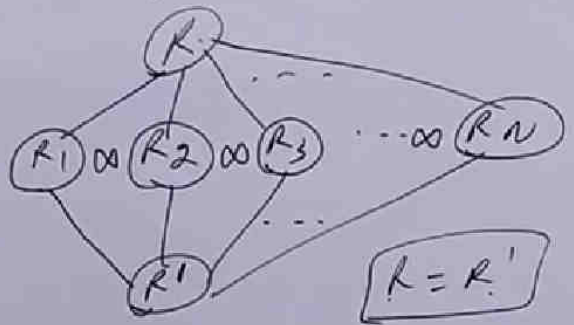
UA ✓

OA ✓

138. 5th normal form

5th Normal form (PJNF)

→ 5th Normal form is based on concept of **Join Dependencies**.



I NF → Atomic values
II NF → Partial Dependencies
III NF → Transitive Dependencies
IV NF → M.V.D
V NF → Join Dependencies

Join Dependency:

→ A table is said to have Join dependency if it is possible to decompose it into 2 or more subtables & by performing natural Join b/w all these tables we get original table

→ Join Dependency is trivial if one of the subtable is original table itself

5th N.F.

→ For each Non trivial Join Dependency each Subtable must be Superkey of R.

2:56 / 6:21

IS 3NF Perfect ?

faculty :-

fid	office	Dept	rank	DOH
f101	2	1	3	4
f101	6	5	7	8
f102	2	1	3	6
f105	7	5	6	4
f103	2	1	7	8
f104	6	5	4	9

⇒ 1NF

faculty (fid, office, dept, rank, DOH)

fid, dept → office, rank, DOH → BCNF

office → dept → 1NF

CK → fid, dept
fid, office

fid, office, dept, rank, DOH

(office, dept)

fid office rank DOH

Fid	office	rank	DOH
f101	2	3	4
f101	6	7	8
f102	2	3	6
f105	7	6	4
f103	2	7	8
f104	6	4	9

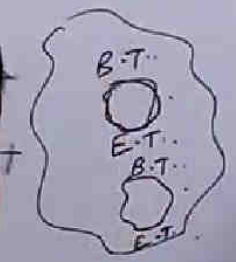
R.A. ✗
I.A. ✗
D.A. ✗
U.A. ✗

office	dept
2	1
6	5
7	5

139. introduction to transactions

TRANSACTIONS

→ Transaction is a part of HLL program which is either completely executed or not executed at all
→ We can identify this part by 2 System calls i.e "begin transaction" & "end transaction"
→ whatever we write in between begin transaction & end transaction constitute a single transaction
→ A HLL program can have more than one transaction also.
→ We can write this program in any HLL like COBOL, C, C++, Java, SQL etc.



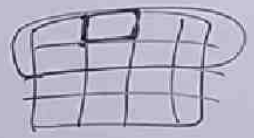
→ Transaction access Database "item".
→ Database item can be cell of table or single tuple or collection of tuples or entire table or entire database (collection of tables)
→ Size of database item is known as "granularity"
→ For simplicity we assume granularity to be cell of table.
→ In simplest scenario transaction will read some items & write some items.
→ If transaction only read items but doesnot write any item in database then it is called as Readonly transaction

139. introduction to transactions

TRANSACTIONS

Read item(x) :- ??

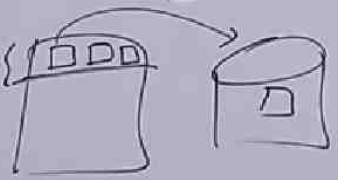
① Get disk address of block that contain the item x.
② Read in block (Transfer block from H.D. to RAM)
③ Copy the value of item x from RAM into program variable x.



Write item(x) :-

① same
② same
③ Copy value of program variable x into block in RAM containing item 'x'.
④ Write this block in H.D.

Buffer →



int x;
x ← item x
x + 50
x − 100
⋮

item x ← x

Buffer :-

→ Blocks from H.D are stored at a common place in RAM called buffer
→ All transactions will read & write data items in buffer only.
→ When transaction commits then respective block from buffer is stored in database.

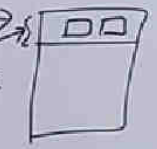
140. concept of read and write item

Read item(x) → Name of item (DB is a collection of named items)

① Get disk address of block that contain item x. (HOW?? implimentation dependent)

② Read in block (i.e transfer block from H.D. to RAM) [Why entire block?]

③ Copy value of item x from RAM into program variable x



int x;

x ← item x

x − 500;

x + 200;

Write item(x):-

① same

② same

③ Copy value of program variable x into block in RAM containing item x.

④ Write block in H.D. [This step make changes in Database]

T1 → r(x) → All 3 steps

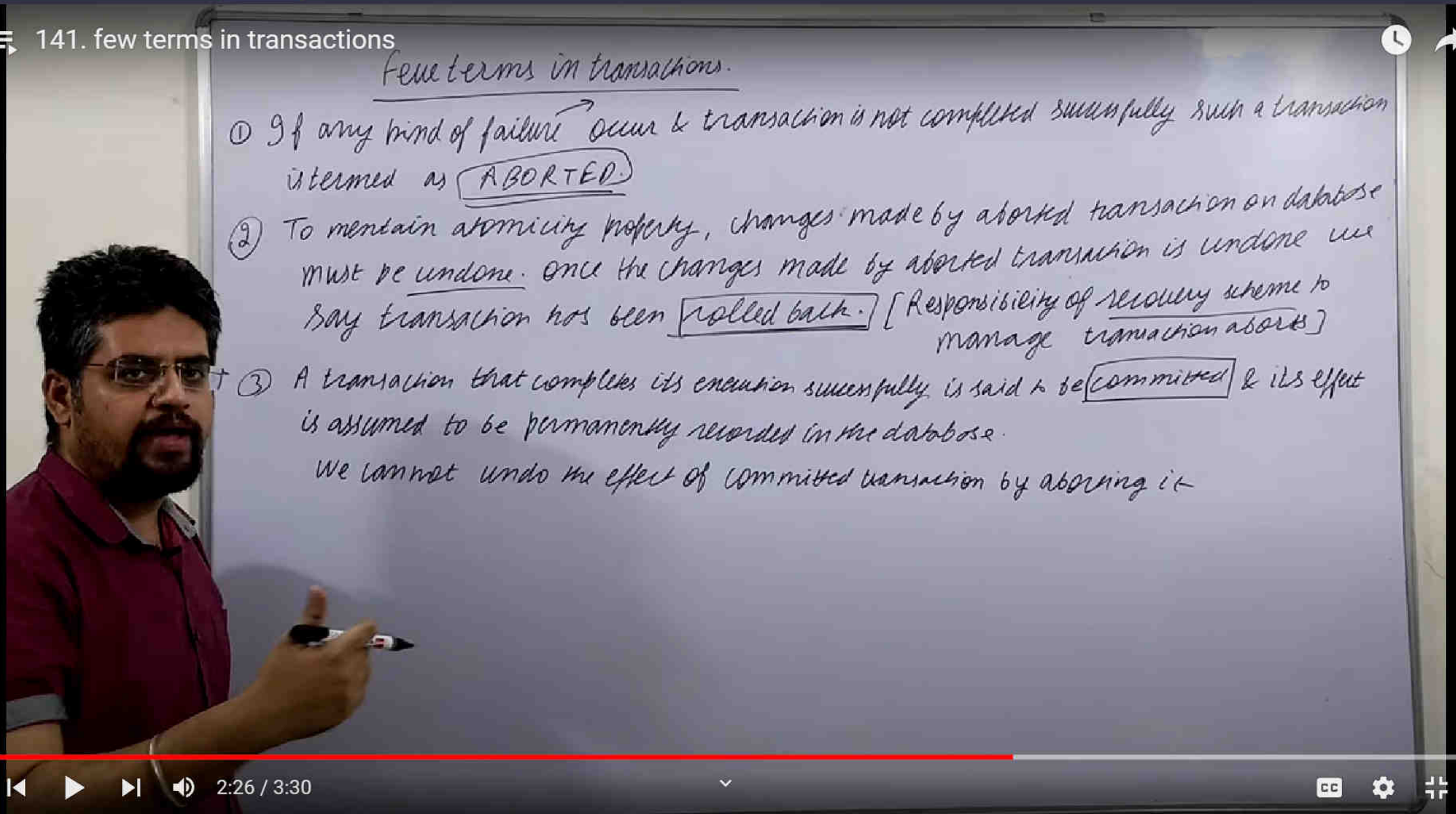
3 → T1 r(x) T2 r(x) → 3 ⇒ common buffer

r(x) → may or may not need all 3 steps

If we assume block containing item x is still in RAM then step 1 & 2 not needed

block not in RAM ∵ it has been replaced by some other block.

if T2 do r(x) immediately after r(x) of T1??



TRANSACTIONS

4 basic properties of transactions

1. Atomicity → Transaction will be either completely executed or not executed at all.
2. Consistency
3. Isolation
4. Durability

"ACID" properties

Transfer 500 Rs from account A to B

T1
R(A)
A = A - 500
W(A)
R(B)
B = B + 500
W(B)
commit

→ If before executing transaction database is in consistent state then after execution of transaction it must remain in consistent state only.

→ Consistent state means all constraints are fulfilled.

Eg:- A + B = 100

T2
R(A)
A = A + 10
W(A)
commit

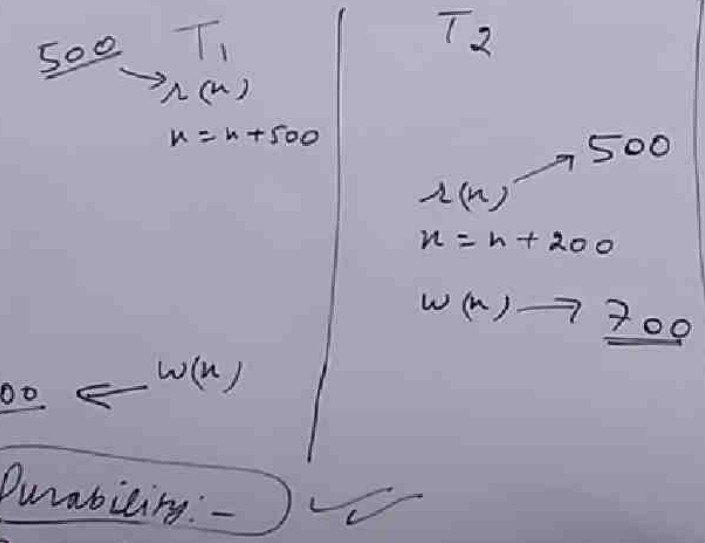
X

A + B = 110

TRANSACTIONS

Isolation:-

For every pair of transactions t_i & t_j it appears to t_j that either t_i has completed its execution before t_j starts or t_i will start its execution after t_j completes its execution.



Property	Responsibility
Atomicity →	Transaction management component of DBMS
Consistency →	Application programmer
Isolation →	Concurrency control component
Durability →	Recovery component.

Durability:-

→ Once transaction commits (i.e completed successfully) changes it has made to the database must persist even if there are some system failures.

143. why concurrency control is needed

Why concurrency control is needed?

→ We want to execute transactions in interleaved fashion so that their execution becomes faster

T_1	T_2
$r(x) \rightarrow$ I/O	$r(y) \rightarrow$ I/O
$x = x + 50 \rightarrow$ CPU	$y = y + 100 \rightarrow$ CPU
$w(x) \rightarrow$ I/O	$w(y) \rightarrow$ I/O

⇒ So transactions mainly do I/O operations & hence executing them in interleaved fashion will give more efficiency.

Few problems :-

→ Temporary update problem

Dirty read problem :-

→ If Transaction reads item written by uncommitted transaction then that read operation is called dirty read.

T_1	T_2
$r(x)$	
$w(x)$	
	$r(y)$
	$w(y)$

	T_1	T_2
500	$r(x)$	
700	$w(x)$	
		$r(y)$ Abort

T_2
$r(x)$ → 700
$w(x)$ → 1000
commit

T_1	T_2
$r(x)$	
$w(x)$	
	$r(x)$ → Dirty read.

w-r conflict RAW

T_1
$r(x)$
$w(x)$
commit

T_1	T_2
$r(x)$	
$w(x)$	
	$r(x)$ → DR

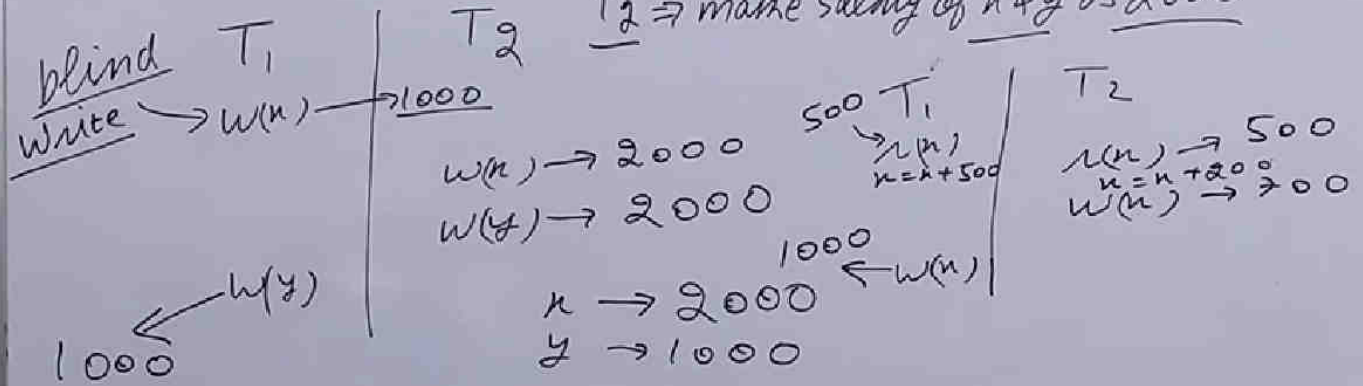
T_2
$w(x)$ → ✗

143. why concurrency control is needed

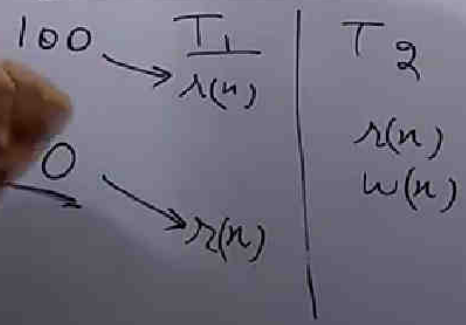
Why concurrency control is needed?

② Lost update problem:- $T_1 \Rightarrow$ make salary of x & y as 1000

$T_2 \Rightarrow$ make salary of x & y as 2000



③ Unrepeatable read problem:-



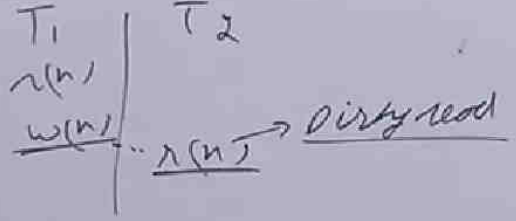
(4) Incorrect Summary problem.

T_1	T_2	T_3	T_4	T_5
R	y	x(2) x(2)	x(w) x(w)	Slock manager

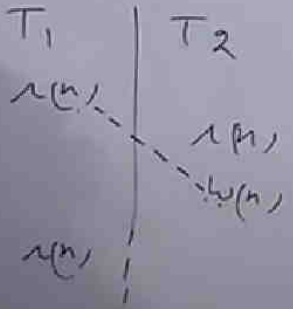
143. why concurrency control is needed

Summary

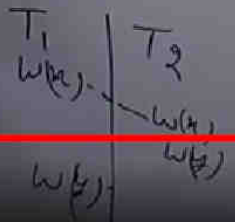
① Dirty read problem or Temporary update problem (W-R conflict) (RAW conflict)



② Unrepeatable read problem (R-W conflict) (WAR conflict)



③ Lost update problem or Overwriting uncommitted data (W-W conflict) (WAW conflict)



144. schedule

Concurrency control techniques:-

→ CCT are used to ensure consistency of database when transactions are executed concurrently or in interleaved fashion [final answer must be correct].

Few terms:-

① Schedule:-

Set of transactions that are executed together in the system

Sa

T_1	T_2
r(x)	
	r(y)
w(x)	
	w(y)

Time

Sb

T_1	T_2	T_3
r(x)		
	r(y)	
		w(x)
w(y)		
	w(z)	

Sa: r1(x); r2(y); w1(x); w2(y)

Sb: r1(x); r2(y); w3(x); w1(y); w2(z)

145. Types of schedules and intro to serial schedule

Types of schedules
- Serial Schedule
- Serializable ...
- Conflict equivalent schedule
- Conflict Serializable ....
- View Equivalent ....
- View Serializable .... —
- Recoverable schedule —
- Cascadeless " " —
- Strict " " —

① Serial schedule :-

T_1	T_2
r(n)	
w(n)	
commit	
	r(n)
	w(n)
	commit

$T_1 \rightarrow T_2$ ✓ ; $T_2 \rightarrow T_1$ ✗

T_1	T_2
r(n)	
	r(y)
w(n)	
abort	
	w(y)

T_1	T_2
r(n)	
w(n)	
	r(y)
	w(y)
	commit

n * n-1 * n-2 ... 1 = n!

→ A schedule S is said to be serial if no transaction starts till running transaction ends [commit or abort]

→ So only one transaction is active at a time & commit or abort of active transaction initiates execution of next transaction

→ If we consider each transaction to be correct (i.e consistency property holds) then all possible serial schedules (n!) are correct.

→ main disadvantage of serial schedules is that if transaction waits for an I/O then CPU cannot be switched to another transaction. Hence lot of CPU cycles are wasted.

I/O

T_1	T_2
r(n)	
	r(z)

3 * 2 * 1 = 6 ; n n-1 n-2 ..... 1 = n!

146. serializable schedule

Types of schedules

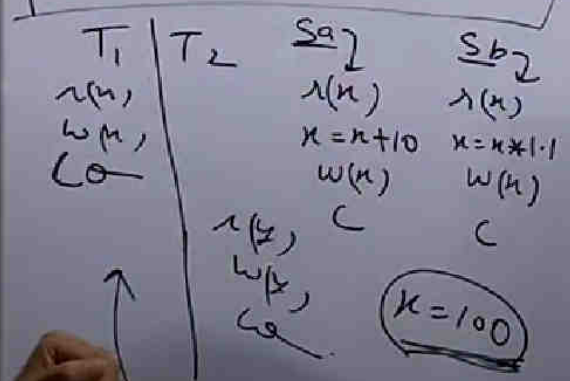
- Serial schedule
- Serializable ... ✓
- conflict equivalent schedule
- Conflict Serializable ....
- View Equivalent ....
- View serializable .... ✓
- Recoverable schedule —
- Cascadeless " " —
- strict " " —

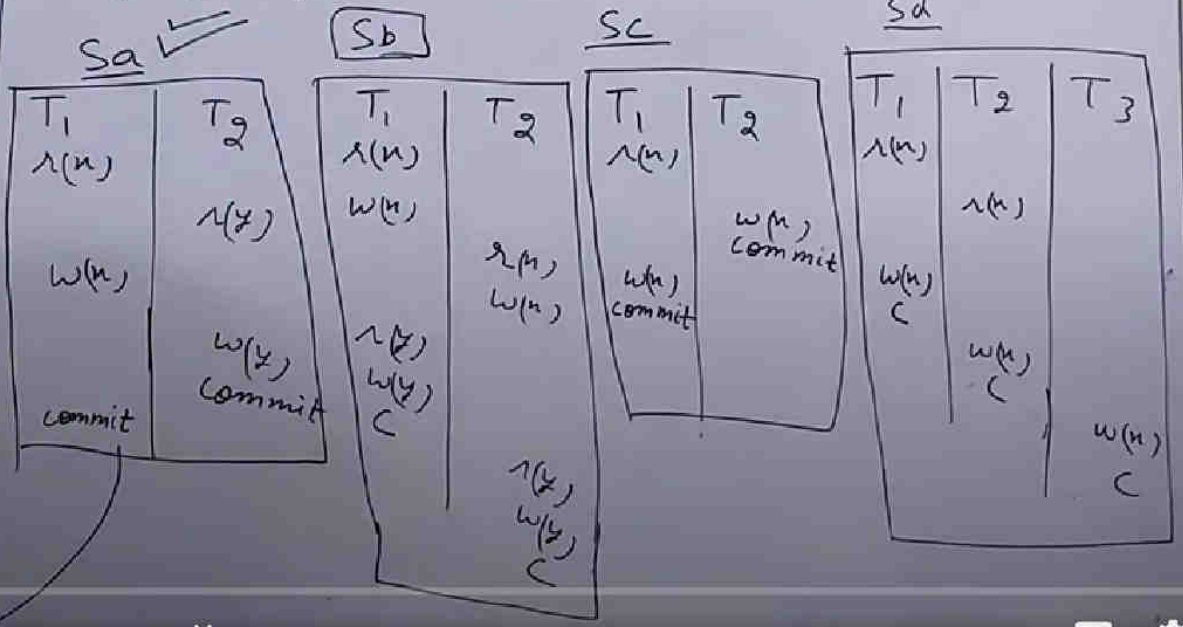
Serializable schedule

→ A schedule S is said to be serializable if its effect on database is guaranteed to be same as some serial schedule.

→ This means if we have a schedule of 'n' transactions then its result must be same as one of the n! possible serial schedules.

→ From defⁿ we can easily say that every serial schedule is serializable but vice versa not true.





146. serializable schedule

Types of schedules

- Serial Schedule
- Serializable ... ✓
- Conflict equivalent schedule
- Conflict Serializable .... ✓
- View Equivalent ....
- View Serializable .... ✓
- Recoverable schedule ✓
- Cascadeless " " ✓
- Strict " " ✓

Serializable schedule

→ A schedule S is said to be serializable if its effect on database is guarenteed to be same as some serial schedule.

→ This means if we have a schedule of 'n' transactions then its result must be same as one of the n! possible serial schedules.

→ Fr... can easily say that every serial schedule is ... but vice versa not true.

$T_1 \to T_2 \to T_3$

$T_2 \to T_1 \to T_3$

Sa ✓ Sb ✓ Sc Sd

146. serializable schedule

Types of schedules
- Serial Schedule
- Serializable ... ✓
- Conflict equivalent schedule
- Conflict Serializable .... ✓
- View Equivalent ....
- View Serializable .... ✓
- Recoverable schedule ✓
- Cascadeless " " ✓
- Strict .... ✓

Serializable schedule

→ checking schedule is serializable or not is an undecidable problem i.e no one in this world will ever be able to write algorithm for this problem.

Non serial schedules (>>> n!)

Correct/Serializable

Incorrect/Non serializable

A, B, E, D

$\frac{4!}{2!2!}$

ABCD } ⇒ 1
BACD }

Serial ⇒ n Transaction → n!

T_1	T_2
r(x)	
w(x)	
	r(y)
	w(y)

⇒ 2!

n n-1 n-2 — — ⇒ n! ✓

$\frac{4!}{2!2!} \Rightarrow \frac{4\times3\times2\times1}{2\times2} = 6$ ⇒ Total No. of Schedules

6 − 2! ⇒ 4 ✓

$T_1 \to n$
$T_2 \to m$

$\frac{(n+m)!}{n!*m!}$

$\frac{(n_1+n_2+n_3\ldots n_n)!}{n_1!\,n_2!\,n_3!\ldots n_n!}$

146. serializable schedule

Types of schedules

- Serial schedule
- Serializable ...
- Conflict equivalent schedule
- Conflict Serializable ....
- View Equivalent ....
- ...u Serializable ....
- ...overable schedule
- ...delen " "
- ... " "

Serializable schedule

Sc

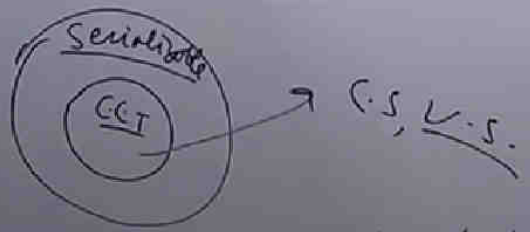
T_1	T_2
100 → R(n)	
	W(n) → 500 abort
110 ← W(n) commit	

$T_1 \rightarrow T_2$

$T_2 \rightarrow T_1$

→ C.C.T must guarantee serializable schedules.

→ But checking, schedule is serializable or not is an undecidable problem. So, we have other types of schedules which are subset of serializable schedules that can be checked by C.C.T.



→ C.C.T allow only correct schedules to be executed in the system. But all correct schedules are not executed.

146.2 gate 2012 question on serializablity

T_1 : R(P)
R(Q)
if P=0 then Q = Q+1
W(Q)

T_1 Q=1

Q
0

T_2 : R(Q)
R(P)
if Q=0 then P = P+1
W(P)

Any non serial interleaving of T_1 & T_2 leads to

$T_1 \rightarrow T_2 \Rightarrow Q=1, P=0$

$T_2 \rightarrow T_1 \Rightarrow P=1, Q=0$

② Before T_2 last operation 1st operation of T_1 is done $\Rightarrow$ P=1, Q=1

GATE 2014

T_1	T_2	T_3	T_4
	R(x)		
		W(x)	
		Commit	
W(x)			
Commit			
	W(y)		
	R(z)		
	Commit		
			R(x)
			R(y)
			Commit

Conflict serializable?
Recoverable?

147. conflict equivalent schedule

Conflict Equivalent Schedules

minimum 2 schedules

Conflicting operations

T_1	T_2
$r(x)$	
	$r(x)$

T_1	T_2
$r(x)$	
	$w(x)$

T_1	T_2
$w(x)$	
	$r(x)$

T_1	T_2
$w(x)$	
	$w(x)$

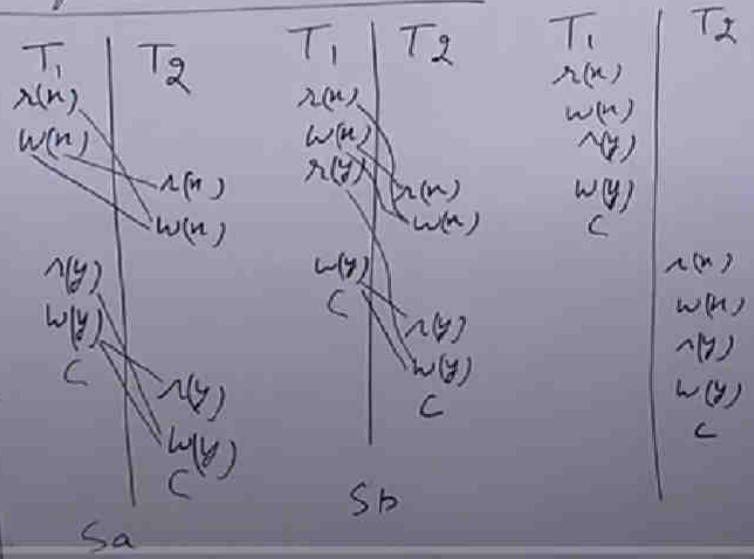
T_1
$r(x)$
$w(x)$

2 operations are said to be conflicting if following 3 cond" are satisfied :-

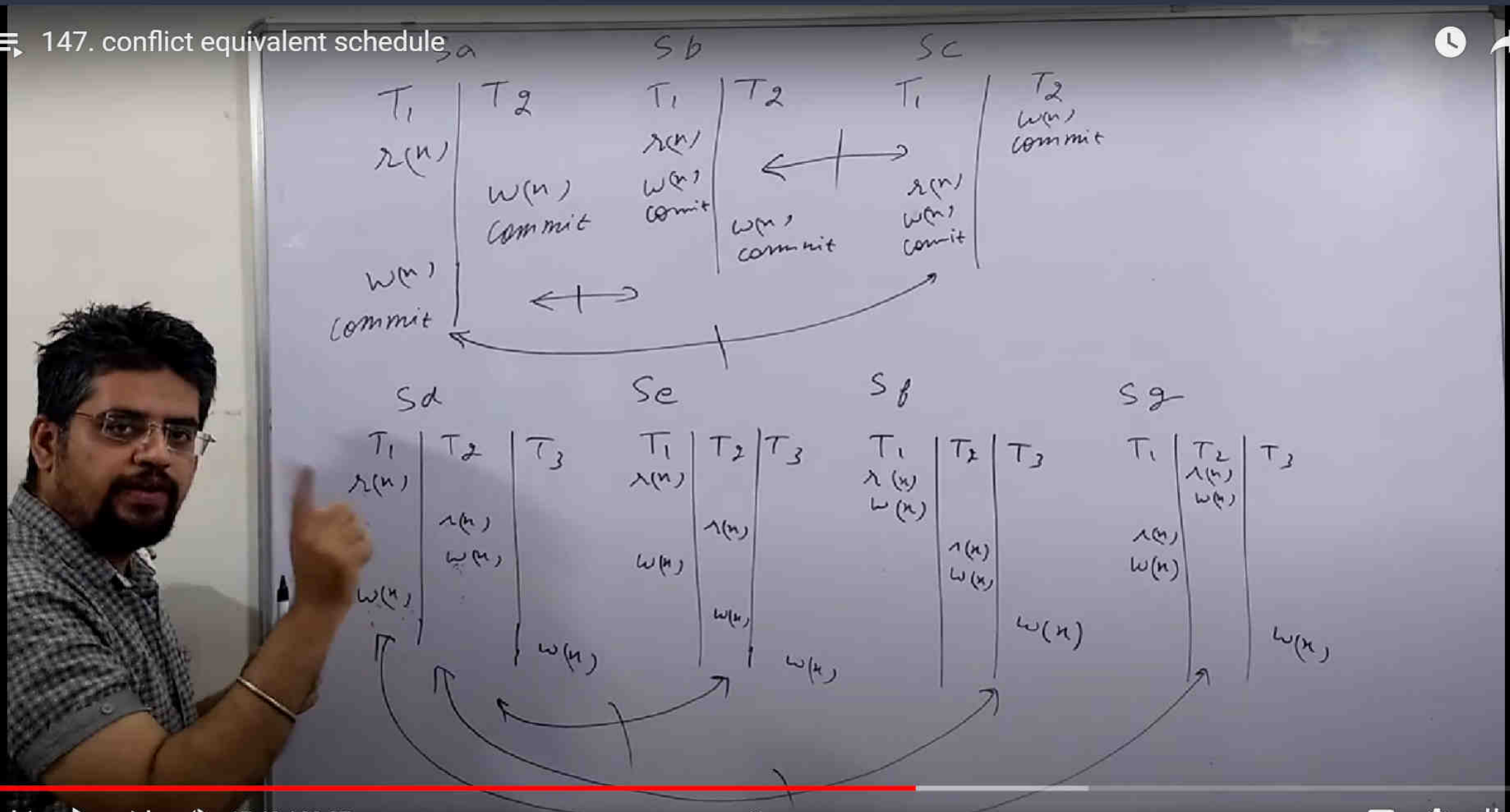
1. They belong to different transactions
2. They access same database item
3. Atleast one of them must be write operation

Def^n: 2 schedules are said to be conflict equivalent if order of every conflicting operation is same in both schedules

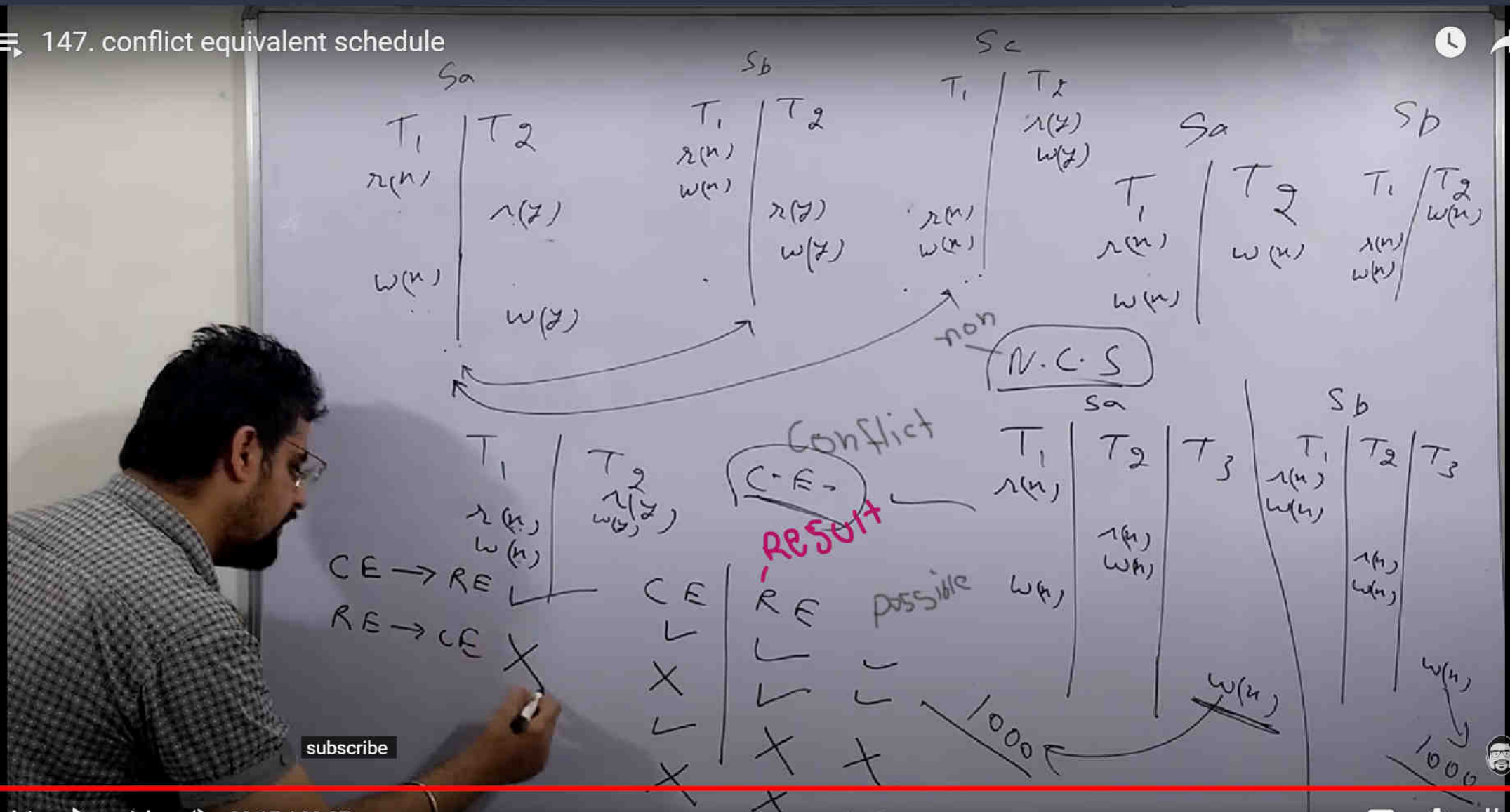
Def^n: 2 schedules are said to be conflict equivalent if we are able to construct one schedule from another by series of non conflicting swaps.



147. conflict equivalent schedule



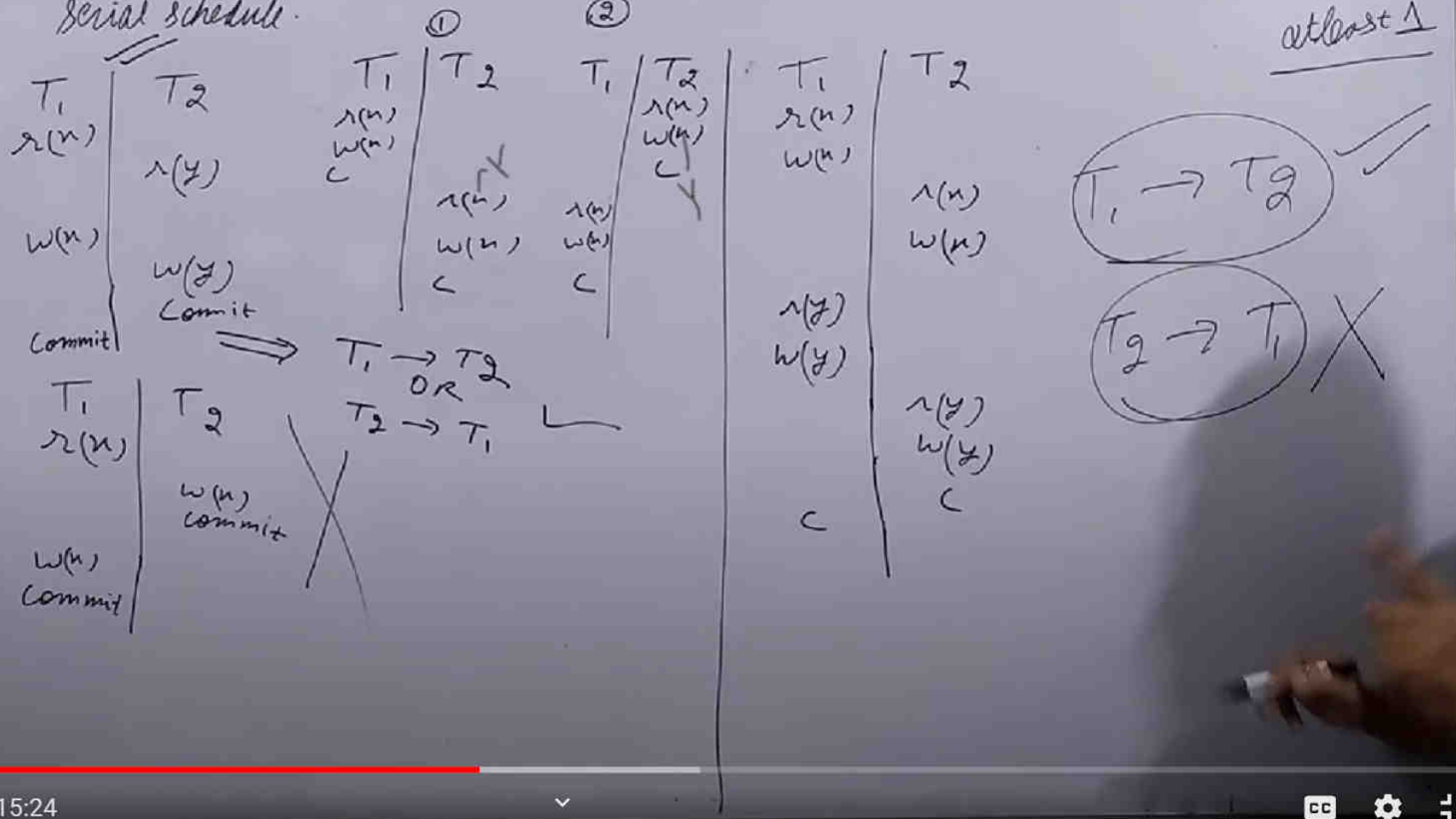
147. conflict equivalent schedule



148. conflict serializability

Conflict Serializable schedule

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



atleast 1

148. conflict serializability

Conflict Serializable schedule

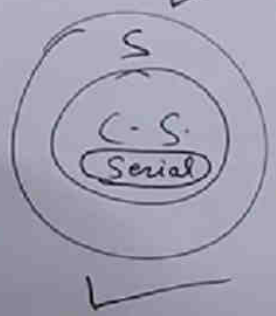
① A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule ✓

② A schedule S is said to be serializable if it is result equivalent to some serial schedule. ⇒ guarantee

③ C.E → R.E ✓
R.E → C.E ✗

Conclusion ??

RESULT



C.S. ⇒ Serializable

R.E. → C.E. ✗

T_1	T_2
r(A), w(A), Commit	r(A), w(A), Commit

T_1	T_2	T_3
r(A)		
	r(A)	
	w(A)	
w(A)		

$T_1 \rightarrow T_2 \rightarrow T_3$ ✗
$T_2 \rightarrow T_1 \rightarrow T_3$ ✗

C.J

R.E ✓

w(A) reason

148.1 number of conflict serializable schedules (GATE 2017)

GATE 2017

$T_1 \rightarrow r(x), w(x), r(y), w(y)$

$T_2 \rightarrow r(y), w(y), r(z), w(z)$

$T_2 \rightarrow T_1$

$T_1 \rightarrow T_2$

Total No. of C.S. schedules ??

5 4

5C2 + 5C1
4C2 + 4C1
3C2 + 3C1
4C2 + 4C1
3C2 + 3C1
3C2 + 3C1
+
1

T2 | T1
— | r(x)
— | w(x)
r(z) |
— | w(z)
r(x) |
w(x) |
w(z) |

149. how to check conflict serializability

Q. How to check schedule is conflict serializable or not?

Algorithm:-

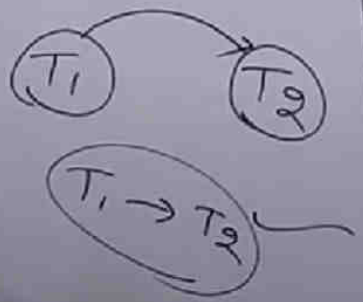
Step 1 Each transaction is represented as node of graph (Precidence graph)

Step 2 If we have $r(x)$ operation in T_i then try to find $w(x)$ operation in T_j below $r(x)$ if we find the same then draw directed edge from T_i to T_j only once for all conflicting operations.

Step 3 If we have cycle in this graph then schedule is not C.S. else it it is C.S. & serializability order is given by topological ordering of nodes of graph.

Q1

T_1	T_2
$r(x)$	
$w(x)$	
	$r(x)$
	$w(x)$
$r(y)$	
$w(y)$	
	$r(y)$
	$w(y)$

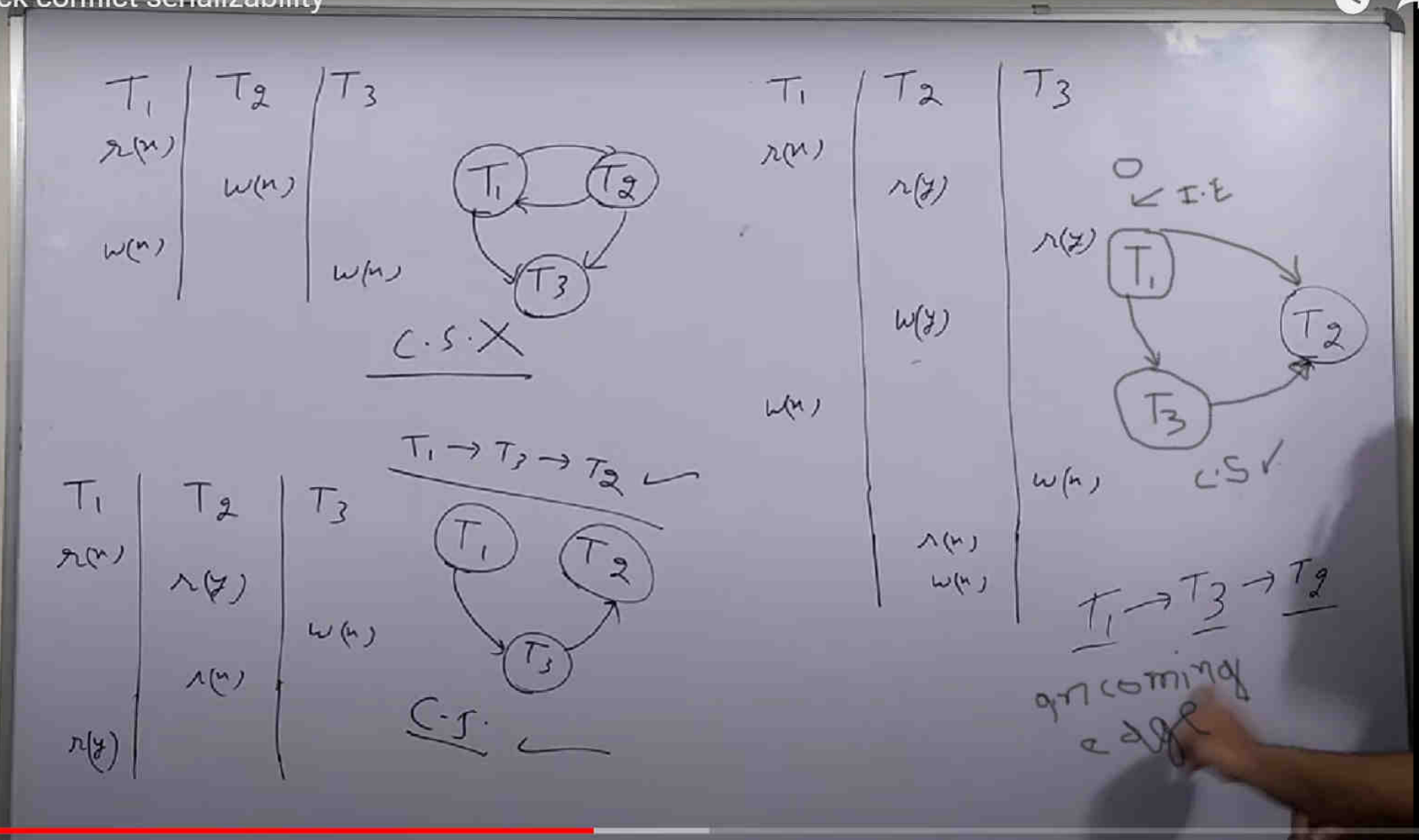


Q2

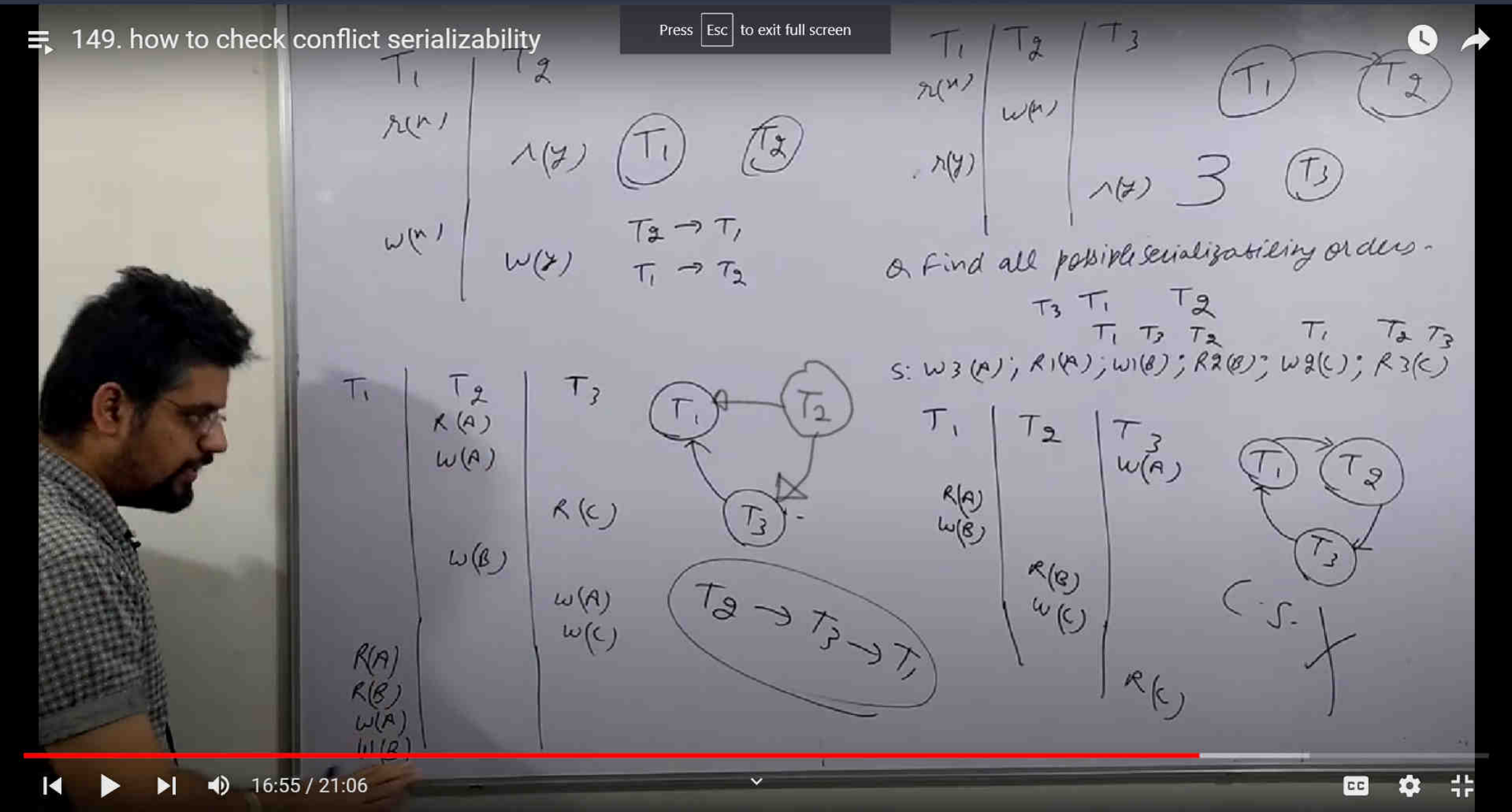
T_1	T_2
$r(x)$	
	$w(x)$
$w(x)$	

CS X

149. how to check conflict serializability

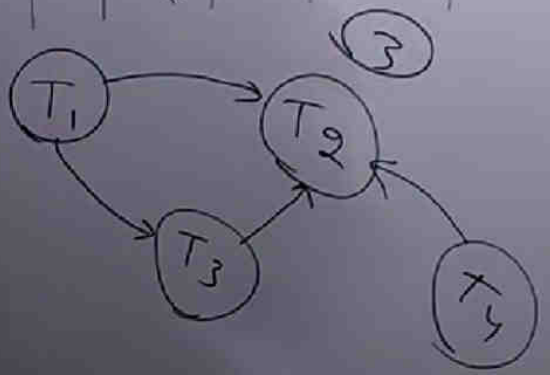


149. how to check conflict serializability



149. how to check conflict serializability

T_1	T_2	T_3	T_4
			$r(A)$
	$r(A)$		
		$r(A)$	
$w(B)$			
	$w(A)$		
		$R(B)$	
	$w(B)$		



⇒ To detect cycle in graph having n vertices we need $O(n^2)$ time.

→ Hence checking schedule is C.S. or not is a P class problem i.e fast algo exists for its checking

→ Most of C.C.T are based on concept of conflict serializability

→ But every correct schedule (serializable) is not C.S.

→ C.C.T based on C.S. will reject those schedules which are not C.S.

→ Hence lots & lots of correct schedules will not be encoured in our system ∴ they are not C.S.

T_1	T_2	T_3
$r(x)$		
	$w(x)$	
$w(x)$		
		$w(x)$

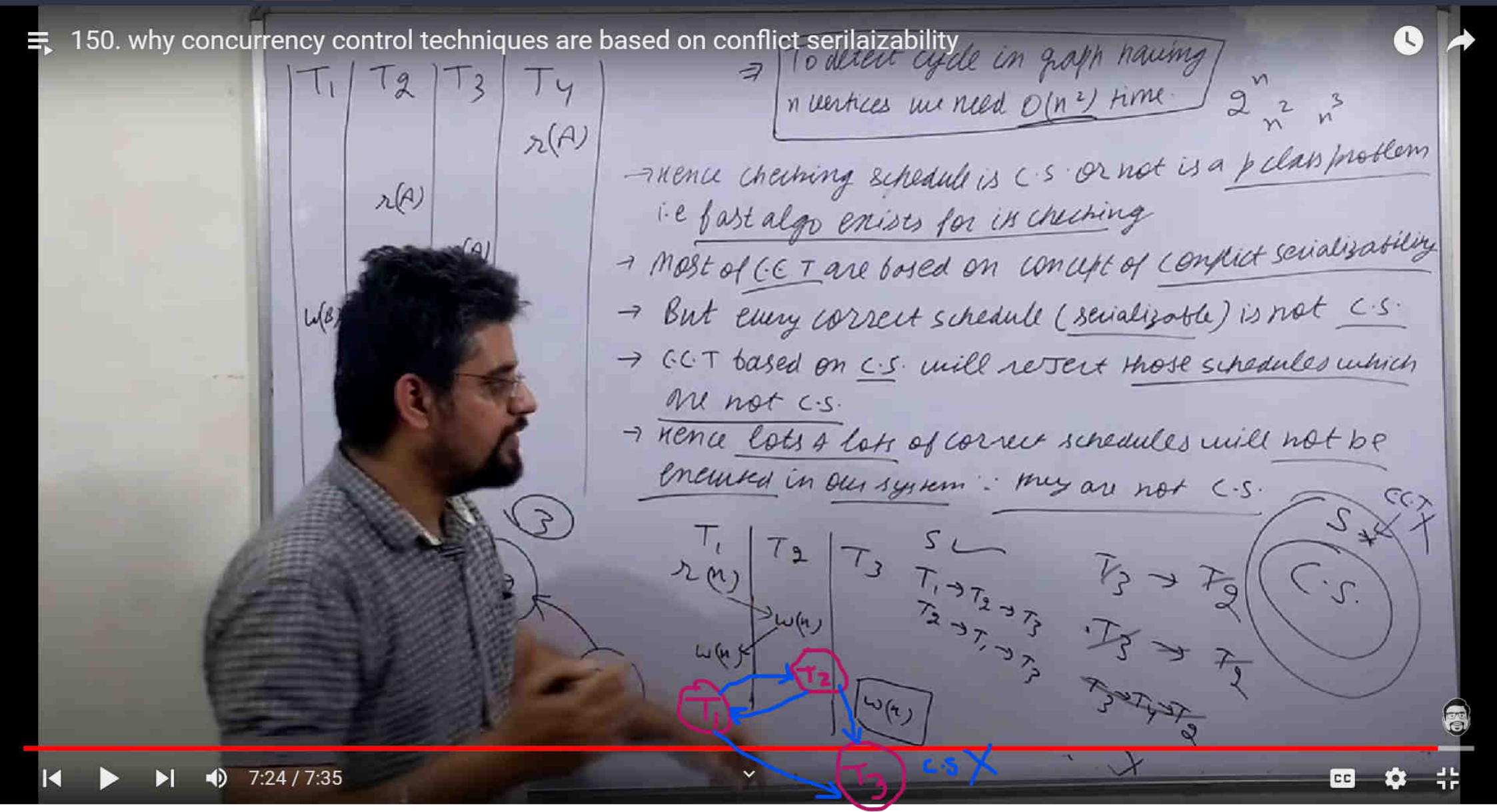
$T_4 \to T_1 \to T_3 \to T_2$

$T_1 \to T_4 \to T_3 \to T_2$

$T_1 \to T_3 \to T_4 \to T_2$

~~$T_1 \to T_2 \to T_4$~~ ×

150. why concurrency control techniques are based on conflict serilaizability



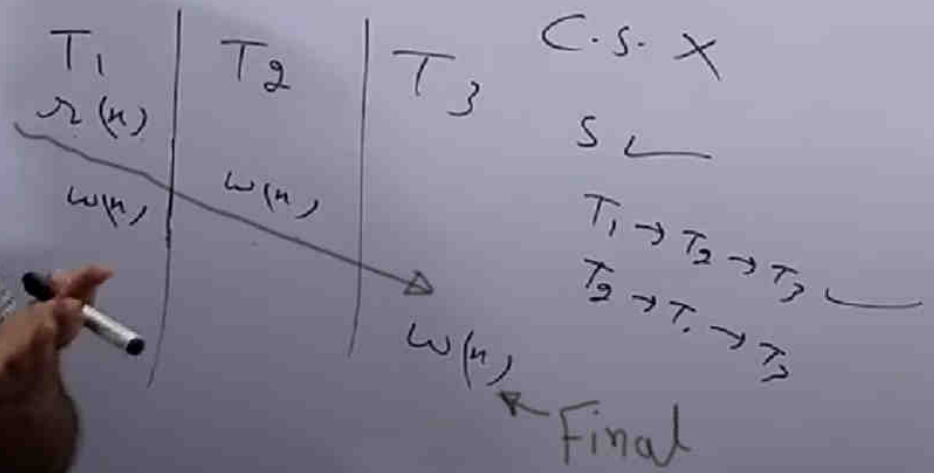
151. why concept of view serializability

→ Definition of C.S is very restrictive : here w-w is always a conflict

→ But w-w is a conflicting operation only if one of w is final write

→ Hence we have other type of schedule called View serializable

→ According to def^n of View equivalence we have only 2 conflicts i.e r-w & w-r (w-w is conflict only if one of w is final write)



152. view equivalent schedules

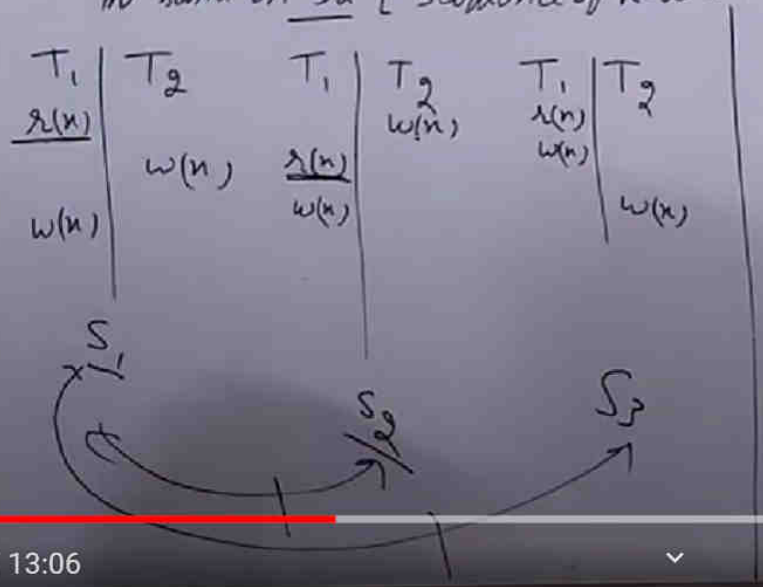
View Equivalent schedules

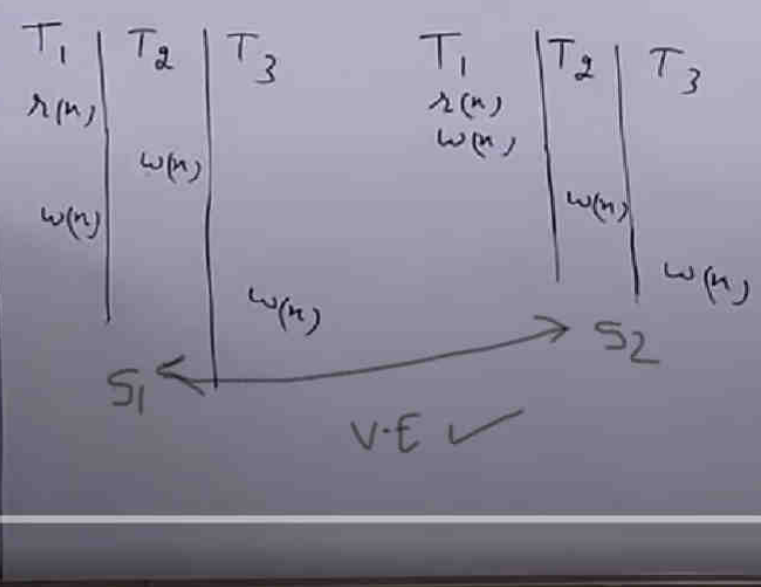
→ 2 schedules S1 & S2 over same set of transactions are view equivalent if following 3 conditions are satisfied. → original

① If T_i read initial value of item x in S1 then it must also read initial value of item x in S2.

② For each data item x the transaction that perform final write on x in S1 must also perform the final write on x in S2

③ If transaction T_i reads value of item x written by T_2 in S1 then it must also do the same in S2 [Sequence of R-W & W-R conflicts must be maintained]





152. view equivalent schedules

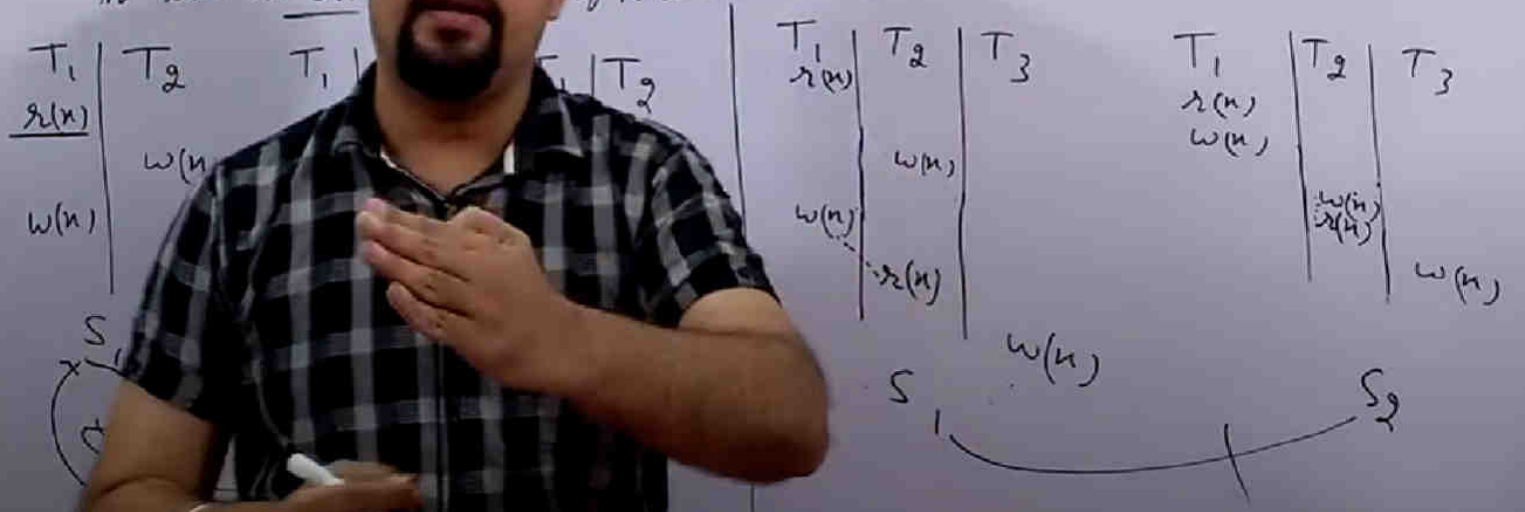
View Equivalent schedules

→ 2 schedules S1 & S2 over same set of transactions are view equivalent if following 3 conditions are satisfied.

① If T_i read initial value of item x in S1 then it must also read initial value of item x in S2.

② For each data item ... transaction that perform final write on x in S1 must also perform the ... on x in S2

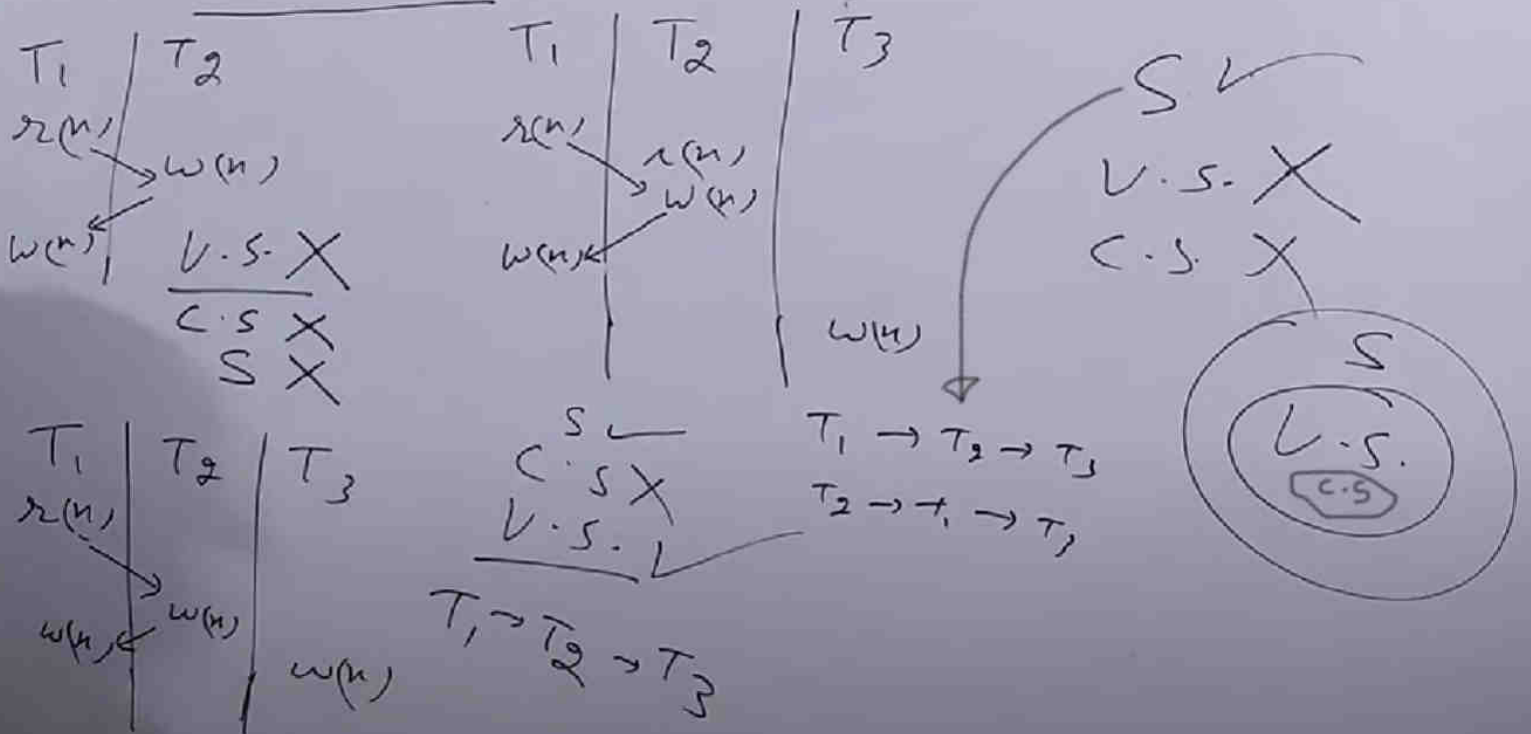
③ If transaction ... value of item x written by T_2 in S1 then it must also do the same in S2 ... of R-W & W-R conflicts must be mentained]



153. view serializability

View Serializable schedule :-

A schedule S is said to be view serializable if it is view equivalent to some serial schedule.

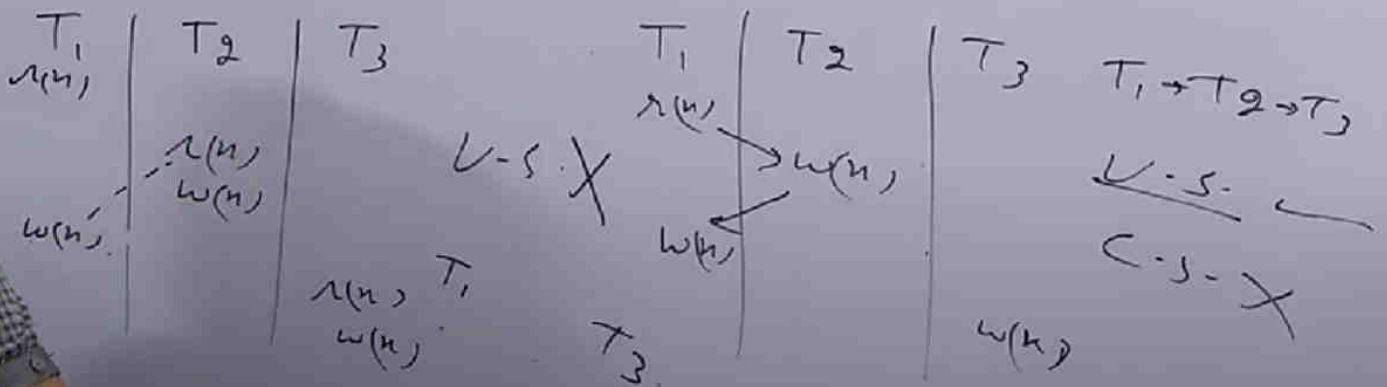


153. view serializability

→ If we have constraint write assumption then blind writes are not allowed.
i.e w/o reading transaction cannot write any item.

→ Under such assumption definition of C.S. & V.S. are exactly same.

→ But if we have unconstrained write assumption i.e blind writes are allowed then definition of C.S & V.S. are different.



154. how to check view serializability

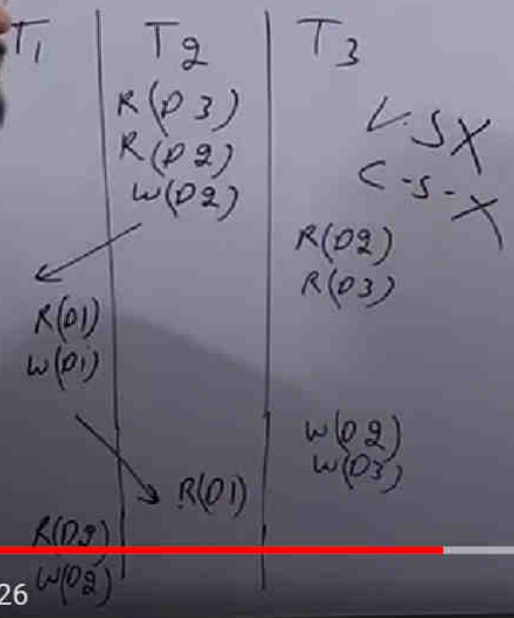
How to check schedule is view serializable or not?

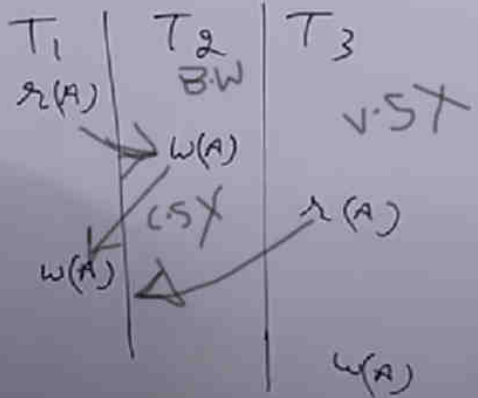
R(A)
W(A)

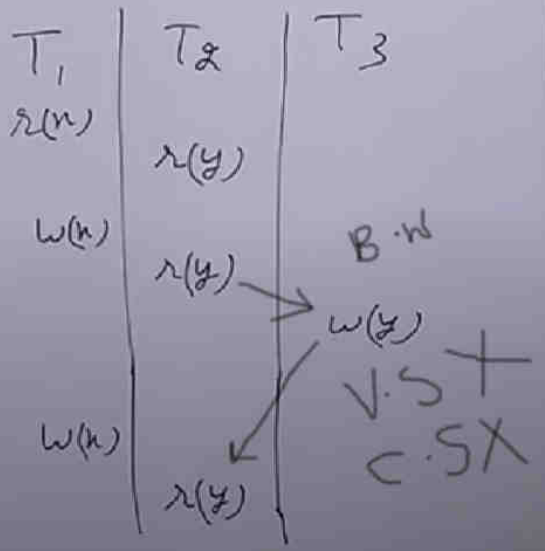
(1) check for C.S. If C.S. then V.S.

(2) If schedule is not C.S. then check for blind writes if No blind writes then it will not be V.S. [∵ def^n of C.S. & V.S. is same in constraint write assumption]

(3) If schedule is not C.S. & it has blind writes then apply def^n of V.S.







155. recoverable schedule

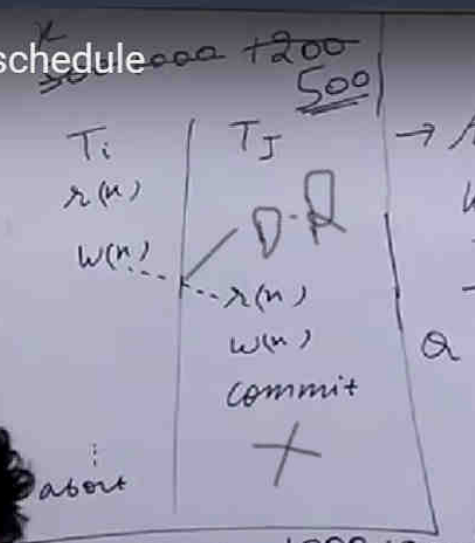
RECOVERABLE SCHEDULE

→ A schedule is said to be recoverable if transaction T_J reads an item written by uncommitted transaction T_i then T_J must not commit before T_i (commit operation of T_i must appear before commit operation of T_J)

Q How to check schedule is recoverable or not?

① check presence of dirty read if dirty read is absent then schedule is always recoverable.

② If dirty read is present then transaction doing dirty read must not commit first.



500, 1000 1000, 1200

Q1 T_1 R(n) W(n) | T_2 R(n) W(n) commit

Q2 T_1 R(n) W(n) | T_2 R(n) W(n)

Q3 T_1 R(n) W(n) | T_2 W(n)

Q4 T_1 R(n) W(n) abort | T_2 R(n) W(n) commit

Q5 T_1 R(n) W(n) commit | T_2 R(n) W(n)

155.1 gate 2016 question on recoverability

GATE 2016

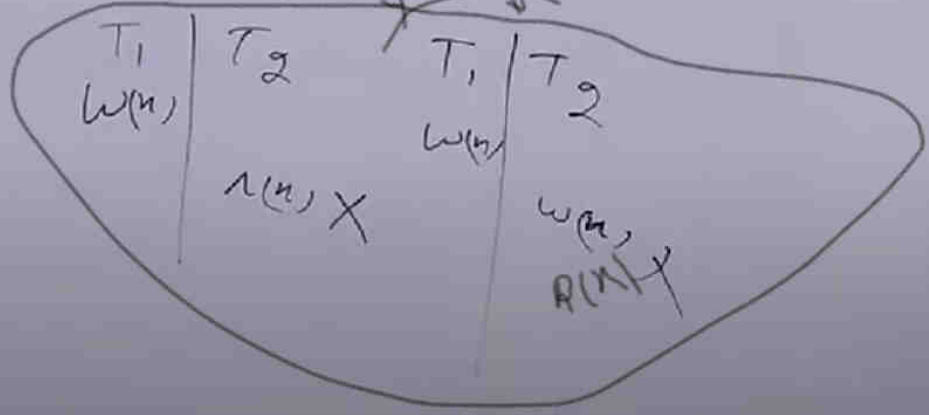
$S = r2(x); r1(x); r2(y); w1(x); r1(y); w2(x); a1; a2$

Recoverable ← D.R → No, commit 1st

Cascadless (No cascading abort) ← D.R X

T_1	T_2
	$r(x)$
$r(x)$	
	$r(y)$
$w(x)$	
$r(y)$	
	$w(x)$
abort	
	abort

strict X



156. cascadeless schedule

Cascadeless Schedule:-

→ Failure of one transaction leads to failure of series of transactions.

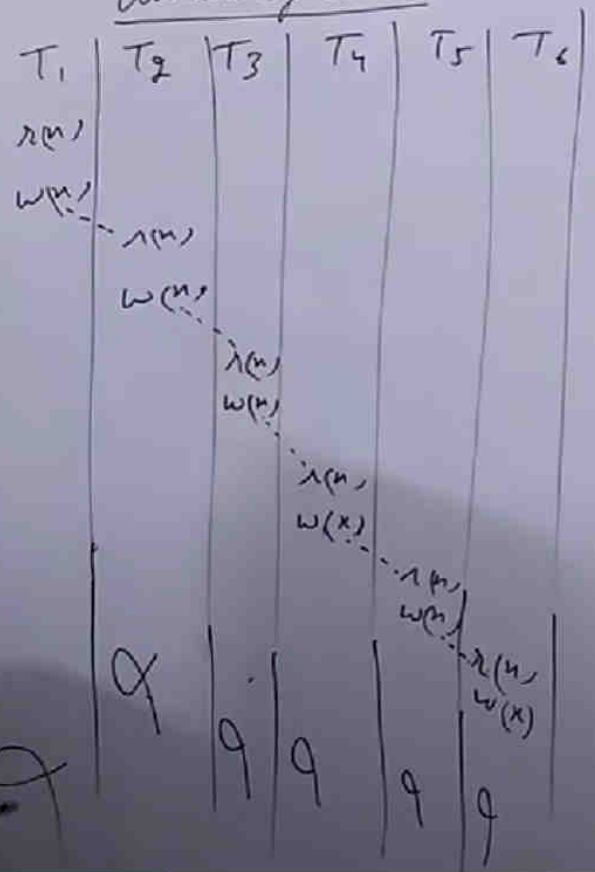
For eg. if T_1 fails then T_2, T_3, T_4, T_5, T_6 must be aborted

→ A schedule is said to be cascadeless if it avoids cascading aborts.

→ Main reason of cascading abort is dirty read so cascadeless schedules must not allow Dirty read

→ In other words, A schedule is said to be cascadeless if transaction reads data written by committed transaction only.

Cascading abort:-



156. cascadeless schedule

How to check schedule is cascadeless or Not?

→ If dirty present then schedule is not cascadeless else it is cascadeless

→ "No Dirty read" is necessary as well as sufficient cond^n for cascadeless schedule

→ "No dirty read" is only sufficient but not necessary cond^n for schedule to be recoverable

R → recoverable ⇒ ??

C → "No dirty read" ⇒ F

recoverable

T_1	T_2
r(n)	
w(n)	
	r(n)
	w(n)
Commit	

R ✓ C ✗

Q2

T_1	T_2
r(n)	
w(n)	
	C ✓
	R ✓
	w(n)

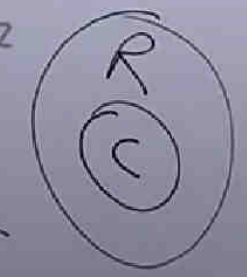
Q3

T_1	T_2
r(n)	
w(n)	
	r(n)
	w(n)
	commit

R ✗ C ✗

R ✓ C ✗ R ✗ C ✓ R ✗ C ✗

Possible



Strict Schedule:-

Q Every cascadless schedule must be allowed in database??

T_1	T_2
W(x)	
	W(x)
abort	

cascadless ✓
recoverable ✓

x: 50 ~~100~~ ~~150~~ 50

T_1	T_2
new W(x)	
	new X
	W(x) X

→ Hence we need another type of schedule which are stricter than cascadless schedules & are known as strict schedules.

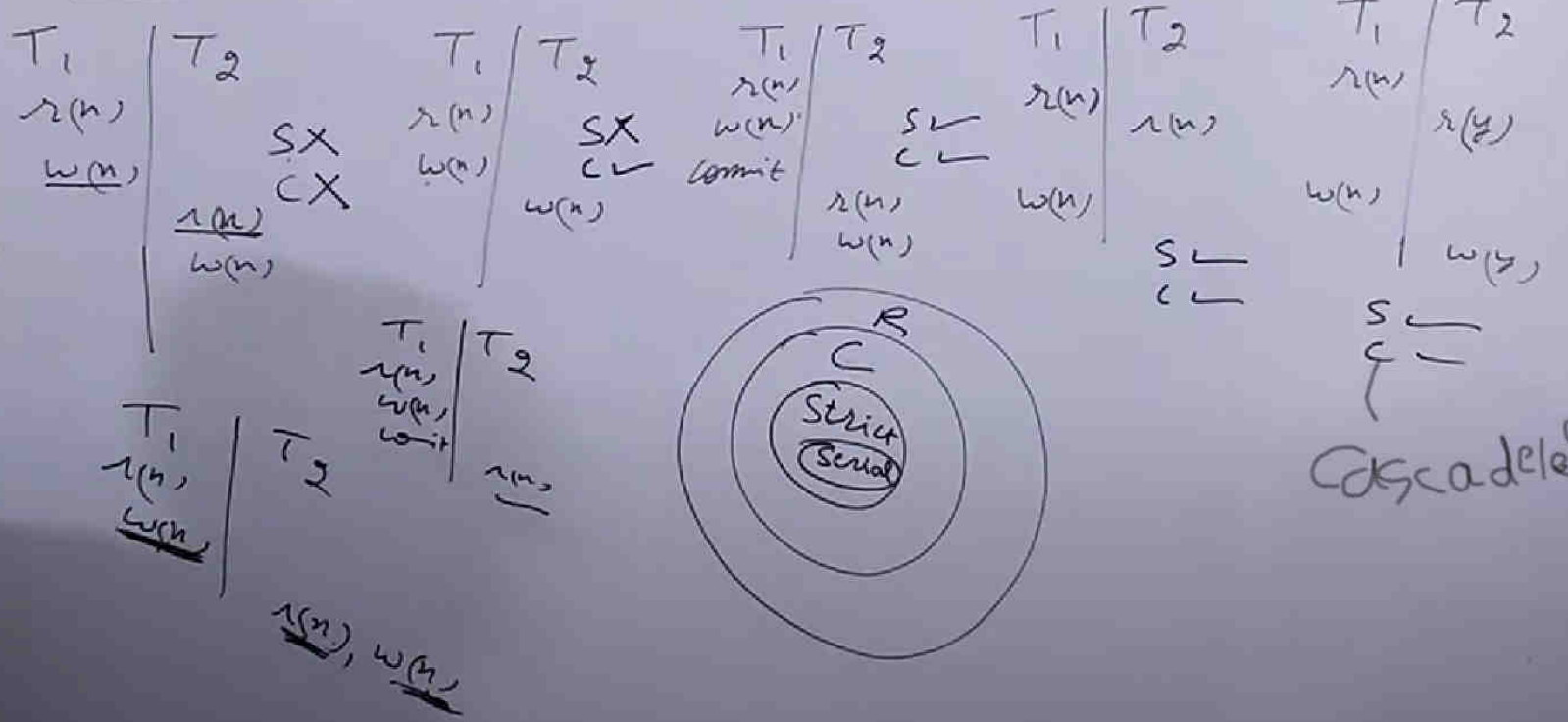
→ In cascadless schedules transactions were not allowed to read any item written by uncommitted transactions (dirty read)

→ But in strict schedules transactions are not allowed to even write any item written by uncommitted transaction

Defⁿ ⇒ In a strict schedule, transactions can neither read nor write an item x until the last transaction that wrote x has committed or aborted.

How to check Schedule is strict or not?

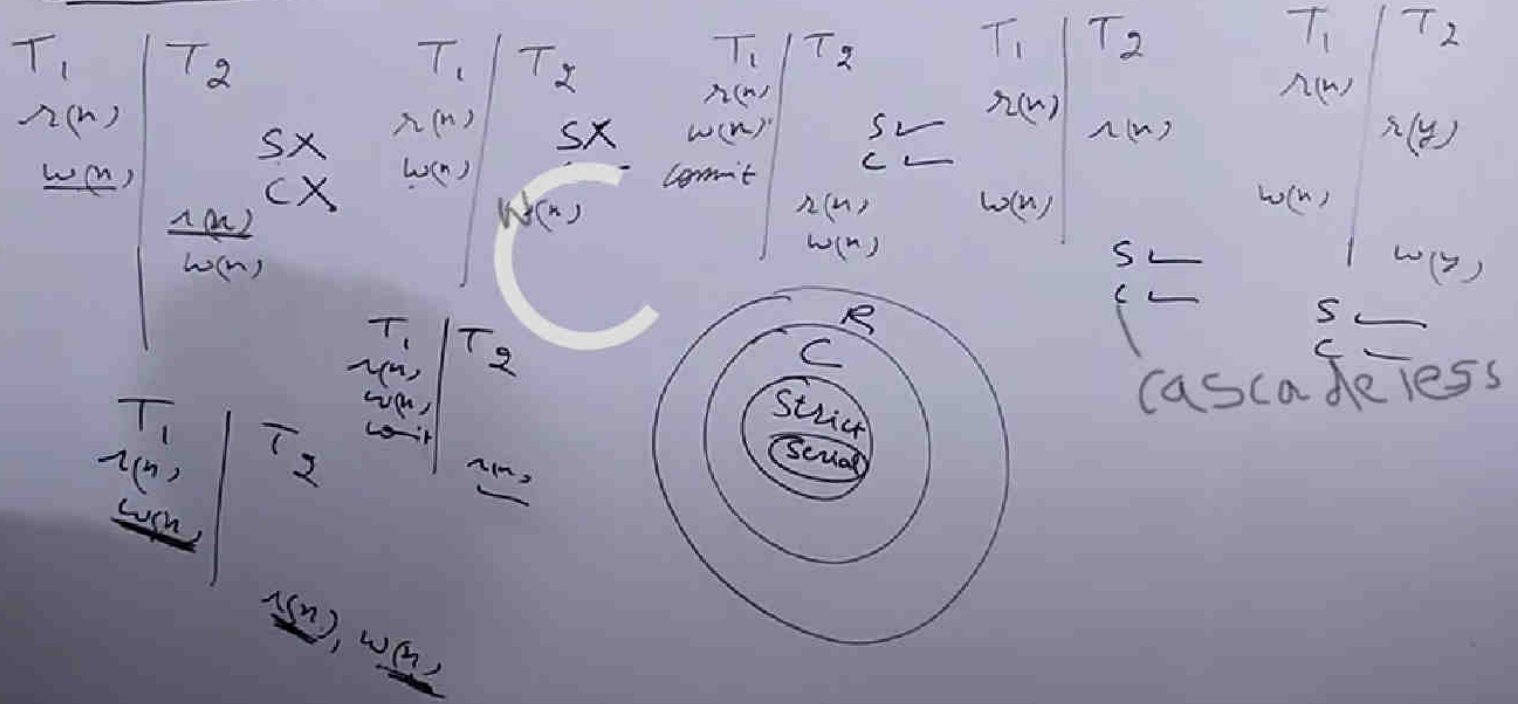
→ If write operation is proceeded by any conflicting operation (ie read or write) then that conflicting operation must be done after transaction doing write operation (commit or abort).

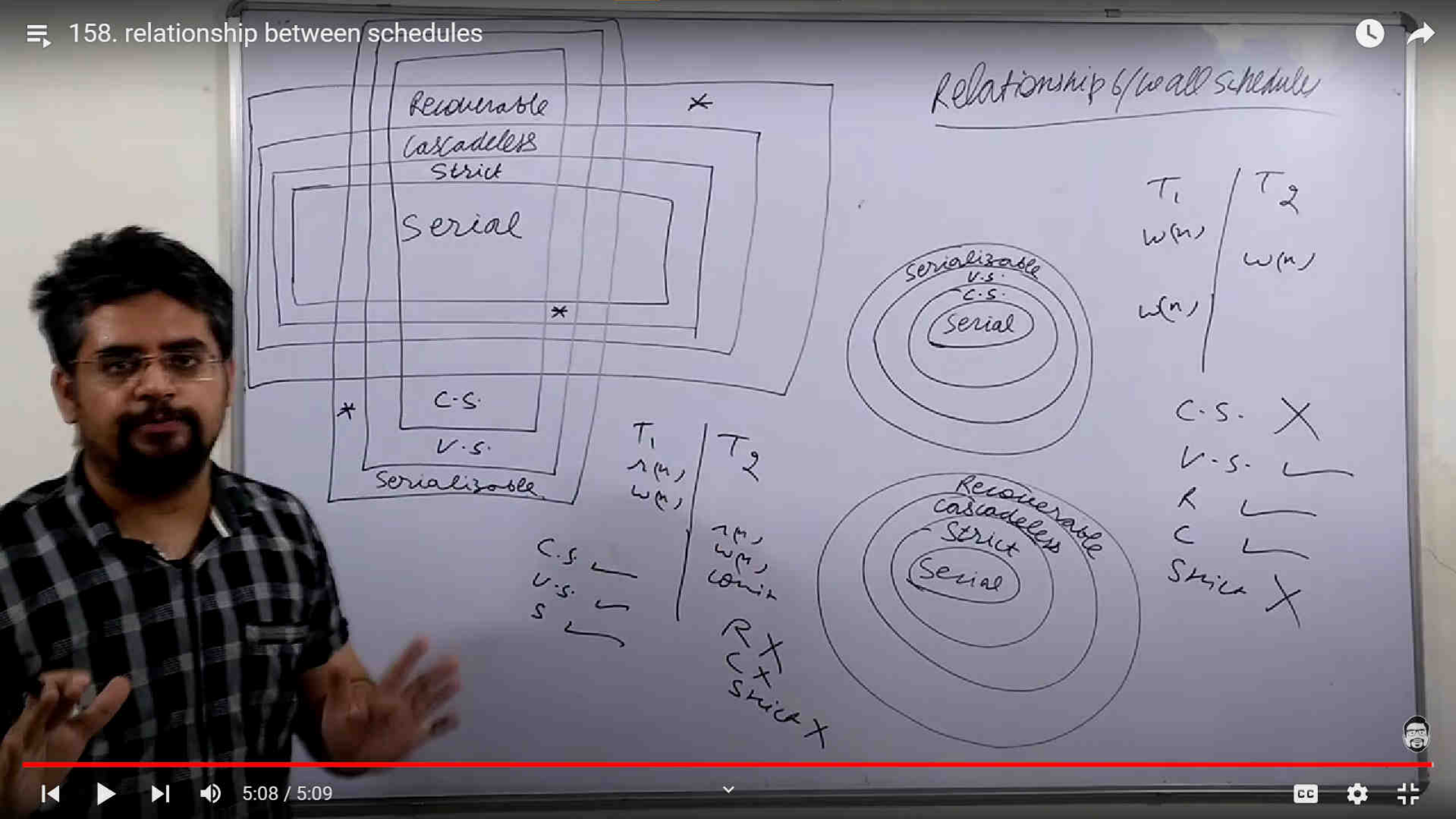


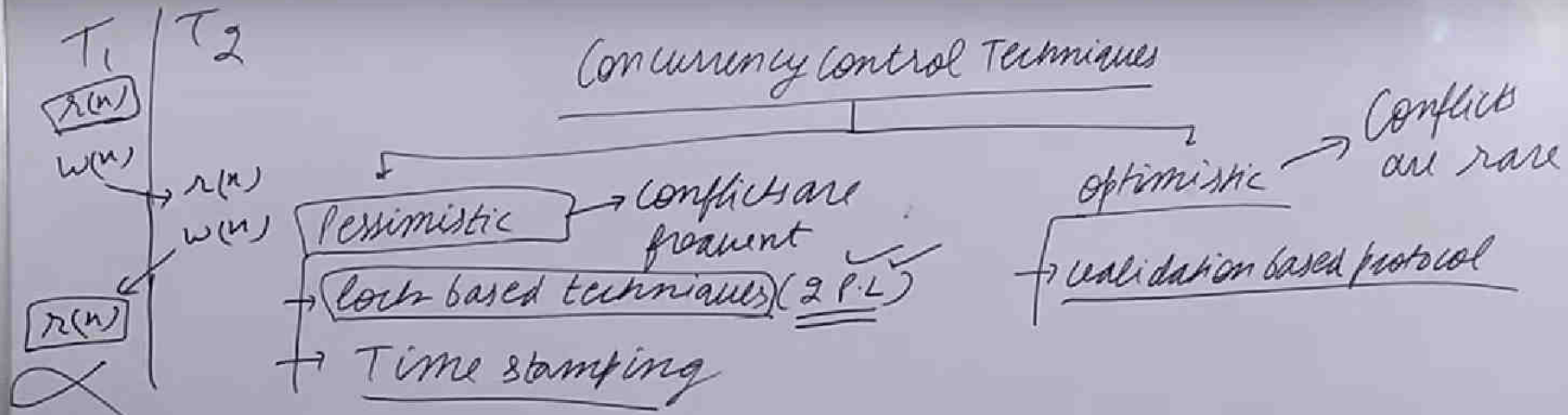
157. strict schedule

How to check Schedule is strict or not?

→ If write operation is preceeded by any conflicting operation (ie read or write) then that conflicting operation must be done after transaction doing write operation (commit or abort).




Relationship b/w all schedules
Recoverable
Cascadeless
Strict
Serial
C.S.
V.S.
Serializable
T1
T2
W(A)
W(A)
C.S.
V.S.
S
R
C
Strict
Serializable
V.S.
C.S.
Serial
Recoverable
Cascadeless
Strict
Serial
T1
T2
W(A)
W(A)
C.S.
V.S.
R
C
Strict



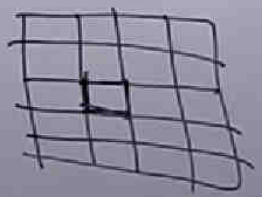
Lock based techniques:-

→ Lock is a variable associated with each database item & it describes the operation which can be performed on item.

→ If item is a cell then lock is associated with each cell

→ " " " " tuple " " " " " " " " tuple

→ " " " " table " " " " " " " " table



Types of locks:-

Binary locks

lock → 0 (unavailable)
lock → 1 (available)

→ If lock(x) = 0 then any transaction say T_i has already applied lock on x so any other transaction say T_j can't access item x [neither read nor write]

→ If lock(x) = 1 then any transaction T_i can apply lock on x & access it (i.e. read or write x)

→ W/o applying lock transaction can't access any item

→ Disadvantage is that 2 transactions can't read same item x.

Shared/Exclusive locks

lock → 0 (Available)
lock → 1 (Shared/Read)
lock → 2 (Exclusive/Write)

→ More than one transaction can have shared lock on same item. but atmost one transaction can have write lock on item.

→ If one transaction is having write lock then no other transaction can read or write the same item.

→ If one transaction is having read lock then no other transaction can write the same item.

T_2 \ T_1	R	W
R	✓	×
W	×	×

→ If transaction wants to access any item then it will 1st ask for lock from lock manager. If lock is available then it is granted else transaction will have to wait.

→ Locks are granted in FIFO order hence lock based techniques are starvation free.

WHY LOCKING PROTOCOLS?

T_1	T_2
$r(x)$	$x \to 500-$
$x = x - 500$	$y \to 500+$
$w(x)$	
Unlock(x)	
	$r(x)$
	$r(y)$
	display(x+y)
	Unlock(x)
	Unlock(y)
$r(y)$	
$y = y + 500$	
$w(y)$	
Unlock(y)	

T_1	T_2
$r(x)$	
$w(x)$	
	$r(y)$
	$r(x)$
$w(y)$	

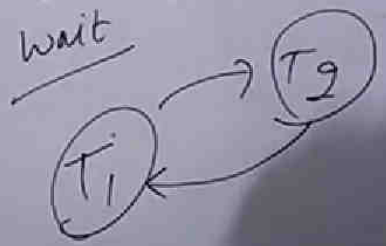
→ Transaction must not release lock too early or too late

→ Hence we need protocol (set of rules) which must be followed by each transaction.

T_1, T_2, T_3 Shared Lock x

T_4 → Exclusive lock x

Wait

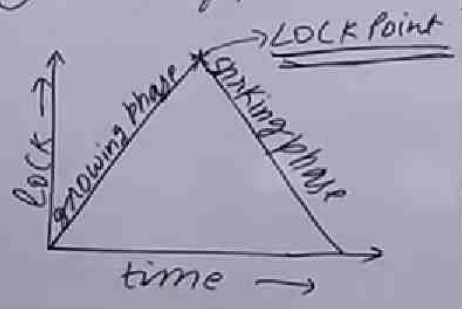


2 Phase Locking

→ As the name suggest here we have 2 phases i.e growing phase & shrinking phase

① Growing phase → Transaction can only acquire lock & it will not release any lock.

② Shrinking phase → Transaction can only release lock & it cannot acquire lock.



T_1	T_2	T_3
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		

2PL ✓

T_1	T_2	T_3
	R(A)	
W(B)		
W(C)		
		R(B)
	R(B)	
R(A)		
C		
	R(C)	
	C	
		W(A)
		C

T_1	T_2
lock-X(A)	
R(A)	
W(A)	
unlock(A)	→ S.P.
	lock-S(A)
	R(A)
lock-S(B)	→ L.P.
R(B)	

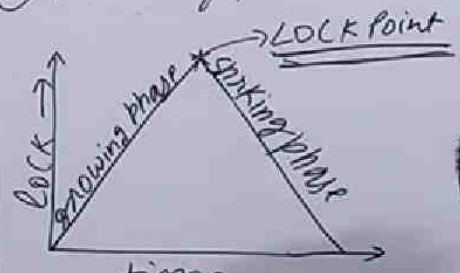
2PL

2 Phase locking

→ As the name suggest here we have 2 phases

① Growing phase → Transaction can only acquire lo

② Shrinking phase → Transaction can only release lo



T1 T2

X(A) R(A) W(A) UNLOCK(A)

LOCK-S(B)

R(B)

T1 T2

X(A) R(A) W(A)

LOCK-X(A)

T1 T2

X(A) R(A) U(A)

X(A) W(A)

C.S X

2PL X

T1 T2 T3

X(B) X(C)

S(A) R(A) U(A)

S(A) W(B) W(C) U(B) U(C)

2 P.L

S(B) R(B) U(B)

S(B) R(B) U(B)

R(A) Commit U(A)

S(C) R(C) Commit

U(C) U(A)

X(A) W(A) Commit

Disadvantages of 2PL

① It guarantees conflict serializability of schedules hence Serializability is guaranteed

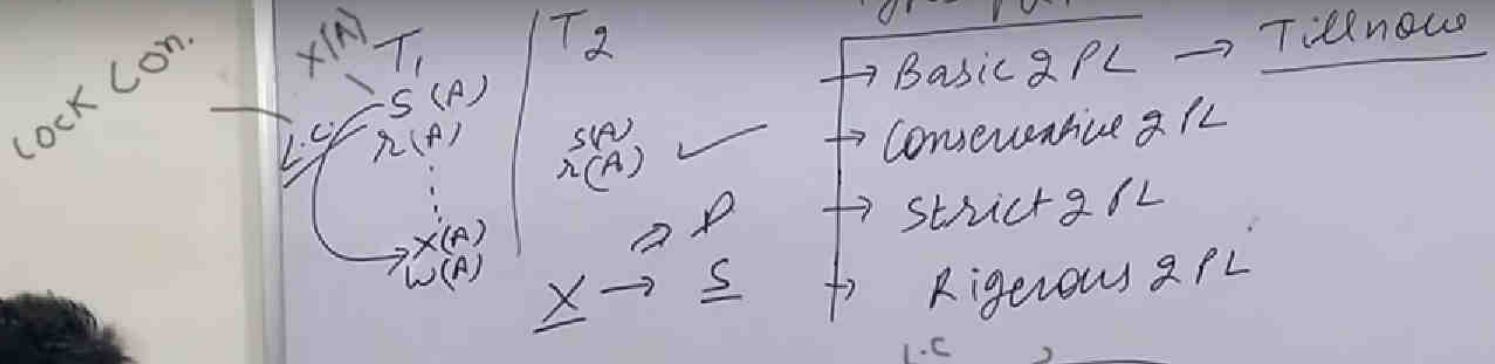
Disadvantages :-

① 2PL doesn't allow all C.S. schedules

② 2PL is not deadlock free

③ 2PL doesn't guarantee recoverable, Cascadeless & strict schedules

④ 2PL limits the amount of concurrency



Basic 2PL will lock conversions

→ In Basic 2PL if transaction wants to read any item & then after some time it wants to write that item. It will demand exclusive lock on that item by lock manger directly i.e read lock is not converted to exclusive lock.

→ But in Basic 2PL with lock conversion read lock can be converted to exclusive lock.

→ Conversion of shared to exclusive lock is known as upgrading of lock & is done in Growing phase only.

→ Conversion of exclusive to shared lock is known as downgrading of lock & is done in shrinking phase only.

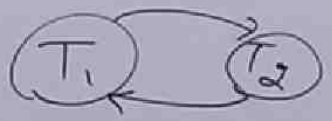
Conservative 2 PL :-

→ Each transaction must predeclare its read set & write set (Assumption)
→ Transaction will not start till it acquire lock on all items in read set & write set
→ Hence transaction start at lock point & hence enters the shrinking phase.

Advantages :-

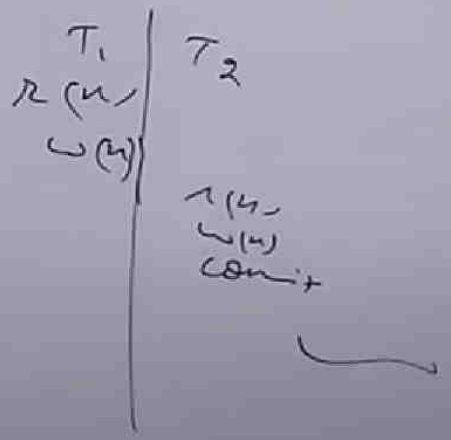
① 9t guarentee C.S. of schedules
② 9t is deadlock free

T_1 → x, y → read
z, w → write



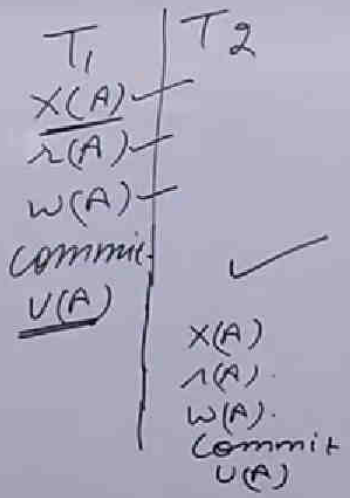
Disadvantages :-

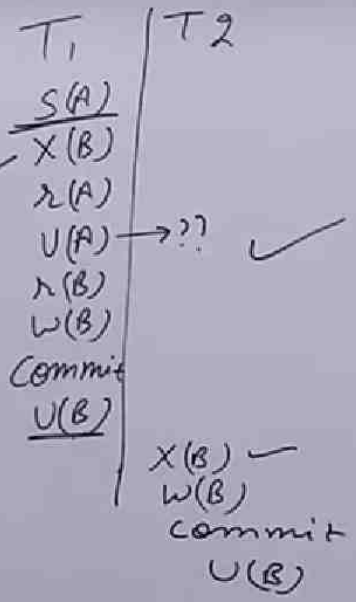
① 9t limits the amount of concurrency
② 9t doesn't guarentee recoverable, cascadeless & strict schedules
③ Not practical to use in DBMS
④ All C-S schedules are not allowed.

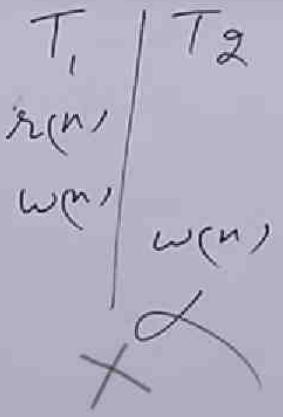


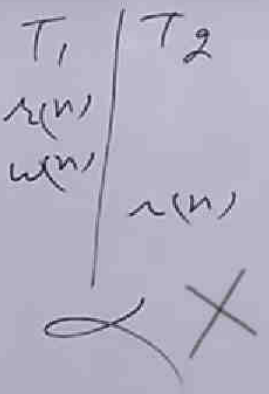
Strict 2 P.L

→ Transaction will not release any of its exclusive locks until it commits or aborts.









Advantages:-

① It guarantee C.S. schedules

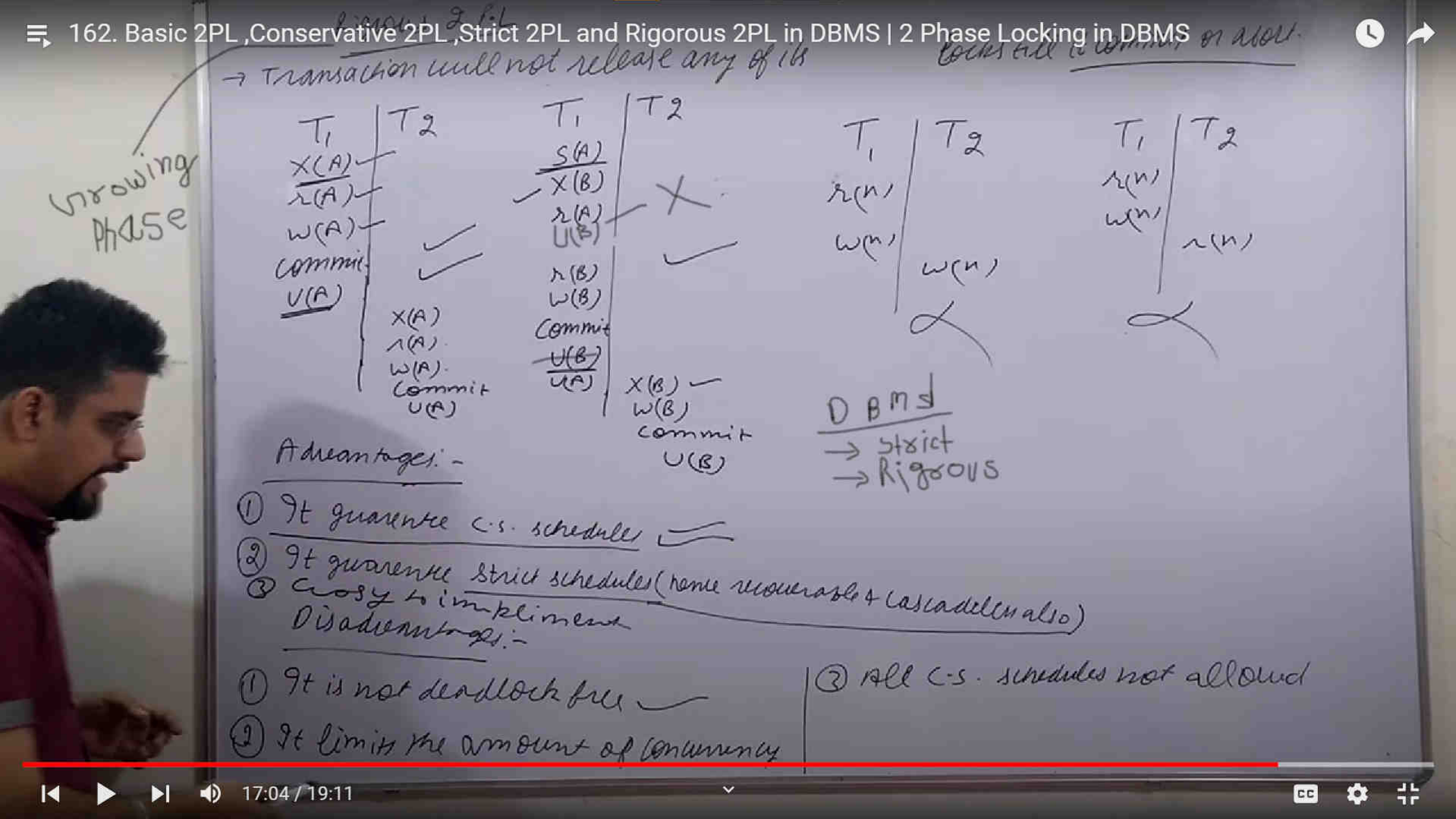
② It guarantee strict schedules (hence recoverable + cascadeless also)

Disadvantages:-

① It is not deadlock free

② It limits the amount of concurrency

② All C.S. schedules not allowed


Growing Phase
→ Transaction will not release any of its locks till it commits or aborts
T1 | T2
X(A) | X(A)
R(A) | R(A)
W(A) | W(A)
Commit | Commit
U(A) | U(A)
T1 | T2
S(A) | X(B)
X(B) | W(B)
R(A) | Commit
U(B) | U(B)
R(B) | U(A)
W(B)
Commit
U(B)
T1 | T2
R(A) | W(A)
W(A) | Commit
T1 | T2
R(A) | R(A)
W(A) | W(A)
Commit
DBMS
→ Strict
→ Rigorous
Advantages:-
① It guarantee C.S. schedules
② It guarantee strict schedules (hence recoverable + cascadeless also)
③ Easy to implement
Disadvantages:-
① It is not deadlock free
② It limits the amount of concurrency
③ All C.S. schedules not allowed

Timestamp ordering protocol (2nd C.C.T)

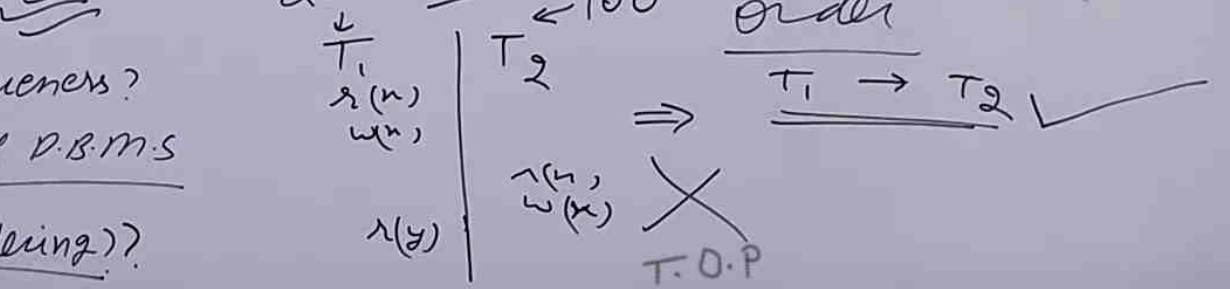
→ Timestamp is a unique value assigned by DBMS to each transaction.
→ We can think timestamp as the start time of transaction.
→ TS(Ti) ⇒ Timestamp of Ti
→ TS(Ti) < TS(Tj) ⇒ Tj is younger (↳ Junior) & Ti is older (↳ senior)

Q1 How to impliment timestamps?
Ans1 2 ways < system clock. / counter

Q2 How to guarantee uniqueness?
Ans2 Its the responsibility of D.B.M.S

Q3 What is timestamp ordering??
Ans3 The main aim of T.O.P is to impliment FIFO order i.e system will allow only those schedules whose equivalent serial schedule contains transactions according to their time of entering the system.
But in 2PL FIFO order was not followed.

C.S. ⇒ Serializability order



Timestamp ordering protocol (2nd C.C.T)

→ Timestamp is a unique value assigned by DBMS to each transaction.
→ We can think timestamp as the start time of transaction.
→ TS(Ti) ⇒ Timestamp of Ti
→ TS(Ti) < TS(Tj) ⇒ Tj is younger & Ti is older (↳ Junior / ↳ senior)

Q1 How to implement tim...
Ans1 2 ways < System clo... / counter...

Q2 How to guarantee uni...
Ans2 Its the responsibility o...

Q3 What is timestamp...
Ans3 The main aim... ent FIFO order i.e s... will allow only those sched... to their ti... transactions according

But in ... not followed.

T1 → x, y
T2 → x, y

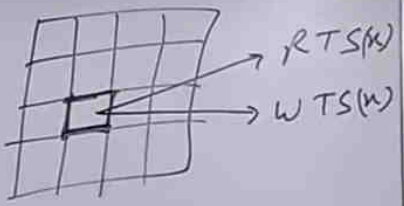
T1 | T2
r(x)
w(x)
r(x)
r(x)
w(x)
r(y)
w(y)
r(y)
w(y)

200 C.S. ⇒ 100
T1 | T2
r(x)
w(x)
r(y)

T2 → T1

Types of Time stamp ordering protocol

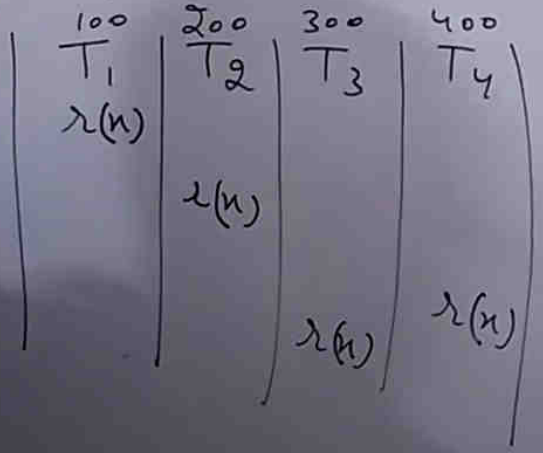
- Basic T.S.O.P
- Strict T.S.O.P
- Total TSOP
- Multiversion TSOP — ??



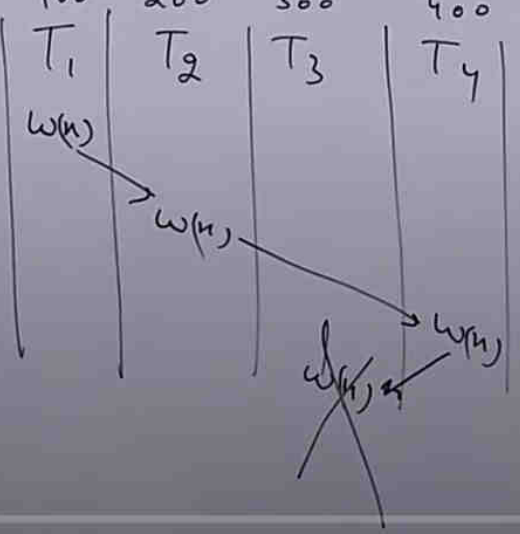
→ 2 timestamps associated with item 'x' i.e RTS(x) & WTS(x)

RTS(x) → Time stamp of youngest transaction that has performed read on 'x'

WTS(x) → Write



RTS(x) = 400 → 200 → 100



WTS(x) = 400 → 200 → 100 → 0

Basic T.S.O.P

If T_i wants to read item 'x'

If $TS(T_i) < WTS(x)$ → 50

$TS(T_i)$ → 100

Abort T_i & restart it with new timestamp.

else

read operation is granted.

$$RTS(x) = MAX(TS(T_i), RTS(x))$$

50 T_J $w(x)$

100 T_i $r(x)$

100 T_1 $w(x)$ | 200 T_2 $r(x)$ | 300 T_3 $r(x)$

T_2 request for read

$TS(T_2) < WTS(x)$ → 100

$TS(T_2)$ → 200

$RTS(x) = TS(T_2)$ → 200

100 T_i $w(x)$ | 200 T_J $r(x)$

$RTS(x) = 300$

If T_i wants to write item 'x'

If $TS(T_i) < RTS(x)$ OR $TS(T_i) < WTS(x)$

Abort T_i & restart it with new T.S.

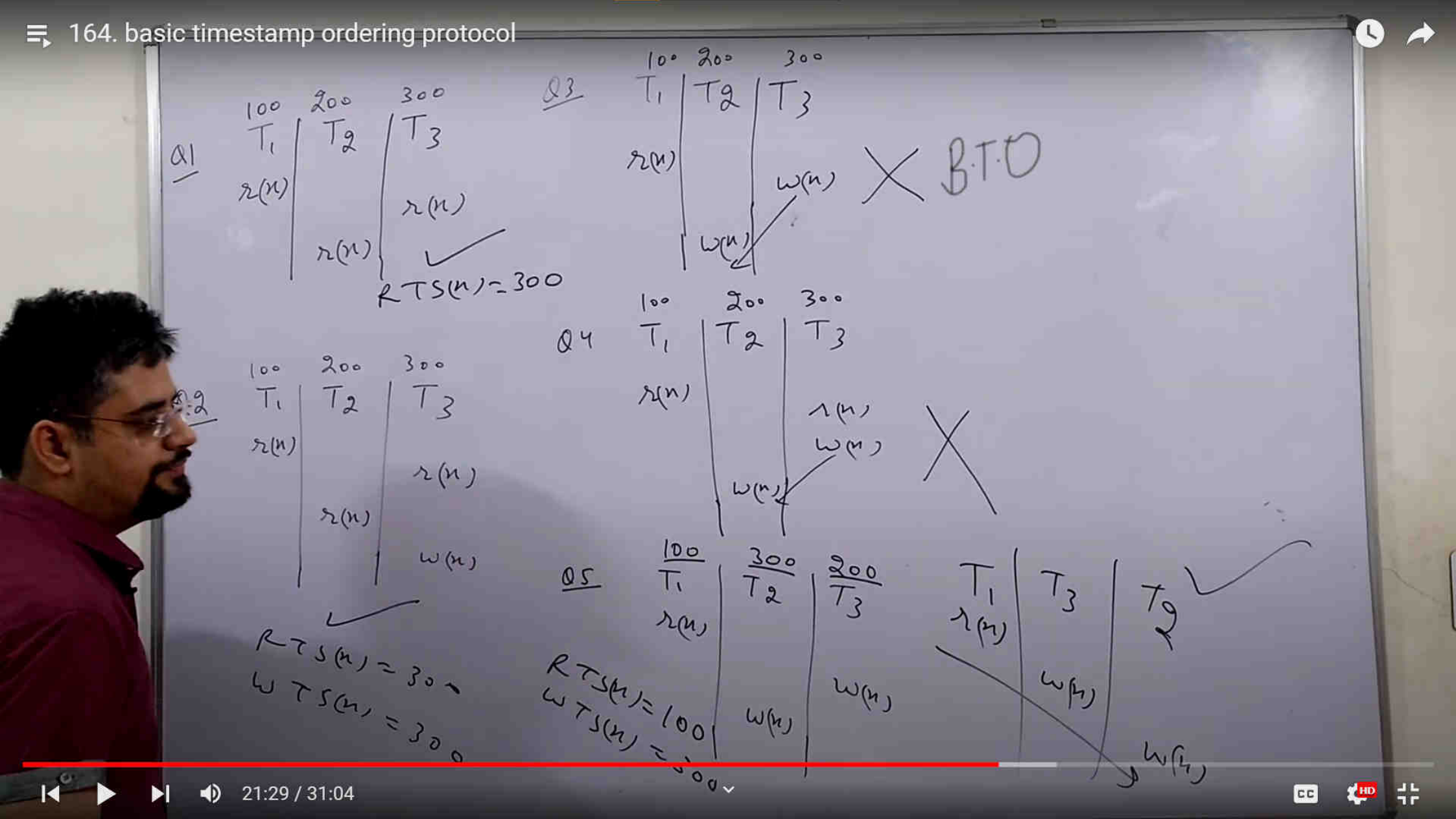
else

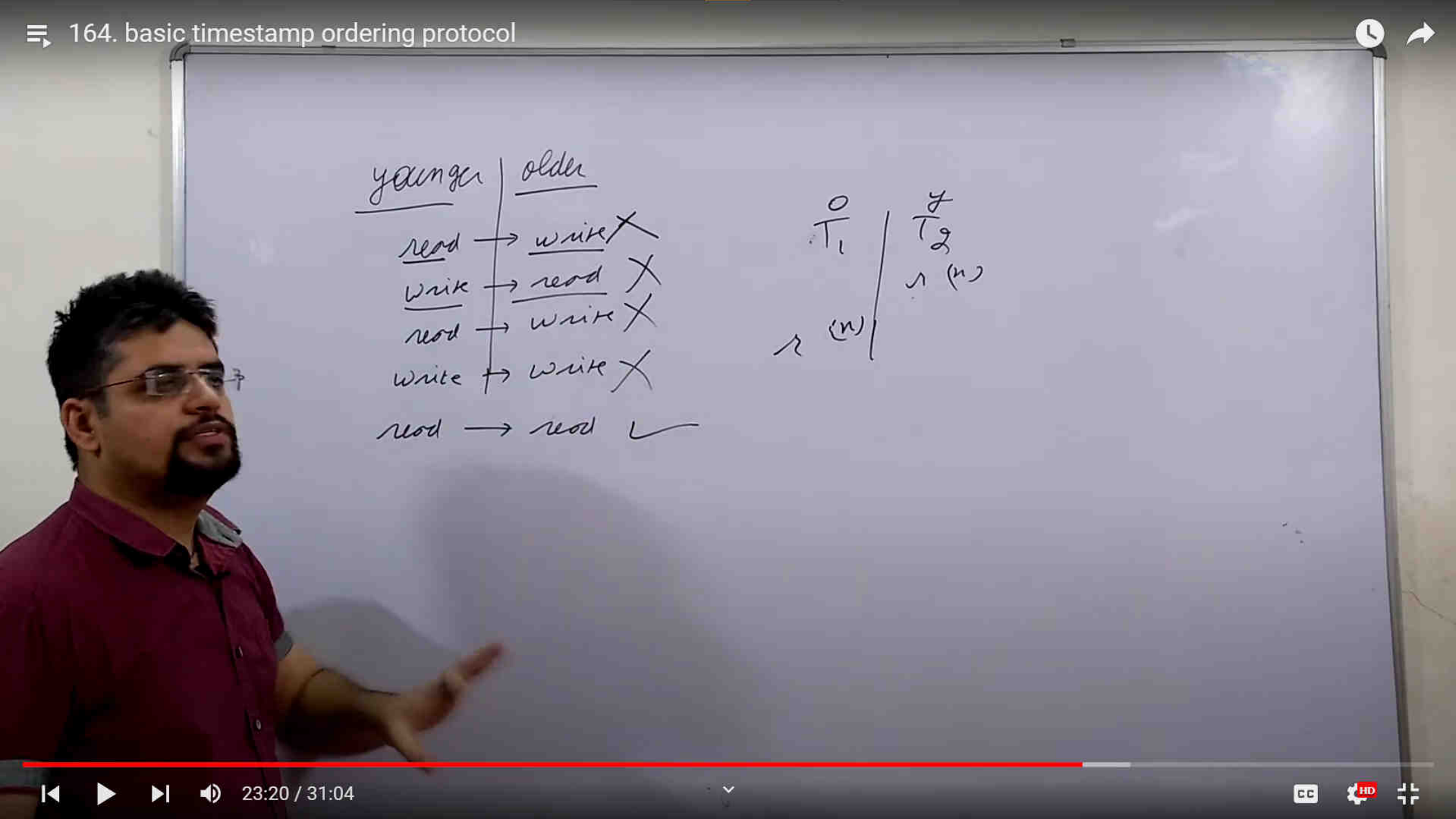
write operation granted

$$WTS(x) = TS(T_i)$$

100 T_i $w(x)$ | 200 T_J $w(x)$

100 T_i $w(x)$ | 200 T_J $r(x)$ ✗


Q1
100 200 300
T1 T2 T3
r(x)
r(x)
r(x)
RTS(x)=300
Q2
100 200 300
T1 T2 T3
r(x)
r(x)
r(x)
w(x)
RTS(x)=300
WTS(x)=300
Q3
100 200 300
T1 T2 T3
r(x)
w(x)
w(x)
BTO
Q4
100 200 300
T1 T2 T3
r(x)
r(x)
w(x)
w(x)
Q5
100 300 200
T1 T2 T3
r(x)
w(x)
w(x)
RTS(x)=100
WTS(x)=300
T1 T3 T2
r(x)
w(x)
w(x)


younger
older
read → write ✗
write → read ✗
read → write ✗
write → write ✗
read → read ✓
T1
T2
(R)
(W)

Adv antages:-

① It guarentee conflict serializability of schedules. ✓

② It is deadlock free technique. ✓

Disadvantages:-

① It does not guarantee recoverable, cascadeless & strict schedules

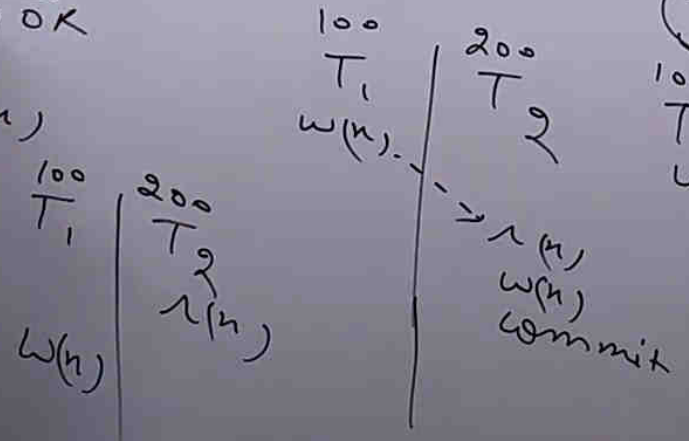
② All C.S. schedules are not allowed (like 2PL)

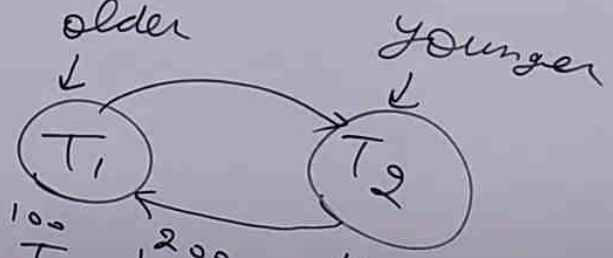
③ It is not starvation free (unlike 2PL)

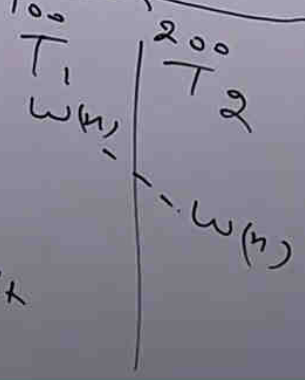
$T_i \rightarrow$ write (x)

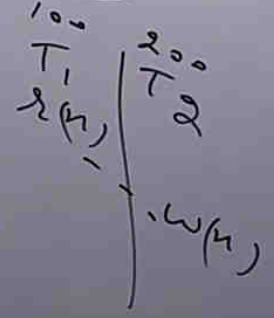
$TS(T_i) < RTS(x)$ OK

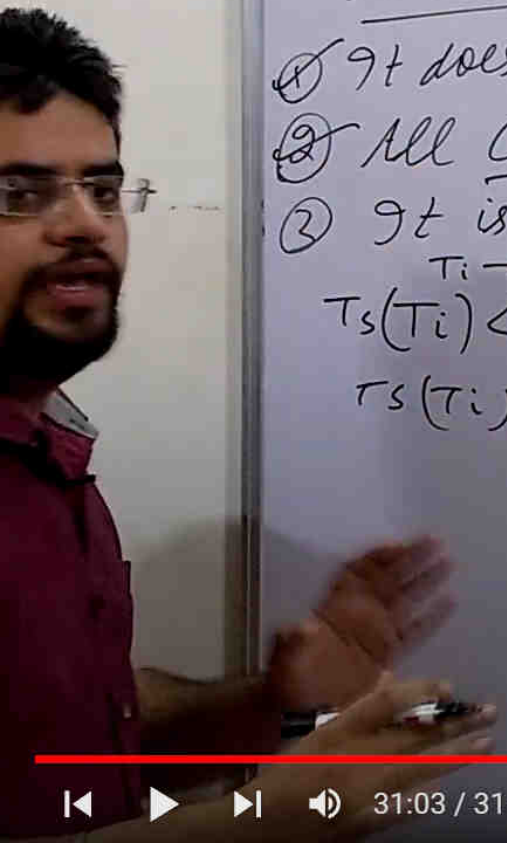
$TS(T_i) < WTS(x)$





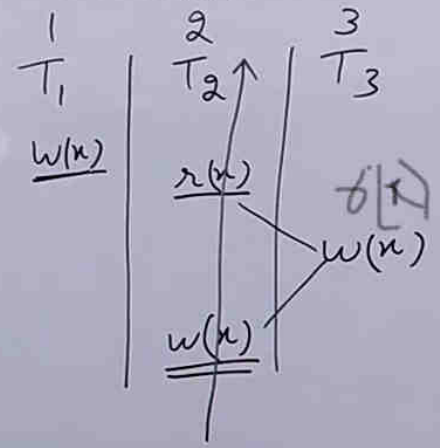






Thomas write rule :-

→ Each transaction must read latest value of item (Not outdated (stale) value) (obsolete)



→ Here T_2 has read latest value of x [written by T_1]

→ T_3 has read value of x written by T_1 & when T_3 read item x then system assumes T_2 doesn't want to write x & hence for T_3 value of x is latest.

→ Now after some time when T_2 wants to write x then its against assumption of system hence it will reject w(x) of T_2 & roll back it. [∵ the given schedule is not serializable]

Q what if younger has done blind write?

Ans If T_3 has done blind write then w(x) of T_2 can be ignored & T_2 can continue to execute

∵ any Transaction T_i with $TS(T_i) < WTS(x)$ (3) will be rolled back according to TSO.P

& any Transaction T_j with $TS(T_j) > WTS(x)$ (3) will read value of x written by T_3 (if it wants to read x) rather than the value that T_2 is attempting to write.

Thomas write rule:-

→ If younger transaction has done r(x) then w(x) of older transaction will be rejected & it will be rolled back.

→ If younger transaction has done blind write then write operation (on same item) of older transaction can be ignored & it can continue execution.

If TS(T_2) > TS(T_1)

T_1	T_2
r(x)	
	r(x)

TSOP → ✓
TWR → ✓

T_1	T_2
r(x)	
	w(x)

TSOP → ✗
TWR → ✗

T_1	T_2
w(x)	
	r(x)

TSOP → ✗
TWR → ✗

T_1	T_2
w(x)	
	w(x)

TSOP → ✗
TWR → ✗

ignore w(x) of T_1 & continue T_1

Thomas write rule:-

→ If younger transaction has done r(x) then w(x) of older transaction will be rejected & it will be rolled back.

→ If younger transaction has done blind write then write operation (on same item) of older transaction can be ignored & it can continue execution.

If TS(T_2) > TS(T_1)

T_1	T_2
r(x)	
	r(x)

T_i wants to write item x

① If TS(T_i) < RTS(x) then reject write request of T_i & roll back it

② else if TS(T_i) < WTS(x) then ignore write request of T_i & continue it.

③ else write operation granted

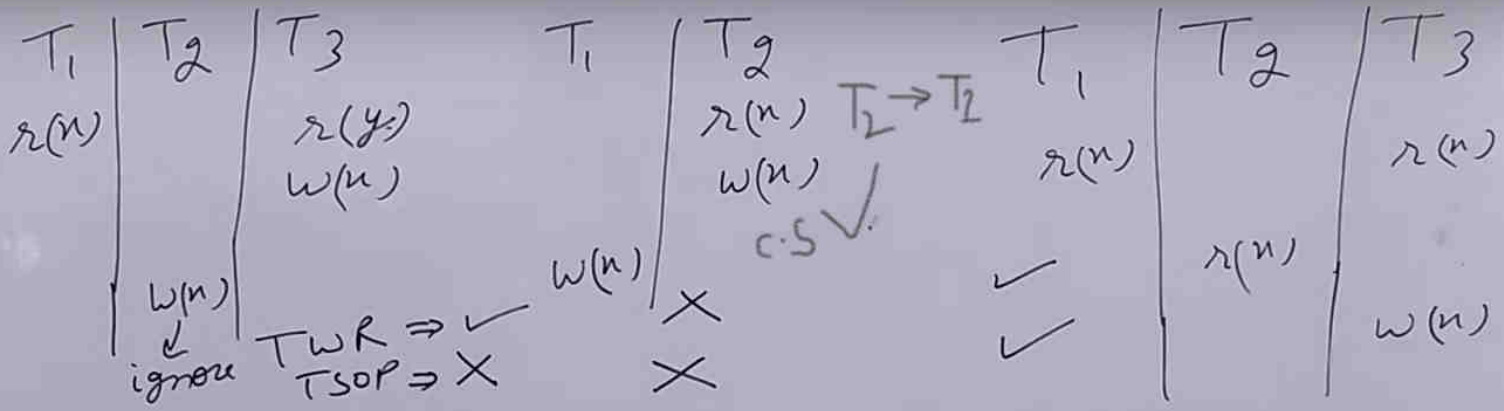
WTS(x) = TS(T_i)

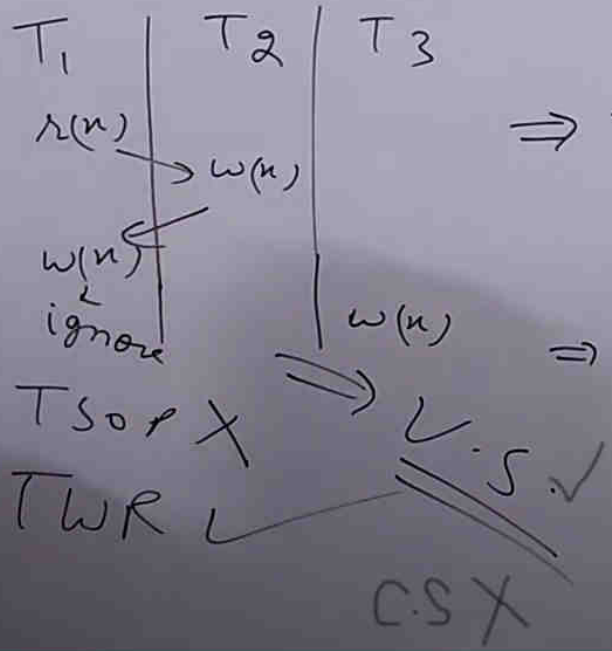
T_1	T_2
r(x)	
	w(x)

T_1	T_2
w(x)	
	r(x)

T_1	T_2
w(x)	
	w(x)

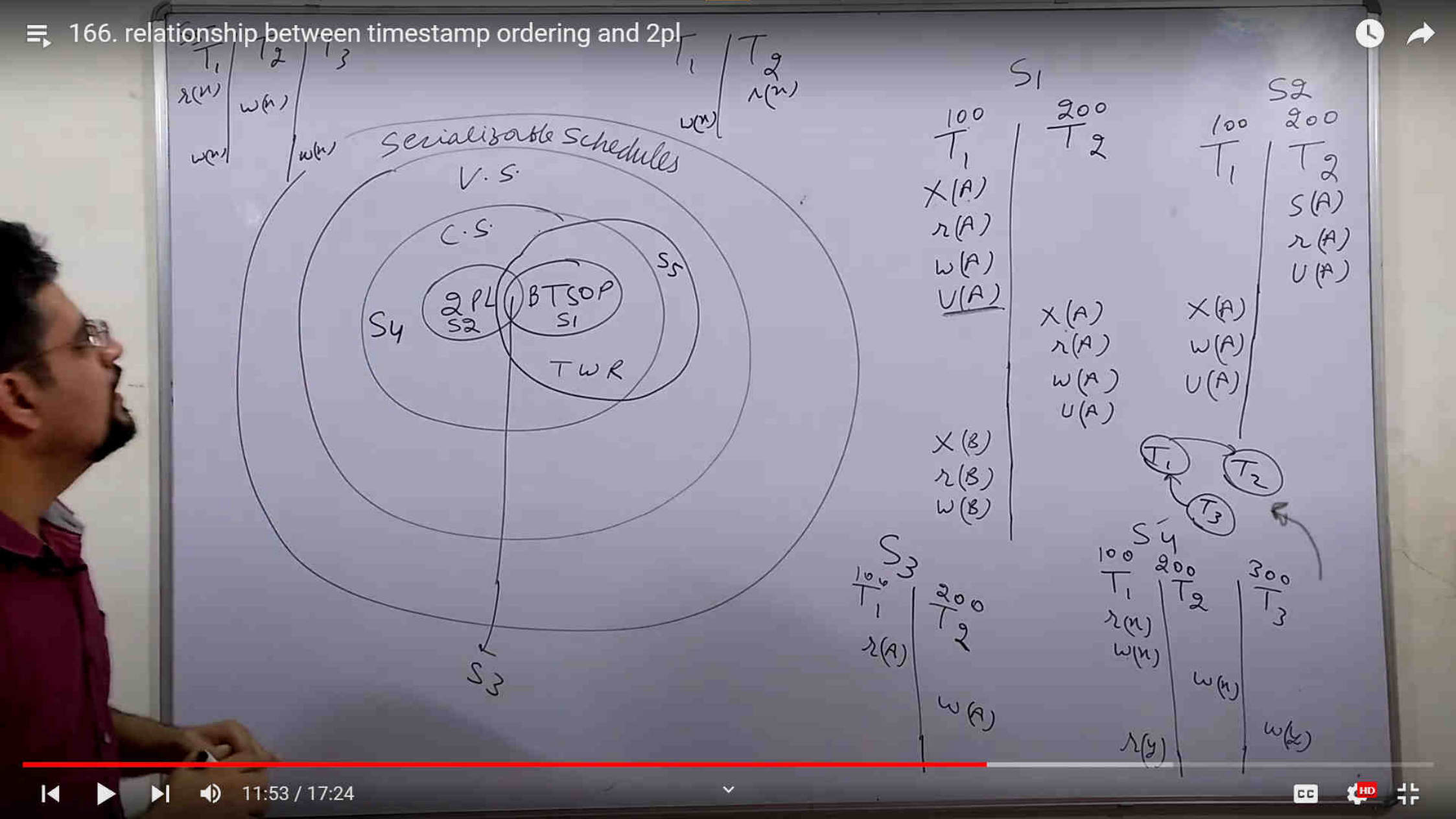
ignore w(x) of T_1 & continue T_1





⇒ TWR doesn't guarantee conflict serializability rather it is based on concept of view serializability So it will result in fewer number of roll backs.

⇒ But all C.S. schedules are not allowed by T.W.R

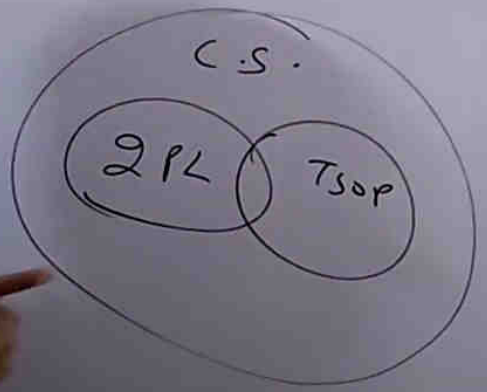

T1 T2 T3
r(x)
w(x)
w(x)
w(x)
T1 T2
w(x)
r(x)
Serializable Schedules
V.S.
C.S.
S5
S4
2PL
S2
BTSOP
S1
TWR
S3
S1
100
T1
X(A)
r(A)
w(A)
U(A)
X(B)
r(B)
w(B)
200
T2
X(A)
r(A)
w(A)
U(A)
S2
100
T1
X(A)
w(A)
U(A)
200
T2
S(A)
r(A)
U(A)
T1
T2
T3
S3
10
T1
r(A)
20
T2
w(A)
S4
100
T1
r(x)
w(x)
200
T2
w(x)
r(y)
300
T3
w(z)

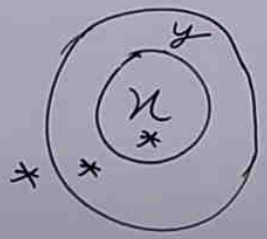
Q1 2PL is sufficient cond^n for conflict serializability ✓ V.S.

Q2 TSOP is sufficient cond^n for ✗ V.S. ✓

Q3 TWR

Q4 TWR is necessary cond^n for conflict serializability ✗ V.S. ✗





x is sufficient cond^n y

Total Timestamp ordering protocol

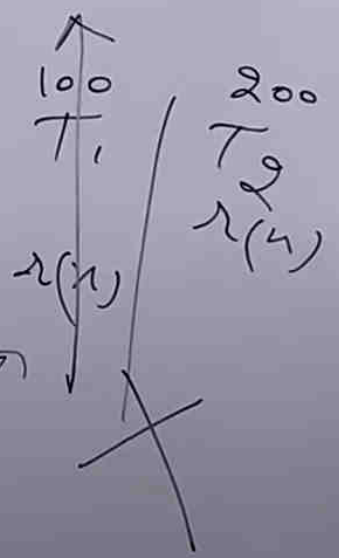
→ With each database item 'x' we associate only one timestamp ie. TS(x)

→ TS(x) → Timestamp of youngest transaction that has performed read or write on item x.

Ti wants to read/write item x :-

If TS(Ti) < TS(x) then
abort Ti & restart it with new timestamp

else
read or write operation is granted
TS(x) = TS(Ti)



Advantage

① Implimentation is easy & memory is saved.

Disadvantage :-

① It will have unnecessary roll backs

Strict Timestamp ordering protocol

→ BTOP doesn't guarentee recoverable, cascadeless & strict schedules
→ So we have a variant of BTOP called Strict Timestamporderingprotocol.
→ As the name suggest it guarentee strict schedules hence recoverable & cascadeless is also guarenteed.
→ If Ti wants to read or write item x such that TS(Ti) > WTS(x) then this read or write operation is delayed till transaction TJ which wrote the value of x (i.e WTS(x) = TJ) commits or aborts.
→ To impliment this scheme we must use concept of locking i.e TJ must not release lock on item x till it commits or aborts.
→ This technique is deadlock free ∵ if Ti is waiting for TJ to release lock on item x then TJ can't wait for Ti to release lock on any other item.

100	200
T_1	T_2
W(x)	?

WTS(x) = 100

Multiversion timestamp ordering protocol

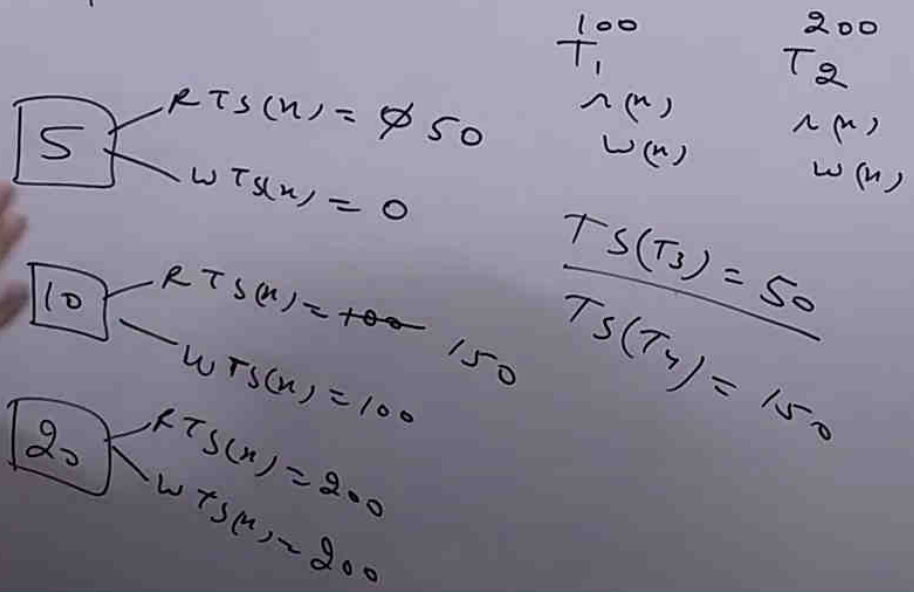
→ As the name suggest we mentain multiple versions of same item 'x'

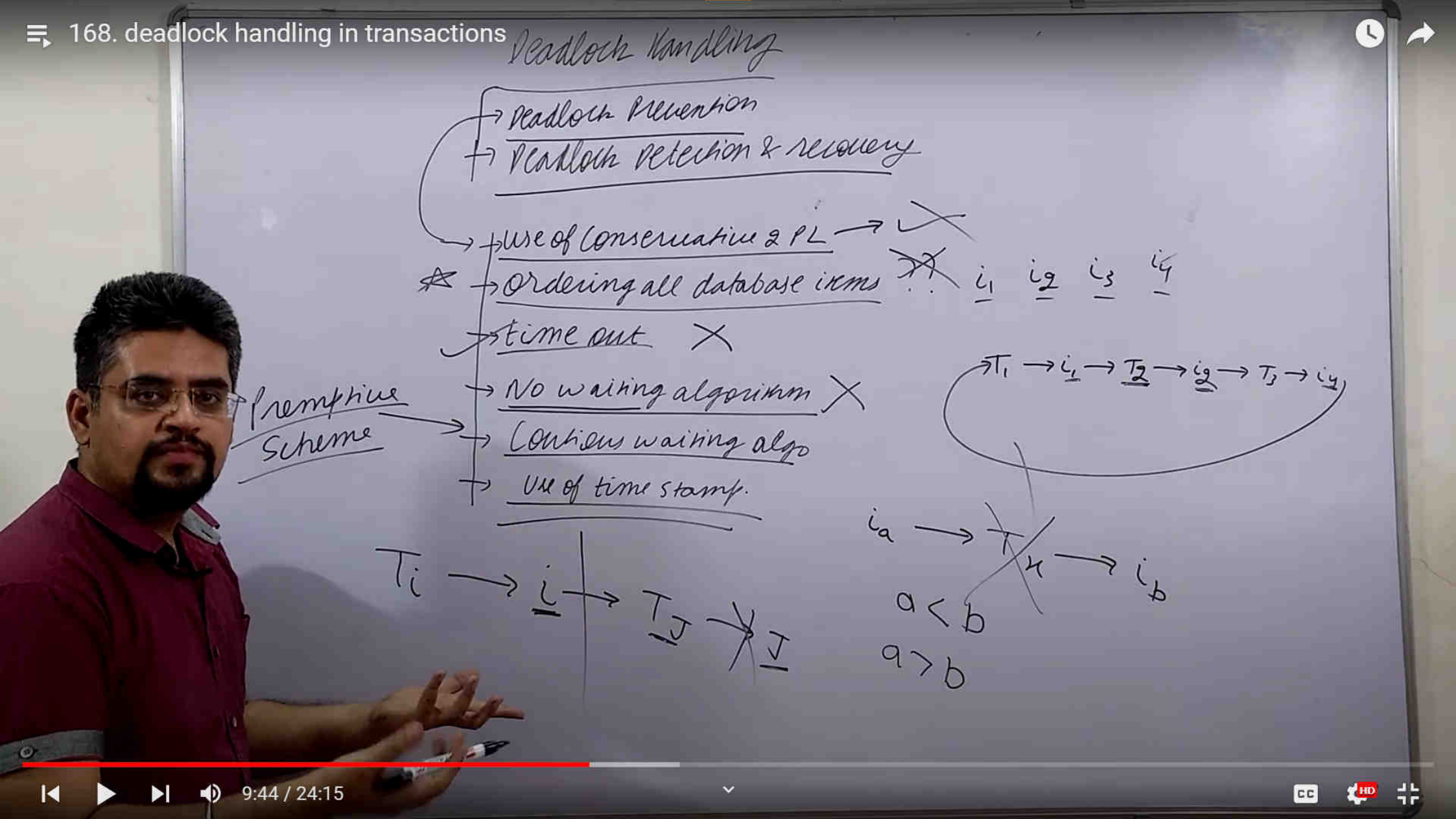
→ Transaction which wants to read any item will read latest value of x according to its timestamp.

→ Lot of memory wasted but it will have least No. of rollbacks.

Rollbacks:-

TTSOP > BTSOP > MVTSOP




Deadlock Handling
Deadlock Prevention
Deadlock Detection & recovery
use of Conservative 2 PL
Ordering all database items
time out
No waiting algorithm
Conservative waiting algo
Use of time stamp.
Preemptive Scheme
i_1 i_2 i_3 i_4
$T_1 \to i_1 \to T_2 \to i_2 \to T_3 \to i_4$
$T_i \to i \to T_j$
$i_a \to i_b$
$a < b$
$a > b$

Use of time stamp:-

Wait & die

→ Ti wants an item held by TJ then:-

If TS(Ti) < TS(TJ) then Ti is allowed to wait

else
roll back Ti & restart it with old time stamp.

→ Non pumptive

Wound & Wait

→ Ti wants an item held by TJ then:-

If TS(Ti) < TS(TJ) then Ti pumpts the item from TJ & TJ is rolled back & started with old time stamp

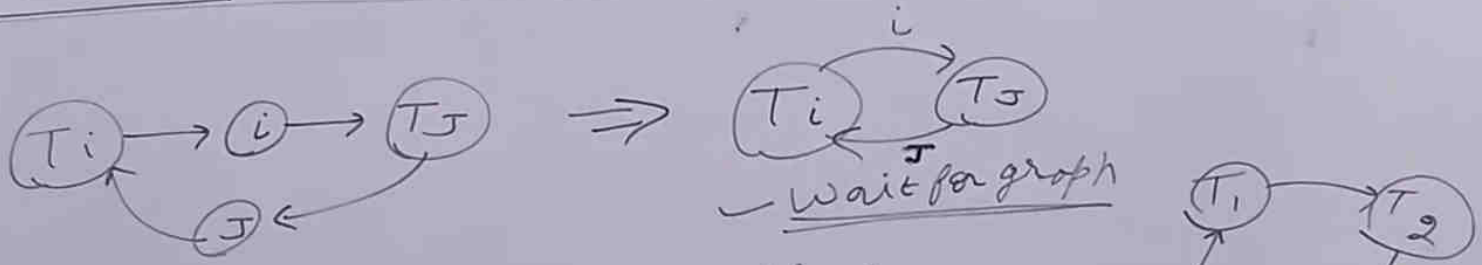
else Ti is allowed to wait.

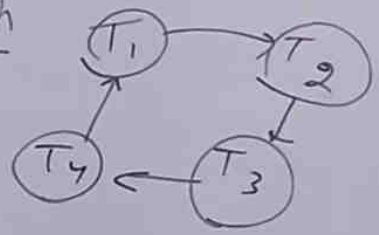
→ Pumptive

→ Both techniques are starvation free & Junior is always roll back.

Deadlock Detection & Recovery

Wait for graph is used for deadlock detection





→ cycle in wait for graph means deadlock

Recovery:-

→ simplest way of deadlock recovery is to rollback any trasaction

→ Transaction which is choosen for rollback is known as victim transaction

→ many criteria for choosing victims like youngest, few locks,

GATE 2016

Consider the following 2PL. Let transaction T access set of objects $\{O_1 \cdots O_K\}$. This is done in following manner.

Step 1 T acquires Locks on $O_1 \cdots O_K$ in increasing order of their addresses

Step 2 The required operations are

Step 3 All locks are released.

T_1 O_1, O_2

T_2 O_1, O_2 ← Not lock

Wait

Guarentee Serializability

Guarentee Deadlock

$T_1 \rightarrow O_1$

$T_2 \rightarrow O_2$

Optimistic Concurrency Control technique

→ In pessimistic technique we have a common buffer in which transactions read or write data.

→ But no concept of common buffer in optimistic technique.

→ Hence each transaction has its own private buffer in which it read & write data

Validation based protocol:-

T1

R(x)

3 phases → Read, validate, write

Read phase → Transaction read items from database & store them in buffer & then perform read & write operations blindly in buffer itself.

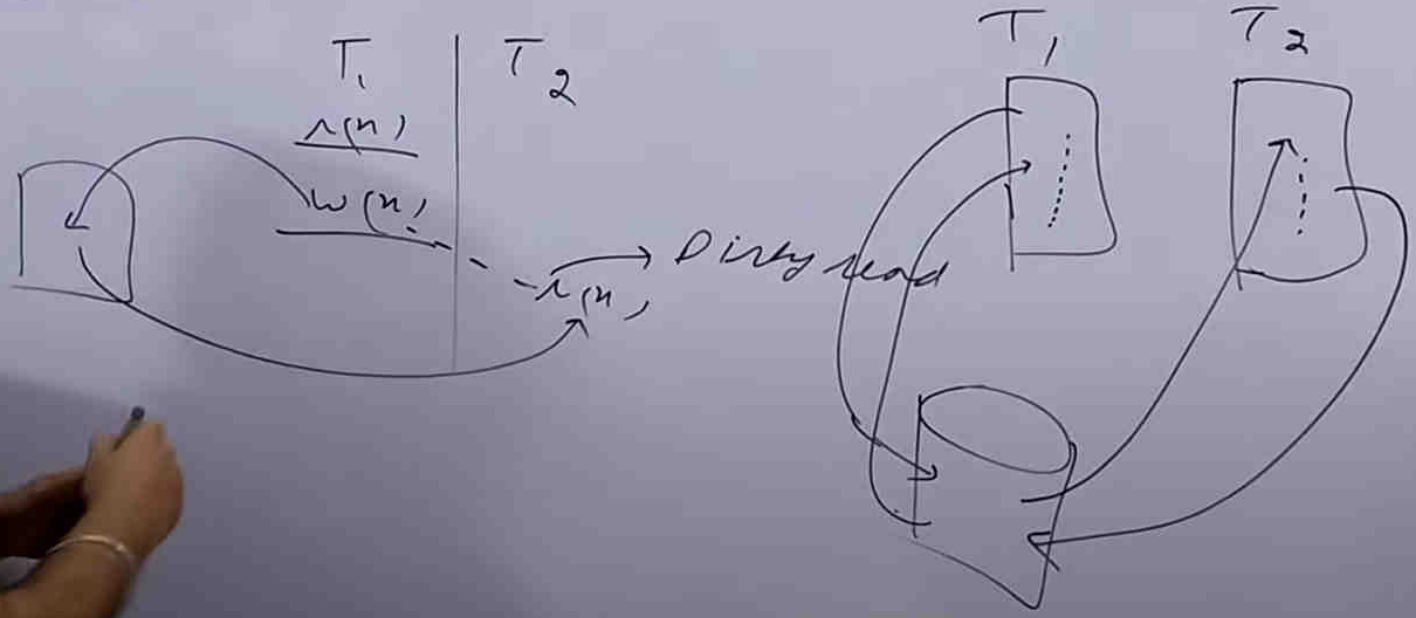
Validate phase → When transaction completes then it enters validation phase where test for serializability is performed. If transaction passes validation test then it enters write phase else transaction is restarted.

Write phase → All updated done in buffer are applied to database.

Advantages:-

① Optimistic techniques are very fast because transactions are executed blindly But in pessimistic techniques some check is performed before transaction read or write data

② It guarantee recoverable & cascadless schedules ∴ No dirty read.



Log based Recovery

Recovery → Recovery is a technique to restore the database to most recent consistent state if any kind of failure occurs.

Failure → Disk failure, catastrophic failure, transaction failure
- Logical error → I/O, item not found
- System error → deadlock.

→ Hence we have to **undo** all the changes done by uncommitted transactions & **redo** all changes done by committed transactions.

→ How DBMS remember all this will be studied in log based recovery.

Log → Log is a special file maintained by DBMS to keep track of all transactions operations that affect the values of database items.

→ Log file is stored on stable storage so that data never loss

Log file :-

$\langle T_1, \text{start} \rangle$ ⇒ start record

$\langle T_1, x_{old} = 500, x_{new} = 1000 \rangle$ ⇒ write record

$\langle T_1, y_{old} = 1000, y_{new} = 1500 \rangle$

$\langle T_1, \text{commit} \rangle$ ⇒ commit record

$\langle T_2, \text{start} \rangle$ — lost time

$\langle T_2, z_{old} = 500, z_{new} = 150 \rangle$

$\langle T_2, \text{abort} \rangle$ ⇒ abort record

Deferred updates

→ If failure occurs then Redo (Set values of all items updated by transaction to new value) those transactions for which commit is found.

→ Simply ignore those transactions for which commit is not found i.e No need of undo [∵ database is not modified].

→ Hence name of recovery algo is No undo/redo algo.

→ Whenever transaction performs the write, the log record for that write must be created before the database is modified otherwise recovery may not be possible.

→ Whenever $\langle T_i, \text{commit} \rangle$ record is written in the log then this guarantee that all modifications done by transaction is successfully recorded in the log file in form of write records in stable storage.

→ 3 possibilities
- Database is not modified at all → Deferred update
- Few writes are done in database (but can't guarantee all writes).
- All writes are done in database. → Restricted immediate update

immediate update — General case

No undo/redo Algo:-

→ This recovery algo is used in deferred updates scheme.

→ After failure scan the log in forward direction.

→ If $\langle T_i, start \rangle$ record is found but $\langle T_i, commit \rangle$ record is missing then ignore such transactions.

→ If $\langle T_i, start \rangle$ & $\langle T_i, commit \rangle$ both records are present then redo T_i

→ **Why forward ??**

$\langle T_1, start \rangle$

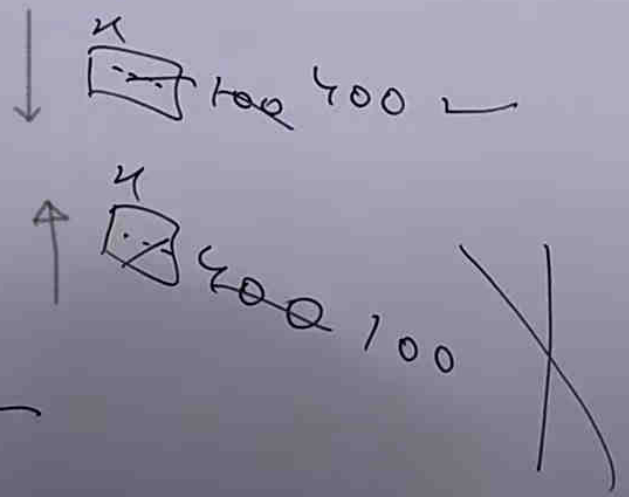
$\langle T_1, x_{new} = 100 \rangle$

$\langle T_1, commit \rangle$

$\langle T_2, start \rangle$

$\langle T_2, x_{new} = 400 \rangle$

$\langle T_2, commit \rangle$



Restricted immediate updates

→ Before writing $\langle T_i, commit \rangle$ record in log file DBMS guarantee that all changes done by transaction T_i are already written in database

→ Hence if any failure occurs then scan the log file & if $\langle T_i, start \rangle$ & $\langle T_i, commit \rangle$ both log records are present then ignore T_i.

→ If $\langle T_i, start \rangle$ is present but $\langle T_i, commit \rangle$ is absent then undo T_i.

Hence name of algo is

No redo/undo

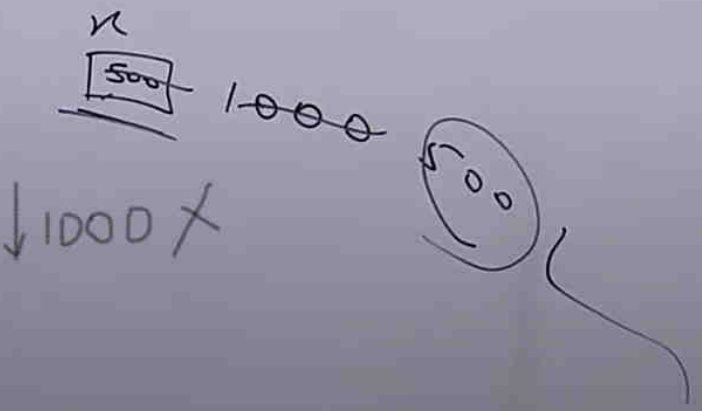
→ Direction ??

$\langle T_1, start \rangle$

$\langle T_1, n_{old} = 500 \rangle$

$\langle T_2, start \rangle$

$\langle T_2, n_{old} = 1000 \rangle$



Immediate updates

→ Before writing <Ti, commit> record in log file No guarantee that all changes done by transaction Ti are already written in database.

→ Hence if any failure occurs then scan the log file & if <Ti, start> & <Ti, commit> both log records are present then redo Ti.

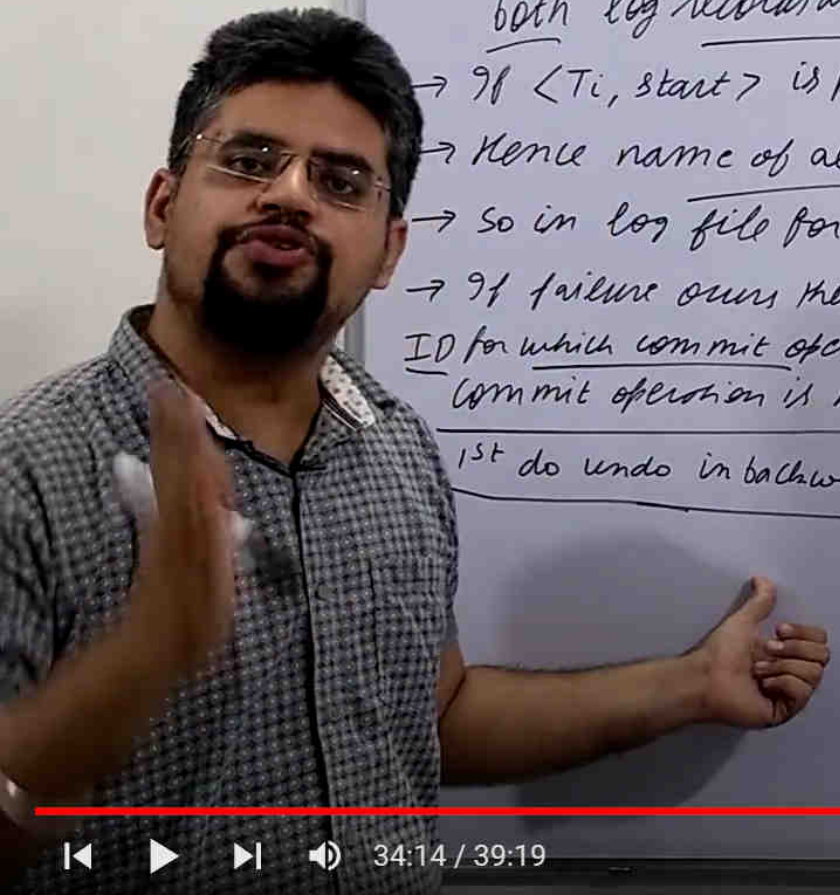
→ If <Ti, start> is present but <Ti, commit> is absent then undo Ti.

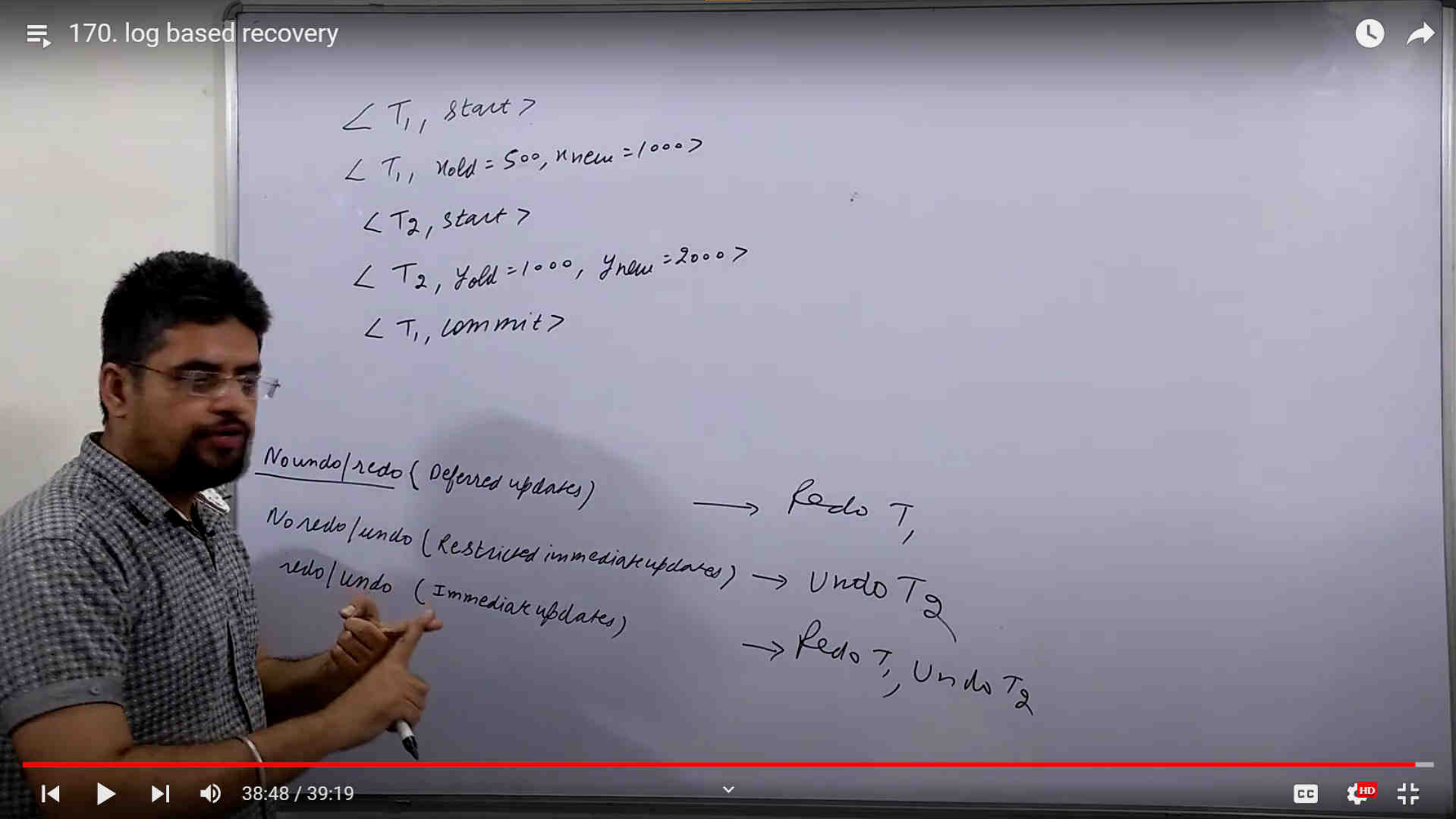
→ Hence name of algo is undo/redo algo.

→ So in log file for each write record both old & new value must be stored.

→ If failure occurs then make 2 lists of transactions ie redo list containing transaction ID for which commit operation is found in log & undolist containing transactions ID for which commit operation is missing

1st do undo in backward direction & then redo in forward direction.




< T1, start >
< T1, xold = 500, xnew = 1000 >
< T2, start >
< T2, yold = 1000, ynew = 2000 >
< T1, commit >
Noundo/redo (Deferred updates) → Redo T1
Noredo/undo (Restricted immediate updates) → Undo T2
redo/undo (Immediate updates) → Redo T1, Undo T2

1. T_1 : start
2. T1 : Bold = 12000, new = 10000
3. T1 : mold = 0, new = 2000
4. T1 : commit
5. T2 : start
6. T2 : Bold = ... new = 10500
7. ~~T2 : co...~~

redo/undo

Database ... hed just before log ... written

T_1 - ... 3

... 6

... 6 then let undo

... 2, 3

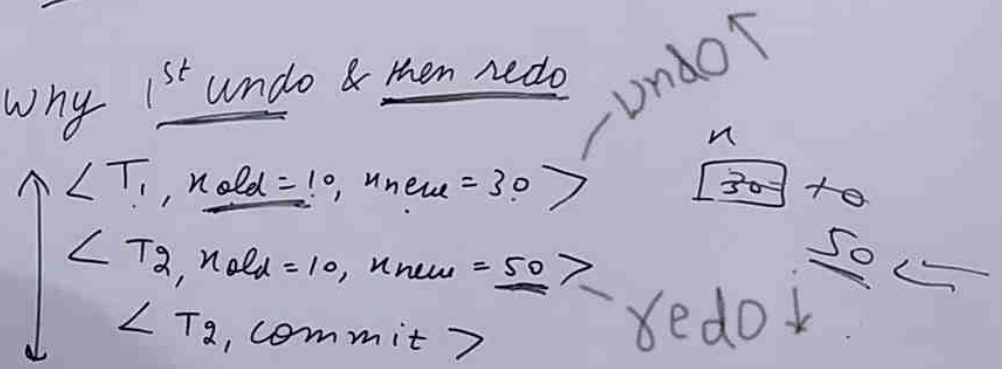
Q1 $T_1 \Rightarrow$ 2, 3, 4, 5 & commit is present

$T_2 \Rightarrow$ 7, 8, 9 & commit is absent

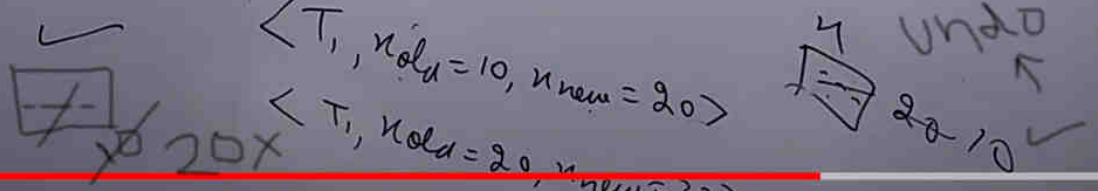
1st Undo $T_2 \Rightarrow 9 \to 8 \to 7$ then

redo $T_1 \Rightarrow 2 \to 3 \to 4 \to 5$

Q2 Why 1st undo & then redo



Q3 Why undo in backward direction & redo in forward direction?



Checkpoints:-

→ We discuss checkpoints for immediate updates (undo/redo algo).

→ If size of log file is very large then we have 2 problems:-

① Search process to make redo & undo list is time consuming

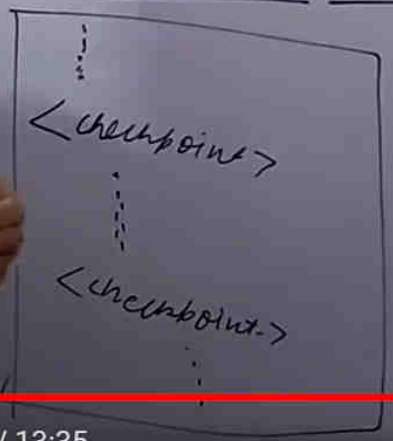
② Most of the transactions updates may be written to the database so if we redo them again then it is wastage of time.

→ Hence system do checkpoints from time to time which require 3 steps.

① Store all log records currently in RAM to stable storage.

② All modified buffer blocks are written to Database.

③ <checkpoint> log record is written to stable storage.



Note:- Transactions are not allowed to perform any update actions like writing to a buffer block or writing a log record while checkpoint is in progress.

Recovery in case of checkpoints.

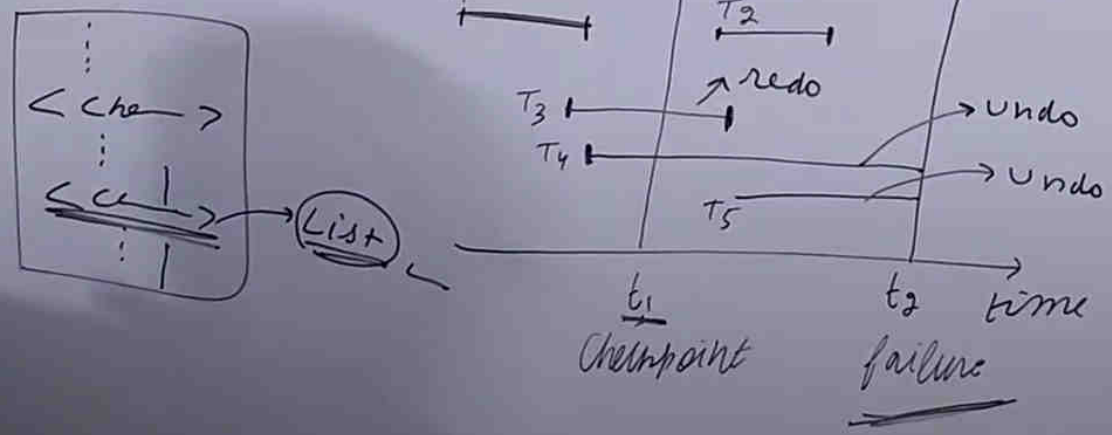
→ If T_i commits before <checkpoint> then we can ignore it.

→ Scan the log file in backward direction & find 1st checkpoint (actually it is last or most recent checkpoint)

→ DBMS will make list of transactions that were active during checkpoint

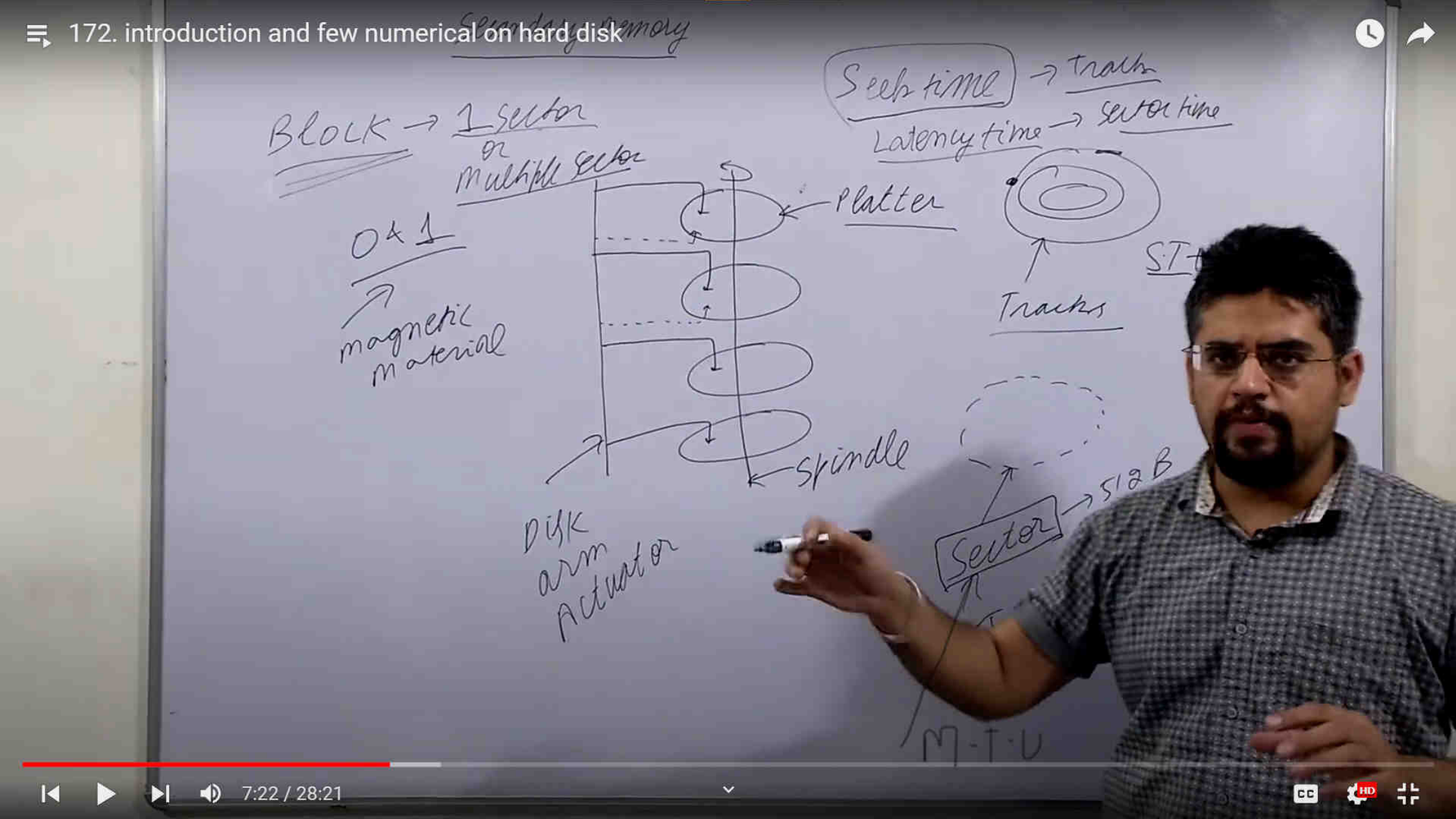
→ From this list make redo & undo lists

→



D → T_1 T_3 redo

R.I.U → T_4 T_5 undo


Secondary memory
Seek time → Track
Latency time → Sector time
BLOCK → 1 Sector or Multiple Sector
0 & 1
magnetic material
Platter
Tracks
S.T
Spindle
Disk arm Actuator
Sector
512 B
M·T·U

Secondary memory

Q1 Disk contains 100 tracks & 500 sectors find Average seek time & Average Latency time if disk is rotating @ 3600 rpm & time needed by read/write head to Jump from one track to another is 6 msec.

Ans Avg seek time = $\frac{1}{3} * \max^{m}$ seek time $= \frac{1}{3} * 99 * 6$ msec

$= \frac{1}{2} * \max^{m}$ seek time $= \frac{1}{2} * 99 * 6$ msec $\Leftarrow$ $\frac{1}{2} * (n-1) *$ Time Jump (n: No. of tracks)

$\frac{1 \text{ to } 1 + 1 \text{ to } 2 + 1 \text{ to } 3 + \cdots 1 \text{ to } 100}{100} =$

$\frac{0 + 1 + 2 + \cdots 99}{100} = \frac{99 * 100}{2 * 100} = 49.5$

$\frac{1}{2} *$ Revolution time

3600 $\rightarrow$ 60 sec

$\frac{1}{2} \rightarrow \frac{60}{3600} * \frac{1}{2}$ sec

Secondary memory

Q2 If track size is 50 KB (Bytes) disk rotation speed is 3600 rpm. Find transfer rate

Ans Amount of data transfered from HD to RAM in 1 sec is known as transfer rate.

3600 → 60 sec

1 → 60/3600 sec

60 sec / 3600 → 50 KB

1 sec → 50 * 60/60 KB

3000 KB per sec

3 MBps

3 MByte per sec

24 MBPS

Secondary memory

Q3 If track size is 50 KB → Bytes disk rotation speed is 3600 rpm. Find transfer rate & Block transfer time if block size is 1 KB

Ans once read/write head is placed at begining of correct block some time is needed to transfer data of block from HDD to RAM that time is Block transfer time.

Transfer rate = 3 MBPS

3000 KB → 1 sec

1 KB → $\frac{1}{3000}$ sec

Block size / Transfer rate

Time needed to find & Transfer Block from HDD to RAM

$S + L + \frac{B}{T.R.}$

Secondary Memory

Q4 If disk contain 200 tracks with track size of 50KB & Rotating at speed of 3600 rpm. What is the time needed to find & transfer 1 block of size 1KB from HD to RAM. Jump time = 6 msec

Ans

Find ⇒ A.S.T + A.L.T

$3600 \rightarrow 60$

$1 \rightarrow \frac{60}{3600}$

$\frac{1}{2} \rightarrow$

$$\text{Find} = \frac{1}{2} * \frac{199}{1000} * 6 + \frac{1}{2} * \frac{60}{3600}$$

$$\text{Transfer time} = \frac{\text{Block size}}{\text{T-R.}} = \frac{1\,KB}{3000\,KB} \text{ sec} \Rightarrow \boxed{\frac{1}{3000} \text{ sec}}$$

$$\frac{6}{1000} * \frac{1}{2} * 199 + \frac{1}{2} * \frac{60}{3100} + \frac{1}{3000}$$

File organization

Basic concepts:-

→ Database → Collection of files → Collection of records (tuples) → Collection of fields

marker

→ A table may be stored as single file or file may contain multiple tables or table may be stored as multiple files

Q How records are stored on H.D.?

name varchar(20);

Types of records

Fixed length records

→ Size of record is fixed

Block

R1 R2 R3 → Internal fragmentation

Inter record gap

→ Unspanned strategy is used

Variable length records

→ Size of record is not fixed ∵ size of field is not fixed like varchar(20) or File contain multiple tables or we don't want to store repeating values

Unspanned

Spanned

B1 B2

R1 R2 | R2 R3

→ Interblock gap

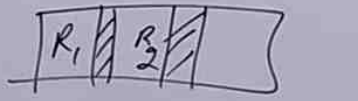
File organizations

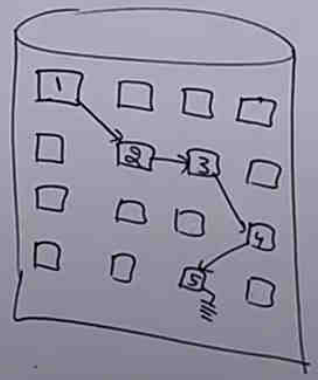
- Pile or Heap
- Sequential file
- Indexed Sequential file
- B^+ Tree
- B Tree

- Insertion of record is done by stacking
- Searching of record is brute force ie linear search so we have to read in block one by one & search each & every record in it.
- If we want to delete any record then 1st we have to search it & then we can delete it.
- Deletion is not physical.

Pile or Heap :-

- File is a Link list of blocks
- Block may contain multiple records
- If records in a file are unsorted then we call it as **Pile**
- Here records are Stacked up or piled up one after another.





Q1 If file is stored in 'B' blocks then on an average how many blocks we searched for perticular record?

Ans B/2

Q2 If we want to search records on the basis of field which can be repeated then on an average how many blocks are searched? B blocks

Ans B

Q3 In file searching is block by block (block search) or record by record (record search)

Ans Record search

→ Hence file can't be used for larger databases ∵ search is major operation which is inefficient in file.

Sequential file

→ Records are kept in sequential order i.e in sorted fashion on some attribute(s) & usually this attribute(s) is primary key.

→ Strict sequential file

bid	title	YR_PUB	
1	A	x	b1
3	B	y	
5	C	z	
8	–	–	b2
10	–	–	
15	–	–	
17	–	–	b3

(2, –, –)

Insertion → If we want to insert a new record then 1st we have to find its position & then it is stored at its proper place.

→ In worst case we have to read & write entire file (If insertion is at begining)

Deletion → Deletion is physical i.e records are shifted

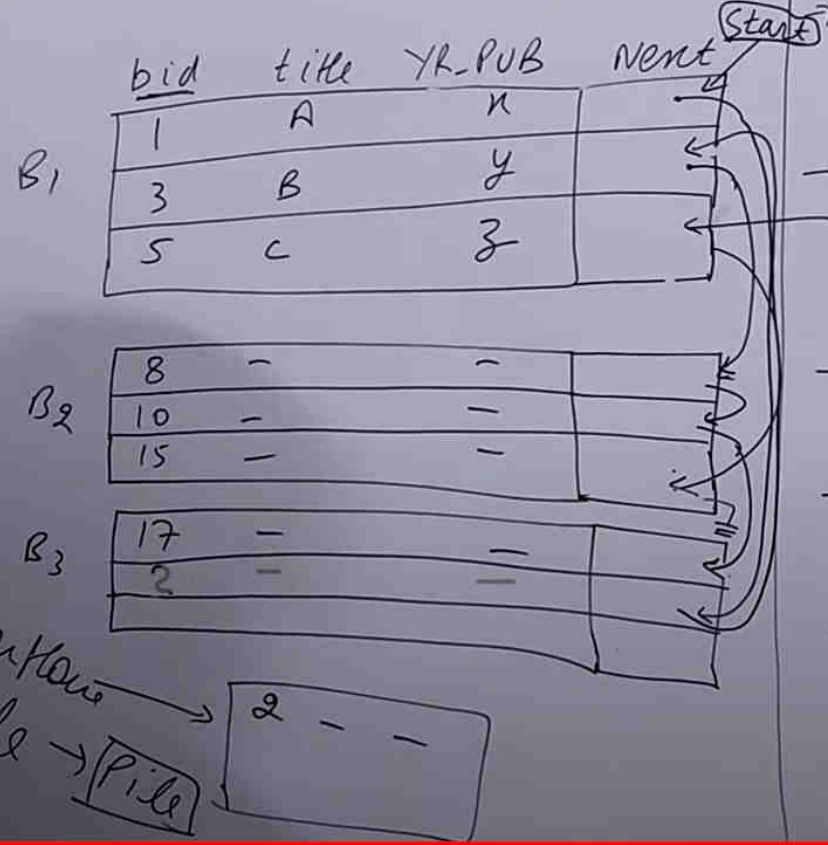
Searching → If we assume all blocks are present in RAM then apply binary search ⇒ $\lceil \log_2 B \rceil$ blocks are searched in worst case.

→ If blocks are not in RAM then still searching is efficient than file because here we have block search & not record search.

Normal sequential file

→ In Normal sequential file, we insert tuples in the end by changing 2 next pointer values.

→ Hence records are logically sorted but physically unsorted



→ To physically sort the records in file time to time reorganization is done

→ Deletion is not physical & we change only 2 next pointer values for deletion.

→ 2 Special pointers are used i.e Start & Available

→ Start always points to 1st record of file & it forms the linklist of sorted records

→ Available points to 1st record which is deleted → LL of deleted Records

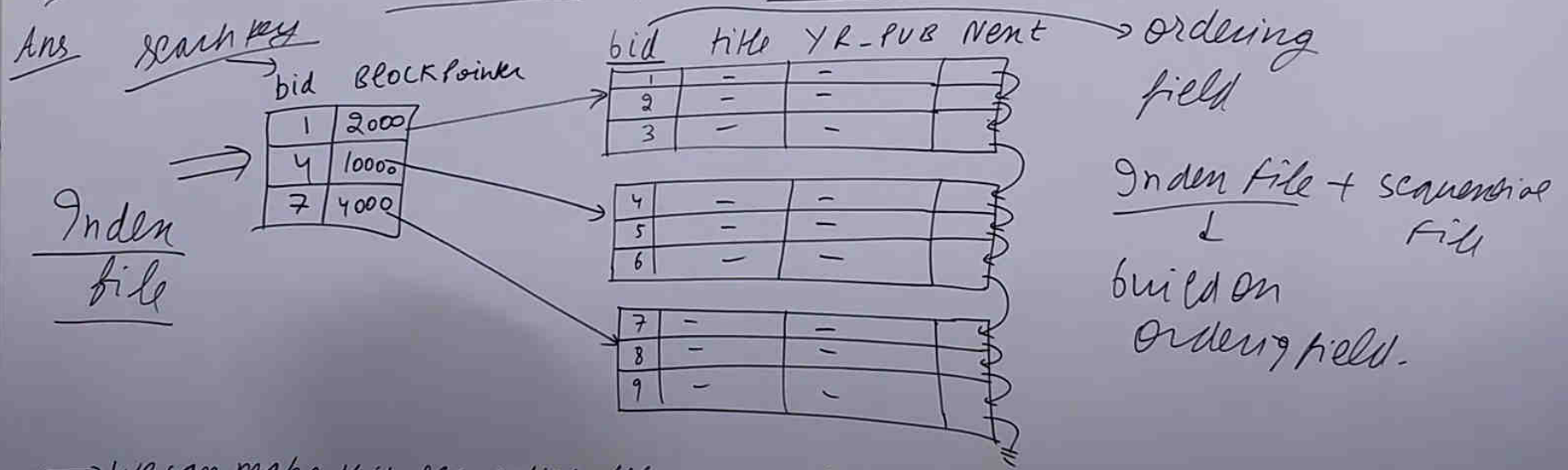
Indexed Sequential file :-

→ Main disadvantage of sequential file is that it is always sequential access ∵ of Link List
→ Hence Binary search can not be applied.
→ If we assume entire file is in RAM or if we assume file header contain address of all blocks of file then we can apply binary search.

Q. How to make sequential file as random access?

file discripor

Ans search key



→ We can make this sequential file as random access by mentaining a small file called inden file which contain only 2 columns i.e search key(field) & block(record) pointer
→ This file is always sorted according to search field value.

Indexed Sequential file :-

Search key :-

→ It is an attribute of table which is present in index file. It may or may not be primary key

→ Search key is a field on which index is build.

→ It may or may not be ordering field.

Q What is the maximum number of disk accesses required to fetch a record from indexed sequential file assuming index file is in RAM?

Ans 1

Q In above question if index file is in H.D & all pointers to index blocks are available then how many disk accesses?

Ans $\lceil \log_2 B_i \rceil + 1$

If not index file

$\lceil \log^B \rceil$ $B >> B_i$

Indexes

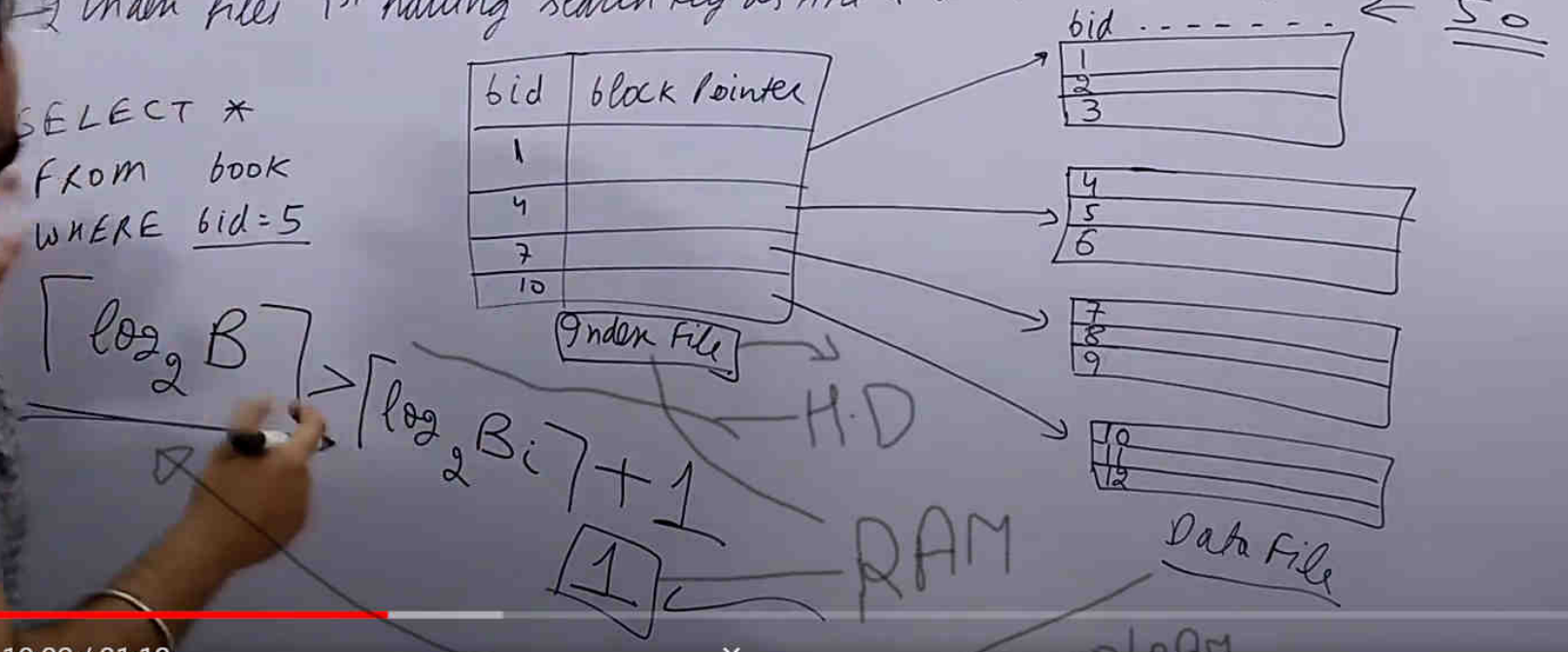
→ Indexes in database are similar to indexes at last of books.

→ They provide faster way of accessing records of file.

→ For a file we can have more than one index also & infact for every field of file we can build index (but not good idea)

→ If lots of query on book table involve conditions on title & bid then make 2 index files 1st having search key as title & 2nd has search key as bid.

```
SELECT *
FROM book
WHERE bid=5
```

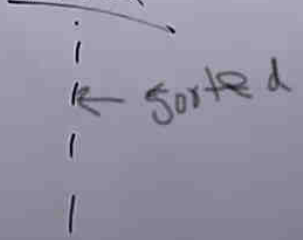


$\lceil \log_2 B \rceil \rightarrow \lceil \log_2 Bi \rceil + 1$

1

Types of Indexes

- Single level ordered Indexes
  - Index file always sorted acc. to search key
  - Primary Index
    - Index is build on ordering field which is candidate key of data file.
    - File is sorted acc. to C.K. & on that C.K. index is build
  - Clustered Index (Sorted)
    - Index is build on ordering field which is not C.K. of data file (i.e. repeating)
    - File is sorted acc. to non key & that non key is used to build index.
  - Secondary Index (Unsorted)
    - Index is build on non ordering field which may or may not be C.K.
- Multilevel Indexes
- Dynamic multilevel Indexes (B tree, B+ tree)



Types of Indexes

- Single level Ordered Indexes
- Multilevel Indexes
- Dynamic multilevel Indexes (B tree, B+ tree)

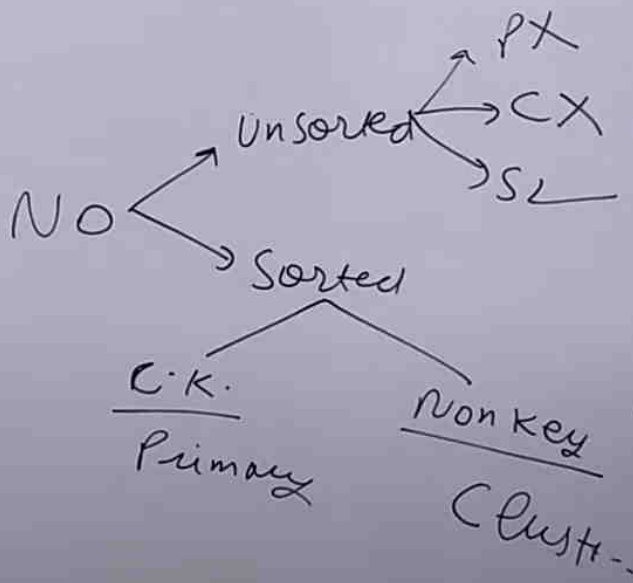
A file can have atmost one primary index

A file can have atmost one clustered index

A file can have any no. secondary index

A file can have both primary & clustered index

A file

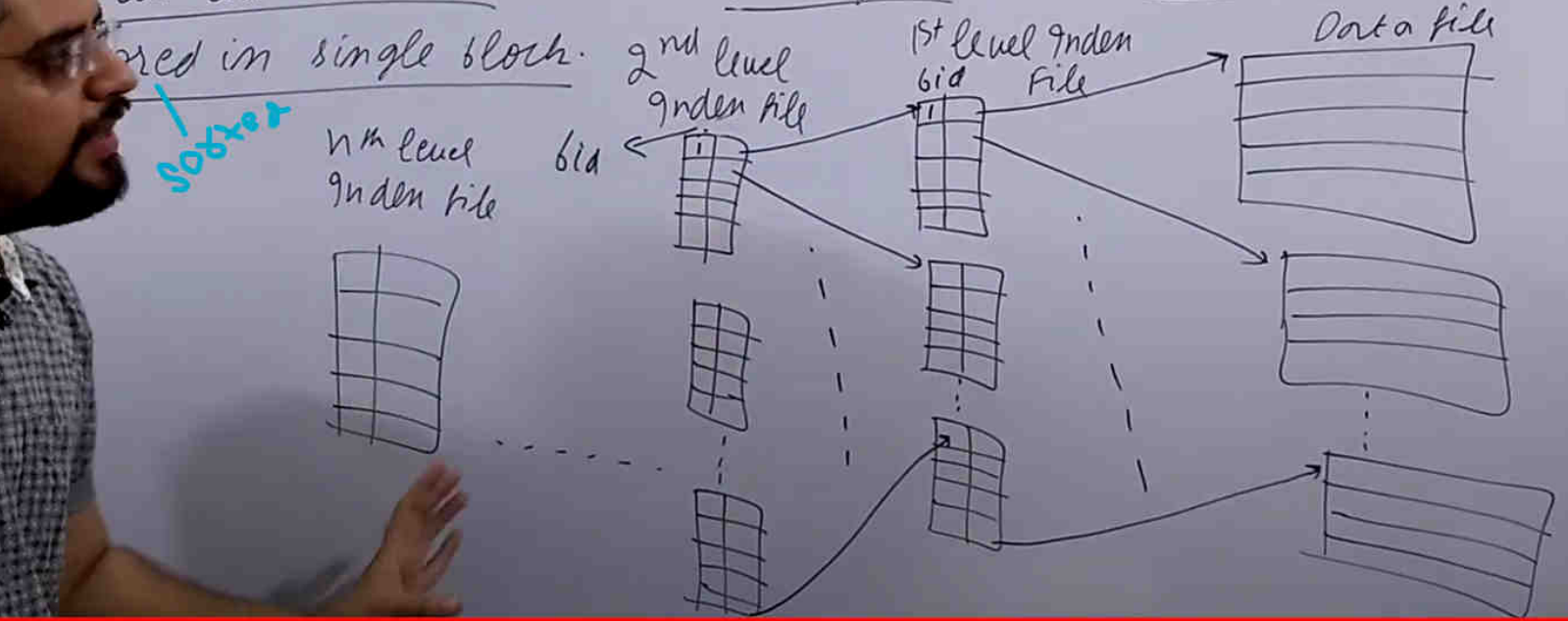


Types of Indexes

- Single level ordered Indexes
- Multilevel Indexes
- Dynamic multilevel Indexes (B tree, B+ tree)

Multilevel Indexes :-

Since index file at 1st level is sorted according to search key we can make index of index file itself till we get index file which can be stored in single block.



Types of Indexes

Dense

- Index entry for every search key value (Hence for every record)
- No. of records in index file = No. of records in Data file
- $B >> B_i$

Sparse

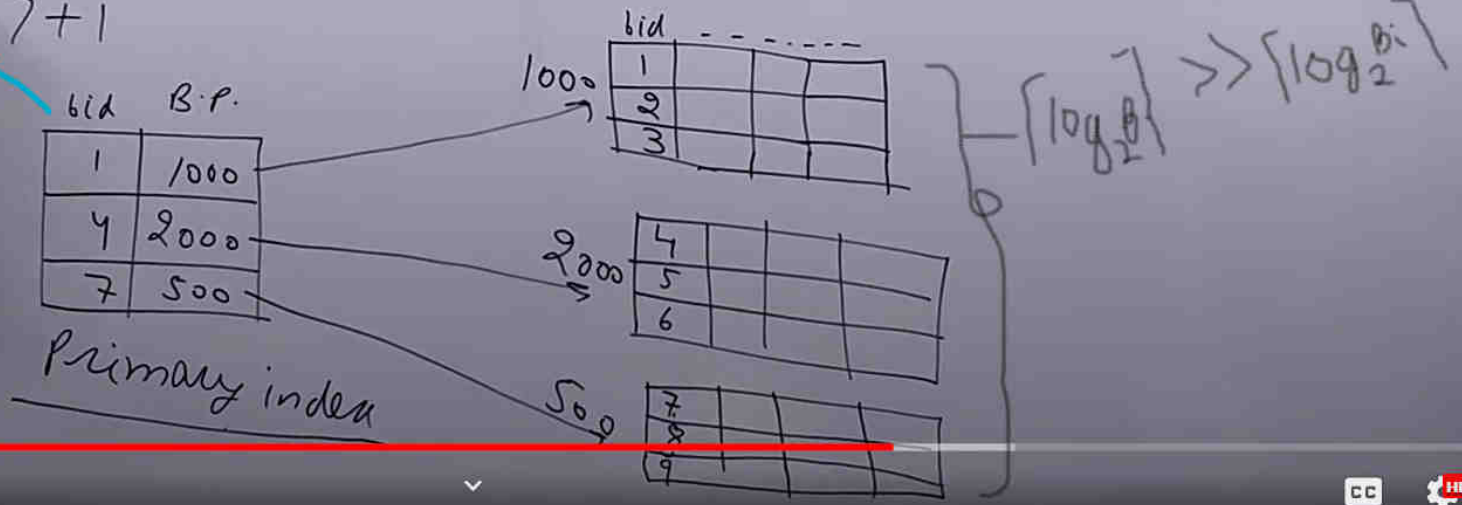
- Index entry for only some of search key values (not for every record)
- No. of record in index file < No. of records in data file
- $B >> B_i$
- $m < n$

Primary Index

→ Primary index is an ordered file with 2 fields i.e Search key & Block pointer
→ Search key is an ordering field of data file & this ordering field must be C.K.
→ There is one index entry (index record) for each block of data file
→ 1st record in each block of data file is called anchor record or block anchor
→ Primary index is non dense or sparse index ∵ index file has one entry for each block rather than each record.
→ $B >> B_i$ ∵ Size of each record is smaller in index file & No. of records are also less.
→ No. of Disk accesses required to fetch a record given a search key value will be

$\lceil \log_2 B_i \rceil + 1$

```
SELECT *
FROM book
WHERE bid = 5
```



A file consists of 30,000 UNSPANNED records, each record is of 100 bytes. Block size is 1024 bytes.
If primary indexing is used & block pointer is of 6 bytes & primary key is of 2 bytes

Q1 What is blocking factor for data file & index file
Q2 No. of blocks needed to store data file & index file
Q3 No. of block accesses to access data with & w/o indexing assuming all index & data block addresses are available but these blocks are in H.D.
How many block accesses if multilevel indexing is used.

Blocking factor → No. of records which can be stored in each Block

Q1

$$\left\lfloor \frac{\text{Block size}}{\text{Record size}} \right\rfloor \quad \left\lfloor \frac{1024}{8} \right\rfloor = \left\lfloor \frac{2^{10}}{2^3} \right\rfloor = 2^7 = 128$$

Data file = $\left\lfloor \frac{1024}{100} \right\rfloor = 10$

= index file

A file consists of 30,000 UNSPANNED records, each record is of 100 bytes. Block size is 1024 bytes.
If primary indexing is used & block pointer is of 6 bytes & primary key is of 2 bytes

Q1 What is blocking factor for data file & index file
Q2 No. of blocks needed to store data file & index file
Q3 No. of block accesses to access data with & w/o indexing assuming all index & data block addresses are available but these blocks are in H.D.
Q4 How many block accesses if multilevel indexing is used.

Blocking factor for index file $BFR_i = 128$
" " " data file $BFR_D = 10$

No. of blocks for data file $= \left\lceil \frac{30,000}{10} \right\rceil = 3000$

" " " index file $= \left\lceil \frac{3000}{128} \right\rceil = 24$

$\lceil \log_2 B_i \rceil + 1$
$\Downarrow$
$\lceil \log_2 24 \rceil + 1$
$\rightarrow 5 + 1 = 6$

Data file
$\lceil \log_2 B \rceil$
$\lceil \log_2 3000 \rceil$
12

A file consists of 30,000 UNSPANNED records, each record is of 100 bytes. Block size is 1024 bytes.
If primary indexing is used & block pointer is of 6 bytes & primary key is of 2 bytes

Q1 what is blocking factor for data file & index file
Q2 No. of blocks needed to store data file & index file
Q3 No. of block accesses to access data with & w/o indexing assuming all index & data block addresses are available but these blocks are in H.D.
Q4 How many block accesses if multilevel indexing is used.

No. of blocks for Data file = 3000
" " " index " = 24

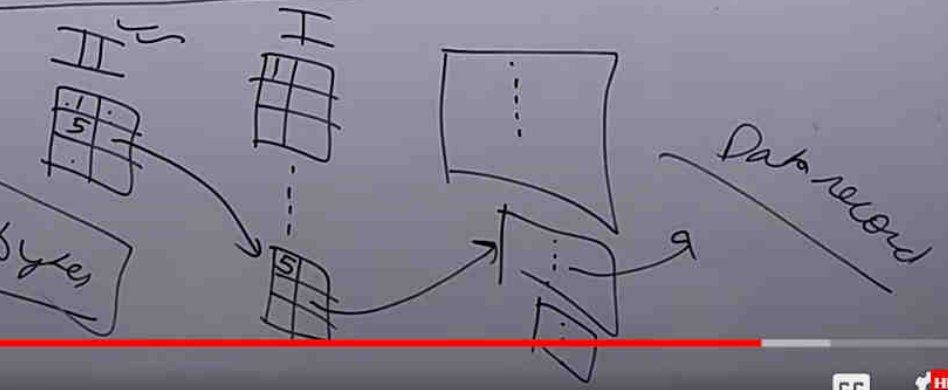
12 → 6 → 3
PI MI
n+1

→ Since index file is stored in more than one block so we can do indexing of index file itself which is known as multilevel indexing & we will stop when we get index file of one block.

3 ← 1 + 1 + 1

24 × 8 = 192 bytes

II I Data file



Clustered Index :-

→ A clustered index is an ordered file with 2 fields i.e search key & block pointer

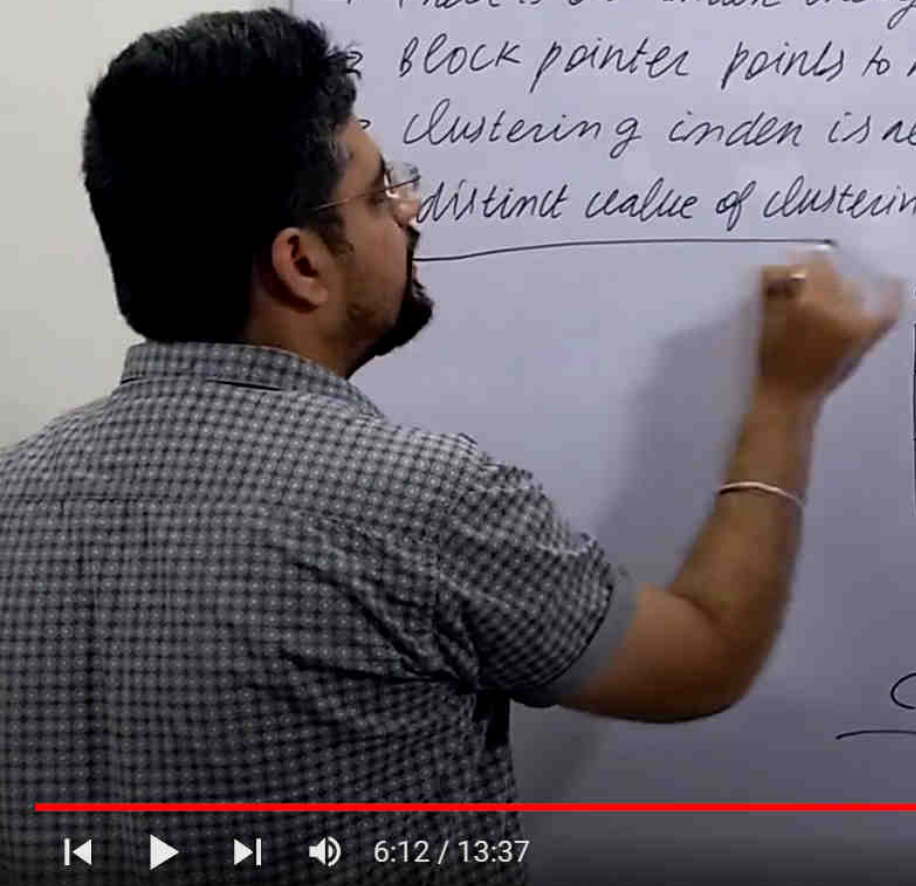
→ Search key is an ordering field of file & it does not have unique value for each record i.e it must be non key.

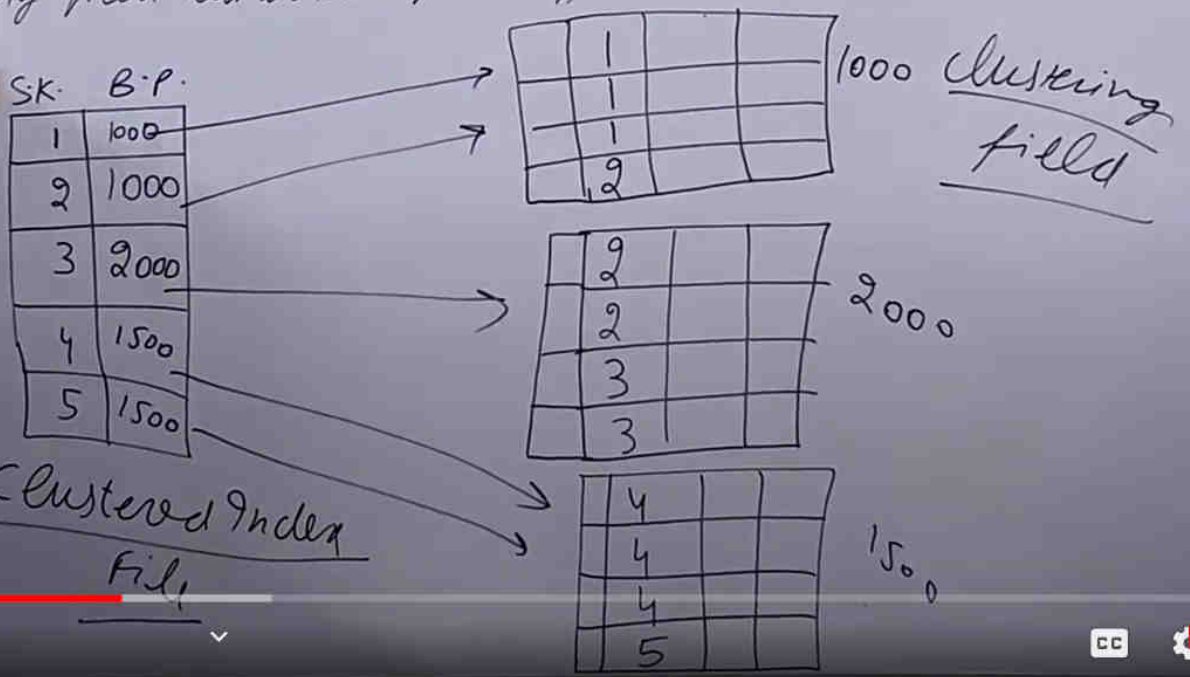
→ We call this ordering field as clustering field ∵ it is used to make clusters of records

→ Their is one index entry (index record) for each distinct value of clustering field.

→ Block pointer points to the block containing 1st record with respective search key value.

→ Clustering index is also non dense or sparse ∵ it has index entry for every distinct value of clustering field rather than for every record of file.



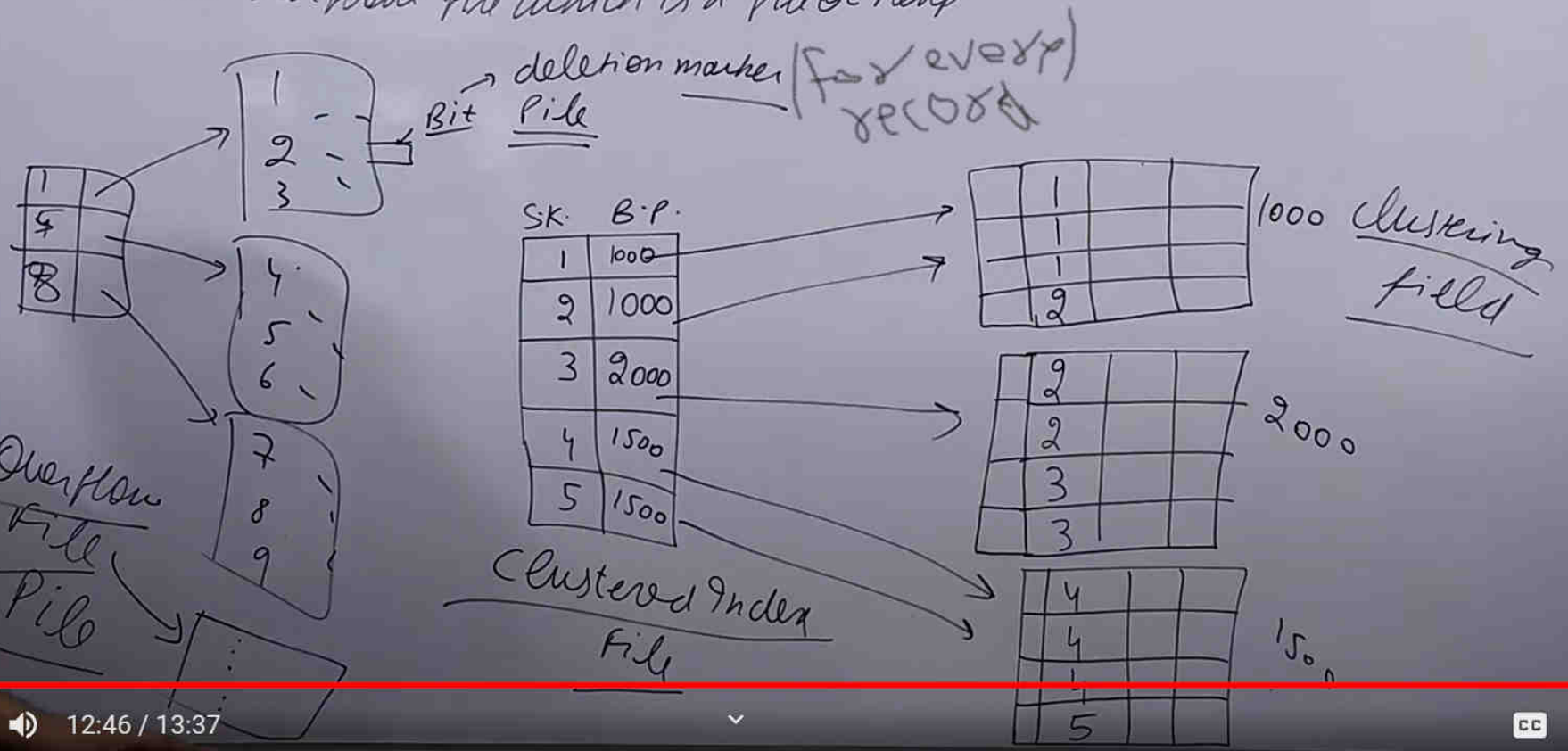


Clustered Index :-

→ Insertion & deletion of records in both primary & clustered indexing is time consuming.

→ In clustered indexing we maintain a seperate block or cluster of contigous blocks for each unique clustering field value.

→ In primary indexing deletion is handled by deletion markers & for insertion we can use overflow file which is a pile or heap.



A file consist of 30000 unspanned records each of size 100 bytes. Block size is 1024 bytes

If clustered indexing is used on a field such that every value is repeated 6 times. Size of block pointer is 6 bytes & clustering field value is of 9 bytes

Q) What is blocking factor for data file & index file

No. of blocks needed to store index file & data file

No. of block access to access data assuming index file is in R.D. & all block addresses available

How many block access if multilevel indexing is used.

$BFR_D = 10$

$BFR_i = 68$

No. of blocks needed to store Data file $= \lceil \frac{\text{No. of records}}{BFR_D} \rceil = \lceil \frac{30,000}{10} \rceil = 3000$

" " " " index file $= \lceil \frac{5000}{68} \rceil = 74$

5000 unique

30000 / 6

A file contains 30,000 unspanned records each of size 100 bytes. Block size is 1024 bytes. If clustered indexing is used on a field such that every value is repeated 6 times. Size of block pointer is 6 bytes & clustering field value is of 9 bytes

Q1 What is blocking factor for data file & index file

Q2 No. of blocks needed to store index file & data file

No. of block access to access data assuming index file is in R.O. & all block addresses are available.

How many block access if multilevel indexing is used.

$\lceil \log_2 74 \rceil + [K] \Rightarrow 2$

$7 + 2 = [9]$

No of blocks in which records having respective value of search key are present

Blocking factor = 10

A file consists of 30,000 unspanned records each of size 100 bytes. Block size is 1024 bytes

If clustered indexing is used on a field such that every value is repeated 6 times. Size of block pointer is 6 bytes & clustering field value is of 9 bytes

Q1 What is blocking factor for data file & index file

Q2 No. of blocks needed to store index file & data file

Q3 No. of block access to access data assuming index file is in M.D. & all block addresses are available.

Q4 How many block access if multilevel indexing is used.

BFR$_i$ ⇒ 68

III II I D

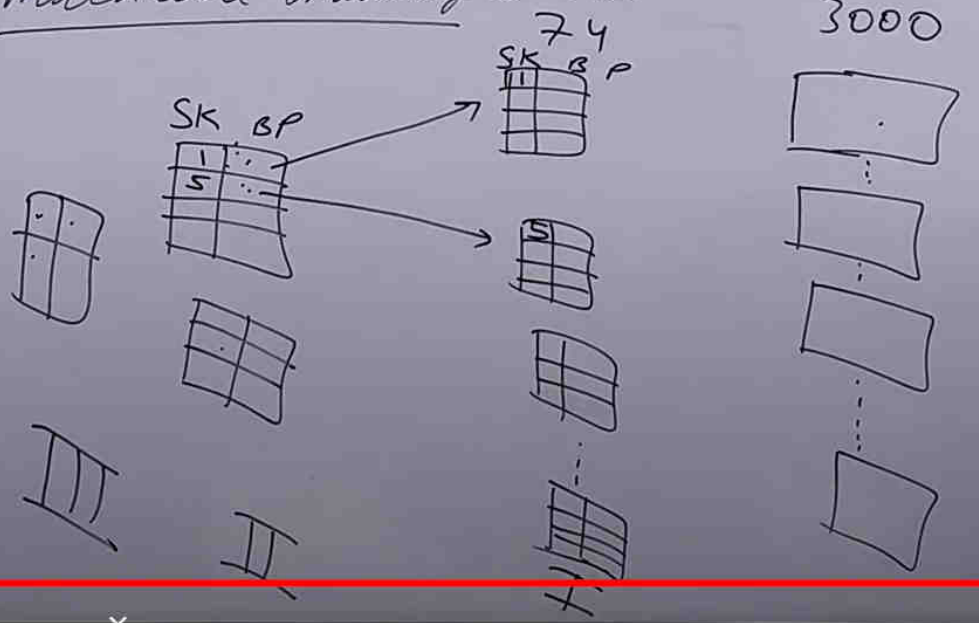
1 1 1 1

1 + 1 + 1 + k

k = 2

$$\left\lceil \frac{74}{68} \right\rceil = 2$$

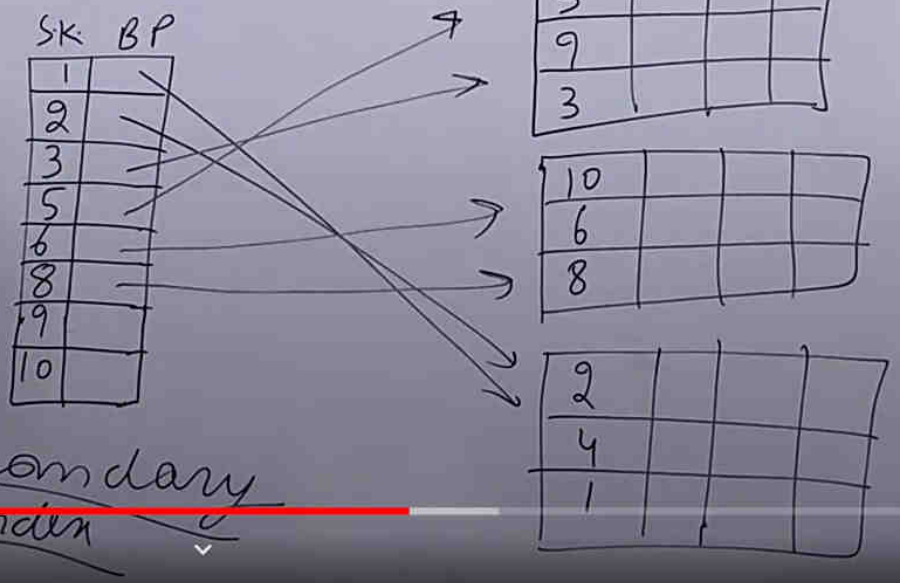
= 5



Secondary Index :-

→ Secondary index is an ordered file having 2 fields i.e search key & block or record pointer
→ search key is a non ordering field & it may or may not be candidate key.
→ Hence their can be many secondary indexes for a file
→ Record pointer is a block pointer appended with some offset
→ Secondary index on non ordering key field (Unique) is always dense i.e index file has one index entry for each record of data file.
→ Index entry contain value of this key field & record or block pointer
→ Why dense? ✓✓

BP = 7 bytes
Block → 256 records
8 bit
K.P. = 7+1 = 8 bytes



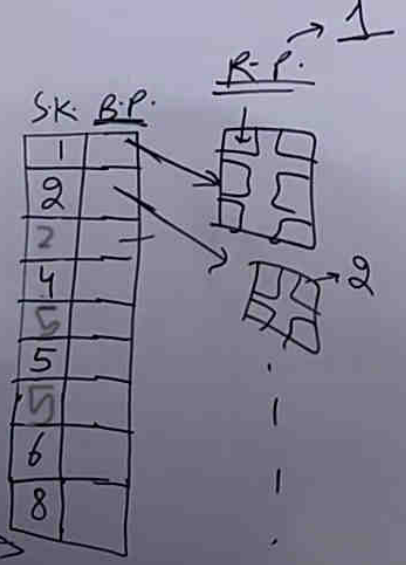
Secondary index

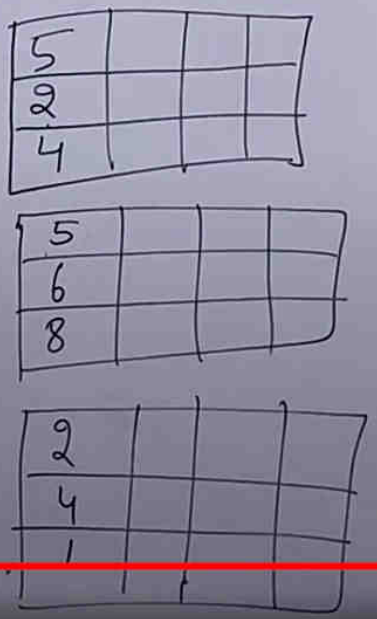
Secondary index :-

→ If secondary index is build on non ordering field which is repeated ie non key then we have many options of creating secondary index.

→ But we can't skip any record.

Secondary index ↔ Key
Non key ×





Secondary index build

File consists of 30,000 unspanned records, each of size 100 bytes, blocksize = 1024 bytes
secondary index is build on a field which is unique of size 6 bytes & block pointer = 9 bytes

Q1 What is blocking factor for data file & index file
Q2 No. of blocks needed to store data file & index file
Q3 No. of block accesses to access data with or without indexing assuming all index & data block addresses are available but these blocks are in H.D.
How many block accesses if multilevel indexing is used.

$BFR_i = 68$

$BFR_D = 10$

$\lceil \frac{30,000}{10} \rceil = 3000$

$\lceil \frac{30,000}{68} \rceil = 442$

File consists of 30,000 Unspanned records, each of size 100 bytes, blocksize = 1024 bytes
Secondary index is build on a field which is unique of size 6 bytes & block pointer = 9 bytes

Q1 What is blocking factor for data file & index file
Q2 No. of blocks needed to store data file & index file
Q3 No. of block accesses to access data with or without indexing assuming all index & data block addresses are available but these blocks are in H.D.
Q4 How many block accesses if multilevel indexing is used.

No. of blocks for data file = 3000
" " " " index " = 442

with indexing $\Rightarrow \lceil \log_2 442 \rceil + 1$
$\Rightarrow 9 + 1 = 10$

w/o indexing $\Rightarrow$
$\Rightarrow$ 3000
1500 Avg
Worst

183. numerical on secondary indexing

File consists of 50,000 Unspanned records, each of size 100 bytes, blocksize = 1024 bytes
Secondary index is build on a field which is unique of size 6 bytes & block pointer = 4 bytes

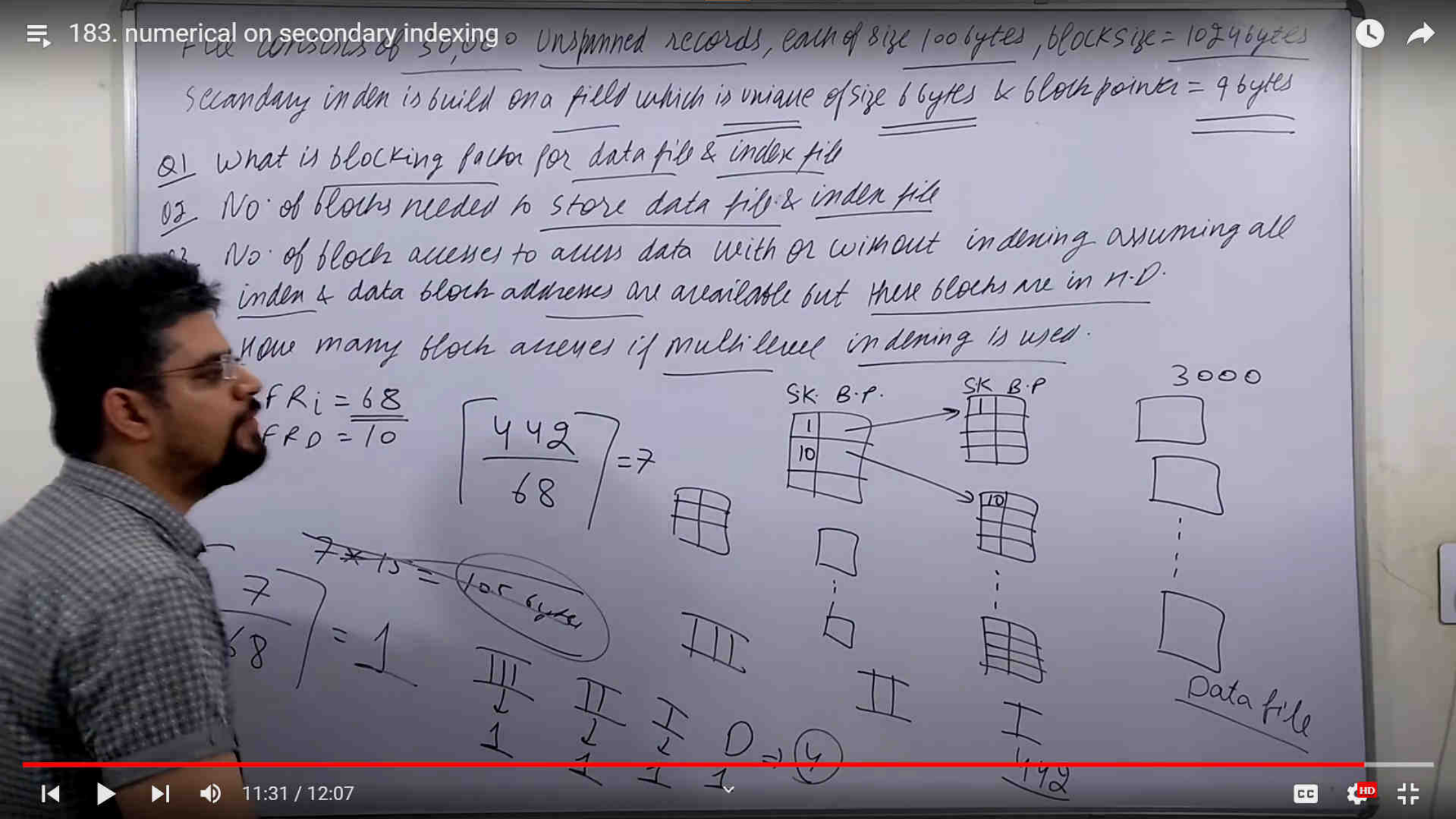
Q1 What is blocking factor for data file & index file
Q2 No. of blocks needed to store data file & index file
Q3 No. of block access to access data with or without indexing assuming all index & data block addresses are available but these blocks are in H.D.
How many block access if multilevel indexing is used.

$fR_i = 68$
$fR_D = 10$

$\left\lceil \frac{442}{68} \right\rceil = 7$

$\left\lceil \frac{7}{68} \right\rceil = 1$

7 * 15 = 105 bytes



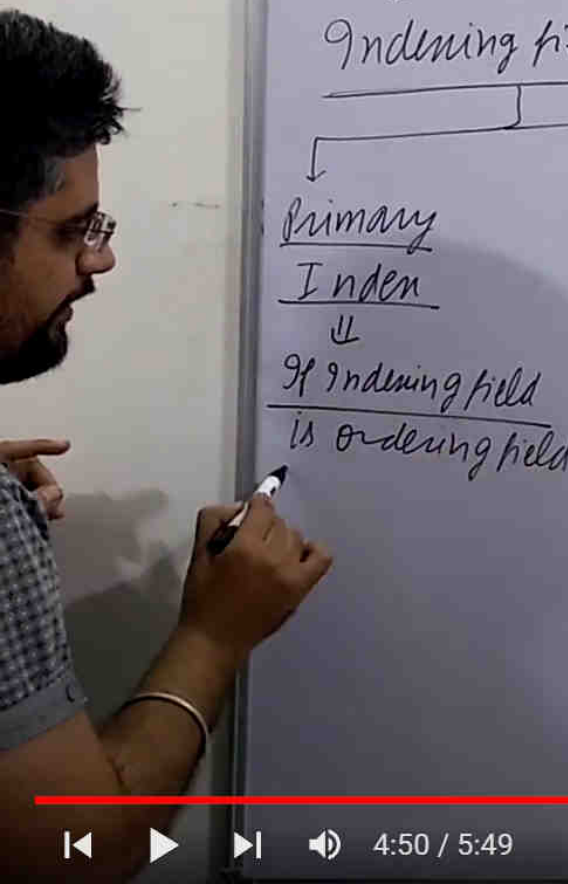
Summary

Type of Index	No. of (1st level) Index Entries	Dense or Sparse	Block anchoring on Data file
Primary	No. of blocks in Data file	Sparse	Yes
Clustering	No. of distinct index field values	Sparse	Yes/No
Secondary (Key)	No. of records in Data file	Dense	No
Secondary (Non Key)	—	Dense/Sparse	No

Summary

Types of indexes

- Indexing field is key
  - Primary Index
    - If indexing field is ordering field
  - Secondary Index (key)
    - ✗
- Indexing field is non key
  - Clustering index
    - Indexing field is ordering field
    - ✗
  - Secondary Index (non key)



→ If 1st level index file (Primary, Secondary, Clustering) is very large then applying binary search on it is time consuming.

→ Hence multilevel indexing is used in such cases to decrease search time from $\log_2 n$ to $\log_{BFR_i} n$

Fanout = BFRi

No of blocks at 1st level = $\frac{n}{BFR_i}$

$BFR_i > 2$

$\lceil \log_{FO} n \rceil$

$\log_2 8 \Rightarrow 3$

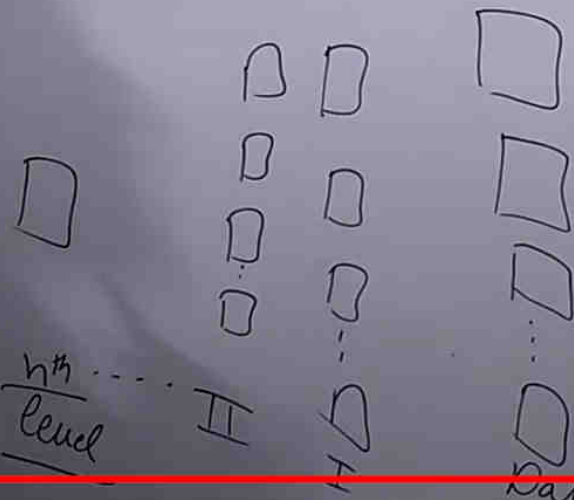
m

$\log_2 n >> m$

Multilevel Indexing

Static Multilevel Indexing

- Insertion & deletion of records from data file will lead to change in index files at various levels & making these changes is time consuming



Dynamic Multilevel Indexing

- Insertion & deletion of records from data file may or may not lead to change in index file at various levels & we have efficient algorithms to handle these problems.
  - B tree index files
  - B^+ tree index files

188. B-tree introduction

B tree → Balanced → All leaf nodes are at same level

→ Index file is stored in B tree D.S so that searching, insertion & deletion of index entry becomes efficient.

→ Assumption → field on which index is build must be key ie non repeating (Unique)

$\lceil \log_2 Bi \rceil$ → used to store index file

Properties of B Tree

① Root of B Tree can have children b/w 2 & p where p is called order of tree
order means maximum No of children a node can have

② Internal nodes can have children b/w $\lceil p/2 \rceil$ & p & keys b/w $\lceil \frac{p}{2} \rceil - 1$ & $p-1$

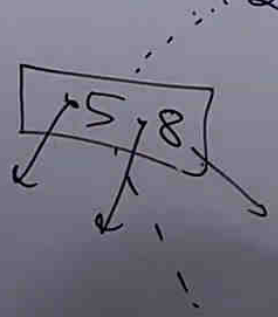
③ Structure of internal nodes & leaf nodes are same

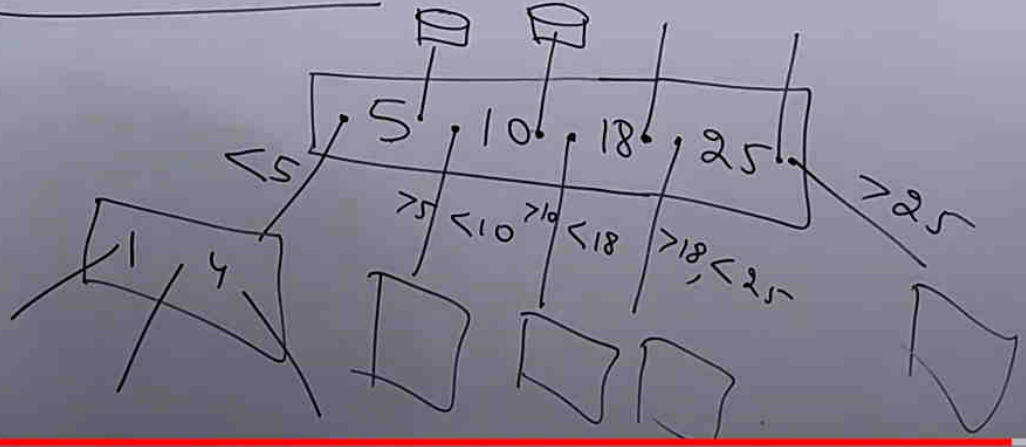
$< P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, P_3 <K_3, Pr_3> \cdots [K_{p-1}, Pr_{p-1}> P_p >$

④ Leaf nodes can have keys b/w $\lceil \frac{p}{2} \rceil - 1$ to $p-1$ & 0 children.

⑤ Keys must be present in sorted order

$p = 5 \Rightarrow \lceil \frac{5}{2} \rceil = 3$





Q. Find Min. & Max. No. of keys & children for root, internal, & leaf nodes of B Tree of order a) 5, b) 10

Order = 10

Children		Keys	
Min	MAX	Min	MAX
2	10	1	9
5	10	4	9
0	0	4	9

ROOT
internal
leaf

$\lceil \frac{p}{2} \rceil$

Consider a B Tree in which

Search key = 15 Bytes

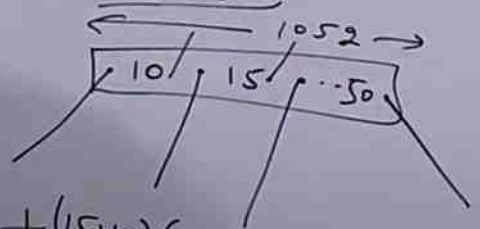
Block size = 1052 Bytes

Record pointer = 14 Bytes

Block pointer = 7 Bytes

Calculate order ⇒ max^m No. of children a node can have

Node size = Block size



$$7*P + (15+14)(P-1) \leq 1052$$

$$7P + 29P - 29 \leq 1052$$

$$36P \leq 1059 + 29$$

$$36P \leq 1088$$

$$P \leq \frac{1088}{36} \Rightarrow 30.2$$

P = 30

root

	Min. Node	Min. Key	Min. Pointers
Level 0	1	$\lceil \frac{P}{2} \rceil - 1$	2
Level 1	2	14×2	15×2
Level 2	30	14×30	15×30
Level 3	450	14×450	15×45

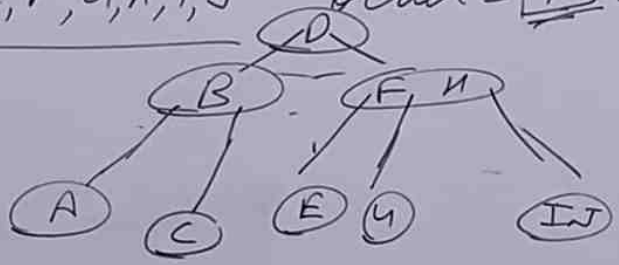
	Max Nodes	Max keys P-1	Max Pointers
Level 0	1	29	30 ← P
Level 1	30	29×30	30×30
Level 2			
Level 3			

B tree Insertion

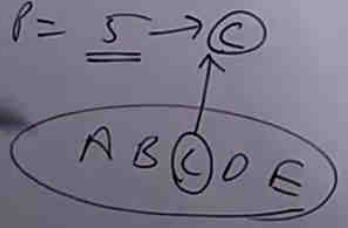
→ Insertion is always done at leaf. Whenever we have overflow then split the node & push median to parent.

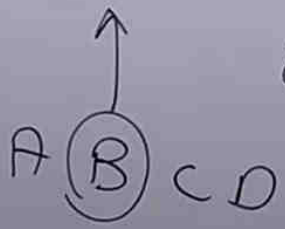
Ex A, B, C, D, E, F, G, H, I, J order = 4 ⇒ max^m ⇒ 3 overflow

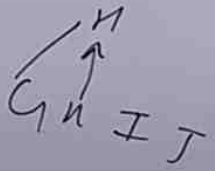
P ⇒ key > P → overflow

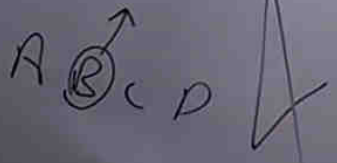


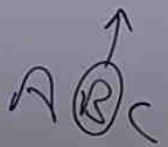
Left biased =
Right biased











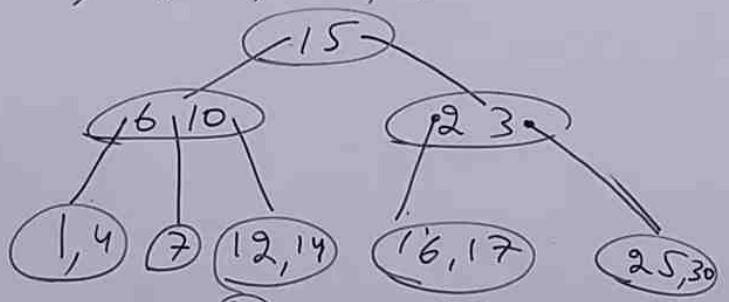
B tree Insertion

→ Insertion is always done at leaf. Whenever we have overflow then split the node & push median to parent.

Ex 7, 12, 15, 16, 1, 10, 14, 6, 4, 17, 23, 25, 30

order = 4
Right biased

underflow ✗

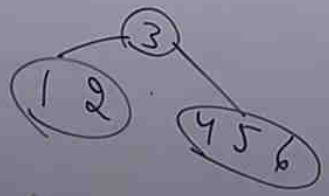


$\lceil \frac{p}{2} \rceil = 2$

P → odd ⇒ 5 ⇒

$\lceil \frac{5}{2} \rceil = 3$

even ⇒ 6



$\lceil \frac{6}{2} \rceil = 3$ — key

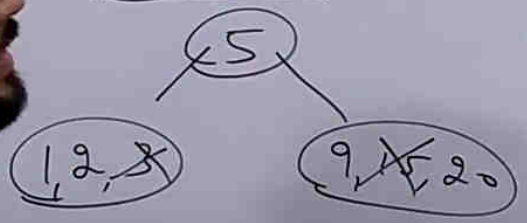
Deletion in B tree

→ Deletion is always done at leaf. If key to be deleted is in internal node then swap it with immediate succesor or predecessor

→ If underflow occurs at leaf then take help from left or right siblings.

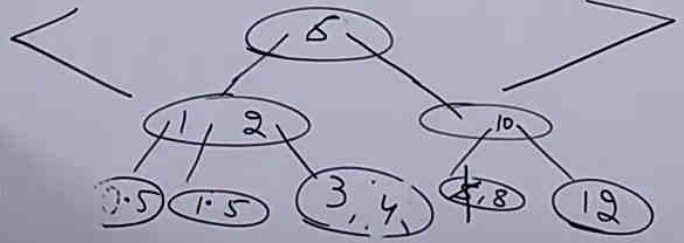
→ If left & right siblings already have minimum No of keys then merge given leaf with its left or right sibling along with one key from parent.

→ If underflow in parent then do same process recursively



Delete = 3, 15

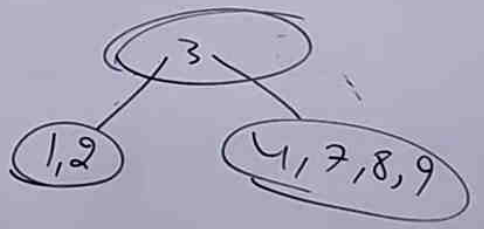
Order = ④ $\lceil \frac{4}{2} \rceil$ = ② = 1



Delete = 5

Order = 4

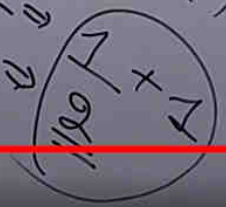
② ⇒ 1



Delete = 6

Order = 5

$\lceil \frac{5}{2} \rceil$ = 3 ⇒ ②

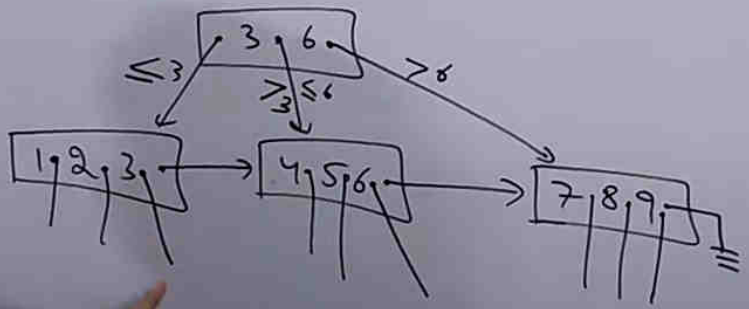


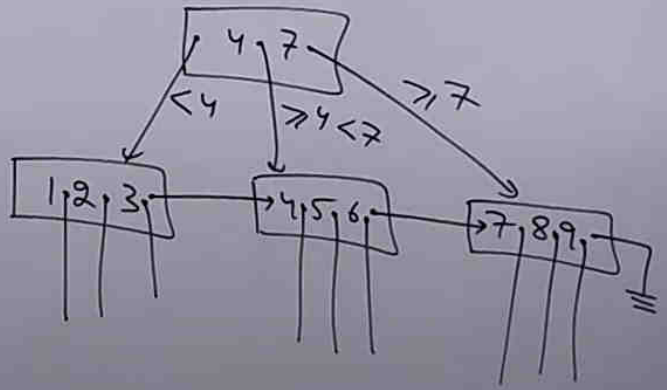
B^+ tree (Similar to B tree)

→ Here record pointer is missing in internal node. Hence internal nodes can have more no of childrun ie order ↑↑ & disk ↓↓

→ Leaves have record pointer, so structure of leaf & internal nodes is diffrent

→ In leaf node last pointer is not record pointer but it is block pointer which points to next leaf node & it help in range queries





B^+ Tree

→ Order of leaf nodes & internal nodes are diffent in B^+ Tree (unlike B Tree)

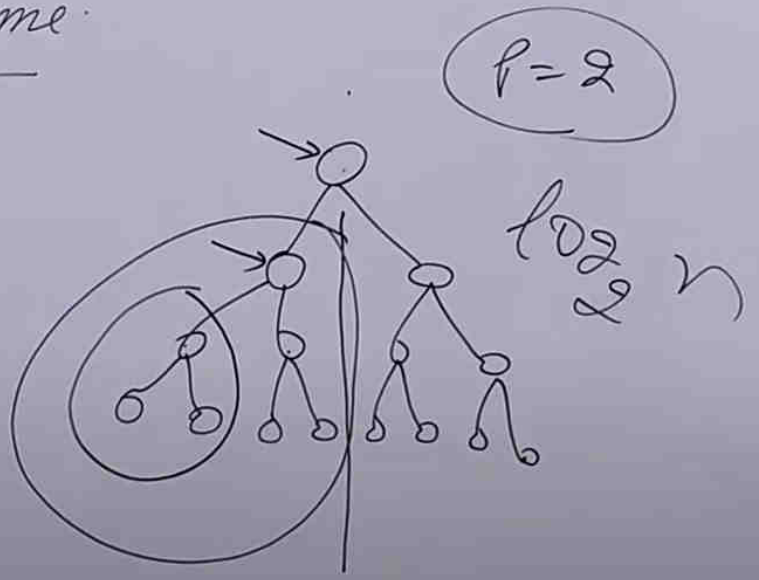
→ Searching a key needs $O(\log_p n)$ where p is order. Order of B^+ tree > Order of B tree

Hence searching is efficient in B^+ tree. So B^+ tree is mostly used for multilevel indexing..

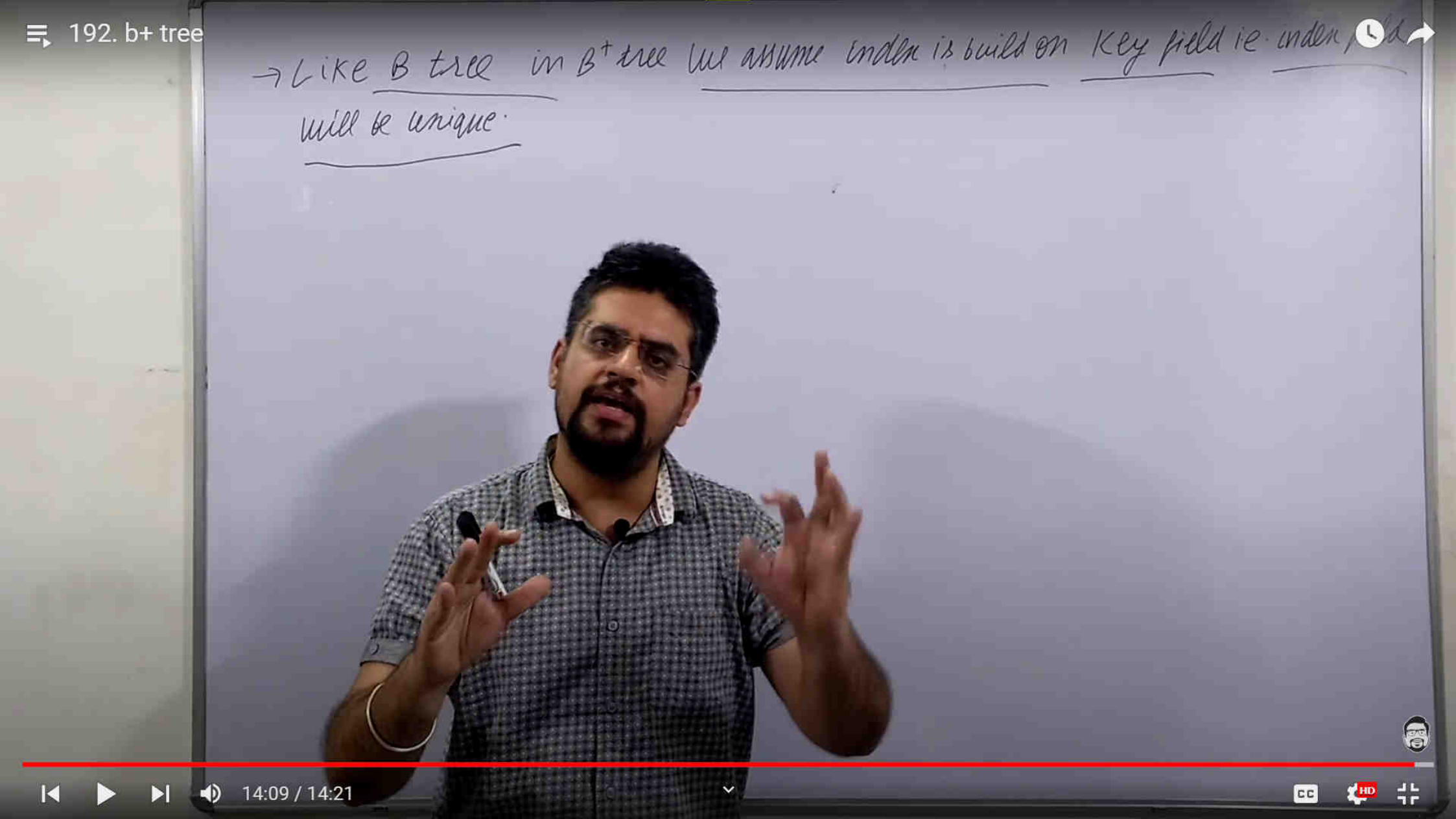
→ Properties of B tree & B^+ tree are same.

$\log_{50} n$

$\lceil \frac{p}{2} \rceil$ $\log_{100} n$



192. b+ tree



14:09 / 14:21

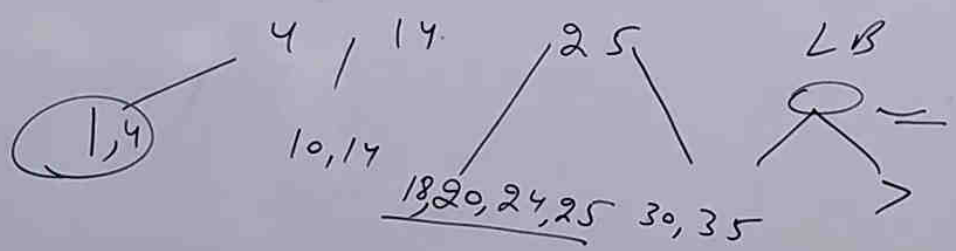
B^+ Tree insertion

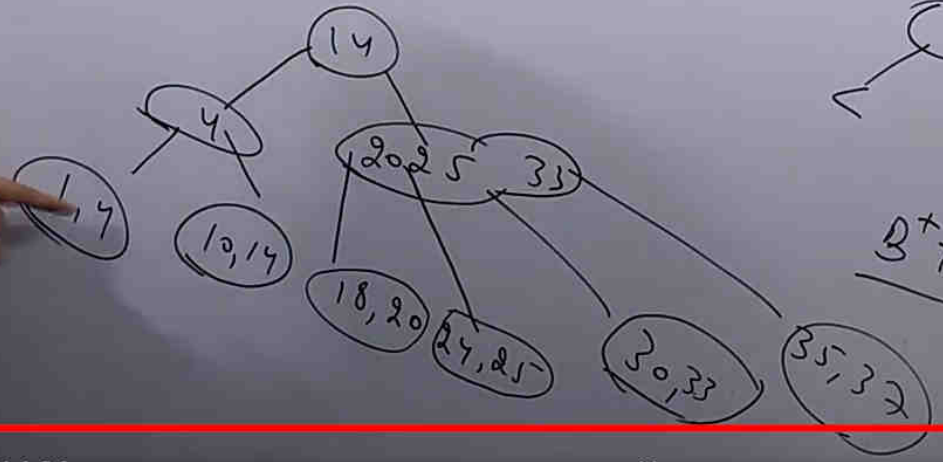
→ Insertion done at leaf (like B Tree). But here difference is that in case of overflow in leaf node median go to parent & we keep it in leaf node also

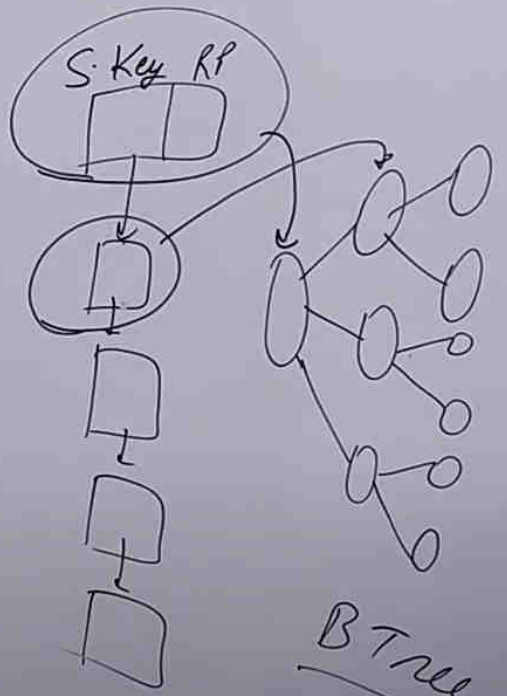
→ In case of overflow in non leaf node median go to parent but don't copy it.

1, 4, 10, 14, 20, 25, 35, 30, 18, 24, 33, 47

Order = 4







B^+ Tree

B Tree

Index file

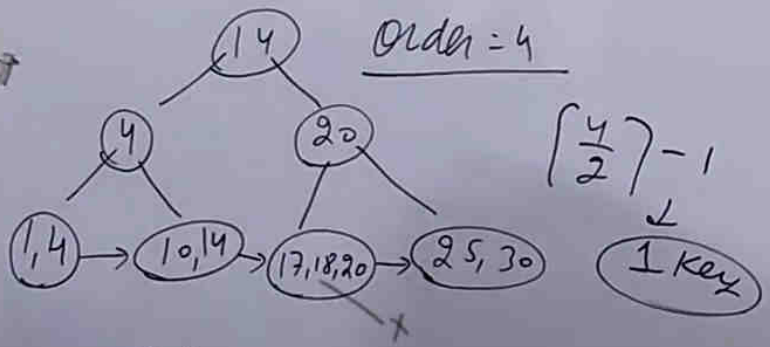
B+ Tree Deletion

→ Similar to B tree but key might appear in almost 2 nodes (leaf, internal or root)

→ Delete entry from root & handle underflow like B tree

→ If entry present in internal node then replace it with immediate successor or immediate predecessor ⇒

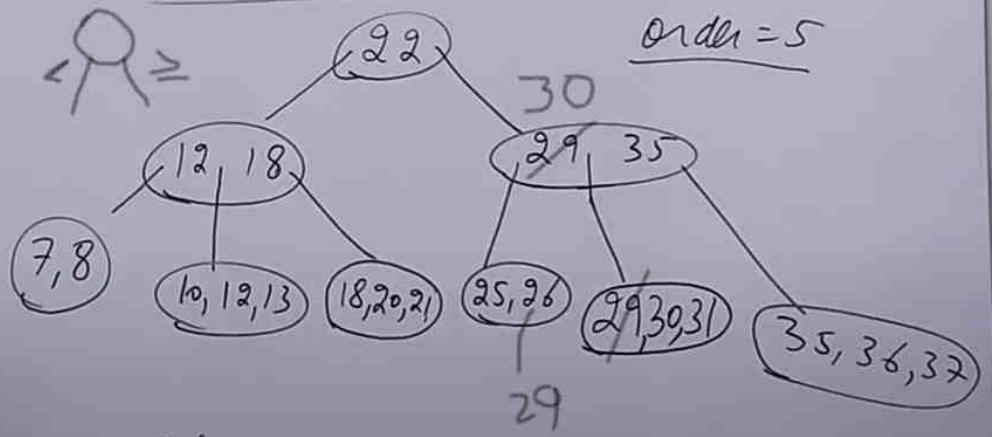
Case 1 Key only in leaf & No underflow



$\lceil \frac{4}{2} \rceil - 1$ ↓ 1 key

Delete → 18

Key only in leaf & underflow



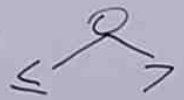
Delete → 26

B+ Tree Deletion

→ Similar to B tree but key might appear in almost 2 nodes (leaf, internal or root)

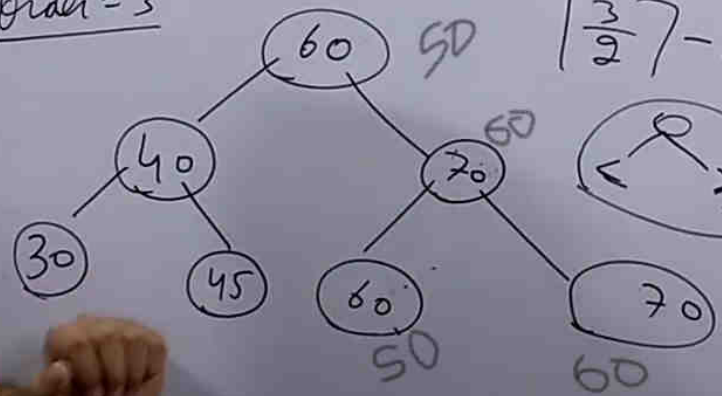
→ Delete entry from root & handle underflow like B tree

→ If entry present in internal node then replace it with immediate successor or immediate predecessor ⇒



Case 3 Delete key present in both leaf & internal node

Order = 3



$\lceil \frac{3}{2} \rceil - 1 \Rightarrow 2 - 1 = 1$ ← key

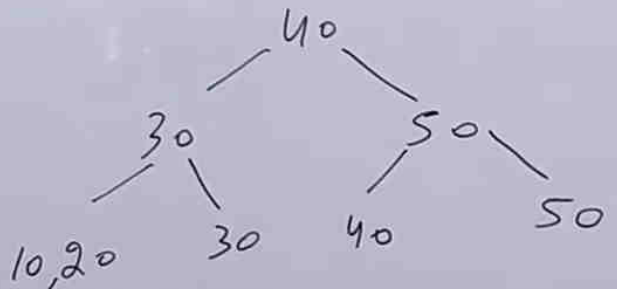
Delete → 50

1st delete leaf

2nd " " internal

GATE2007

B+ Tree → atmost 2 keys & 3 links



Insert 15, 25

Delete 50

GATE 2005

Table T_1 → 2000 records stored in 80 blocks ⇒ 1 block 25 Records

Table T_2 → 400 records stored in 20 blocks ⇒ 1 block 20 Records

Main memory can store 1 block of T_1 & 1 block of T_2

Nested loop Join algo → Record by Record

```
for (i=1; i<=2000; i++)
    for (j=1; j<=400; j++)
        Join operation
```

Block Nested loop Join algo → Block by Block

```
for (i=1; i<=80; i++)
    for (j=1; j<=20; j++)
        Join operation
```

```
for (i=1; i<=400; i++)
    for (j=1; j<=2000; j++)
        Join operation
```

```
for (i=1; i<=20; i++)
    for (j=1; j<=80; j++)
        Join operation
```

T_1 T_2

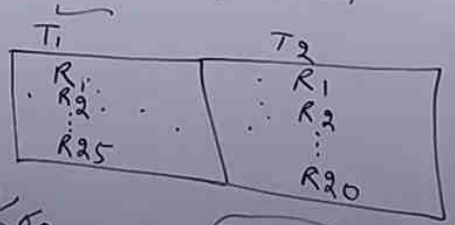
1 → 20

80 → 80 × 20 ⇒ 1600 + 80 = 1680

T_2 T_1

1 → 80

20 → 80 × 20 = 1600 + 20 = 1620



→ 32020