COMPUTER SCIENCE

Database Management System

Transaction & Concurrency Control

Lock Based Protocol - Part - 03

Lecture_10

Vijay Agarwal sir

TOPICS TO BE COVERED

01 Lock Based Protocol

02 Time Stamp Protocol

LOCK BASED Protocol.

SHARED LOCK [S] $\Rightarrow$ Only Read

Exclusive lock [X] $\Rightarrow$ Write | write
Read

Same Data Item	T_i S	X
T_j S	YES	NO
X	No	No

Q. 1

Consider the following database schedule with two transactions, T_1 and T_2.

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

where $r_i(Z)$ denotes a read operation by transaction T_i on a variable Z, $w_i(Z)$ denotes a write operation by T_i on a variable Z and a_i denotes an abort by transaction T_i

Which one of the following statements about the above schedule is TRUE?

[MCQ:2016–2M]

A) S is non-recoverable

B) S is recoverable, but has a cascading abort

C) S does not have a cascading abort

D) S is strict

Q.2 Let S be the following schedule of operations of three transactions T_1, T_2 and T_3 in a relational database system:

$$R_2(Y), R_1(X), R_3(Z), R_1(Y), W_1(X), R_2(Z), W_2(Y), R_3(X), W_3(Z)$$

Consider the statements P and Q below:

P: S is conflict-serializable.

Q: If T_3 commits before T_1 finishes, then S is recoverable.

Which one of the following choices is correct?

[MCQ: 2021-2M]

(A) Both P and Q are true.

(B) P is true and Q is false.

(C) P is false and Q is true.

(D) Both P and Q are false.

Q.3

Consider a simple checkpointing protocol and the following set of operations in the log.

(start, T4); (write, T4, y, 2, 3); (start, Tl);

(commit, T4); (write, T1, z, 5, 7);

(checkpoint);

(start, T2); (write, T2, x, 1, 9); (commit, T2);

(start, T3); (write, T3, z, 7, 2);

If a crash happens now and the system tries to recover using both undo and redo operations, what are the contents of the undo list and the redo list

[MCQ: 2015–2M]

(A) Undo: T3, T1; Redo: T2

(B) Undo: T3, T1; Redo: T2, T4

(C) Undo: none; Redo: T2, T4, T3, Tl

(D) Undo: T3, Tl, T4; Redo: T2

Lock-S(A)

Read(A)

Unlock-S(A)

Lock-X(A)

Write(A) or Write(A)
Read(A)

Unlock-X(A).

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes:

  1. exclusive (X) mode. Data item can be both reads as well as written. X-lock is requested using **lock-X** instruction.
  2. Shared (S) mode. Data item can only be read. S-lock is requested using **lock-S** instruction.

- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-Based Protocols (Cont.)

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be garneted a lock on an item if the requested lock is compatible with lock already held on the item by other transactions
- Any number of transactions can hold shared locks on an item
- But if any transaction holds an exclusive on the item no other transaction may hold any loc on the item.

Schedule with Lock Grants

- A locking protocol is a set of rules followed by all transactions while requesting and releasing locks.
- Locking protocols enforce serializability by restricting the set of possible schedules.

T_1	T_2	Concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B) B:=B – 50 write(B) unlock(B)		
	lock-S(A, T_2)	
		grant-S(A, T_2)
	read(A) unlock(A) lock-S(B)	
		grant-S(B, T_2)
	read(B) unlock(B) display(A+B)	
lock-X(A)		
		grant-X(A, T_1)
read(A) A:=A + 50 write(A) unlock(A)		

2 Phase locking Protocol [2PL]

1. Growing Phase [Acquire lock]

2. Shrinking Phase [Release lock]

Here we Issue Lock & Unlock Request into the Two Phases.

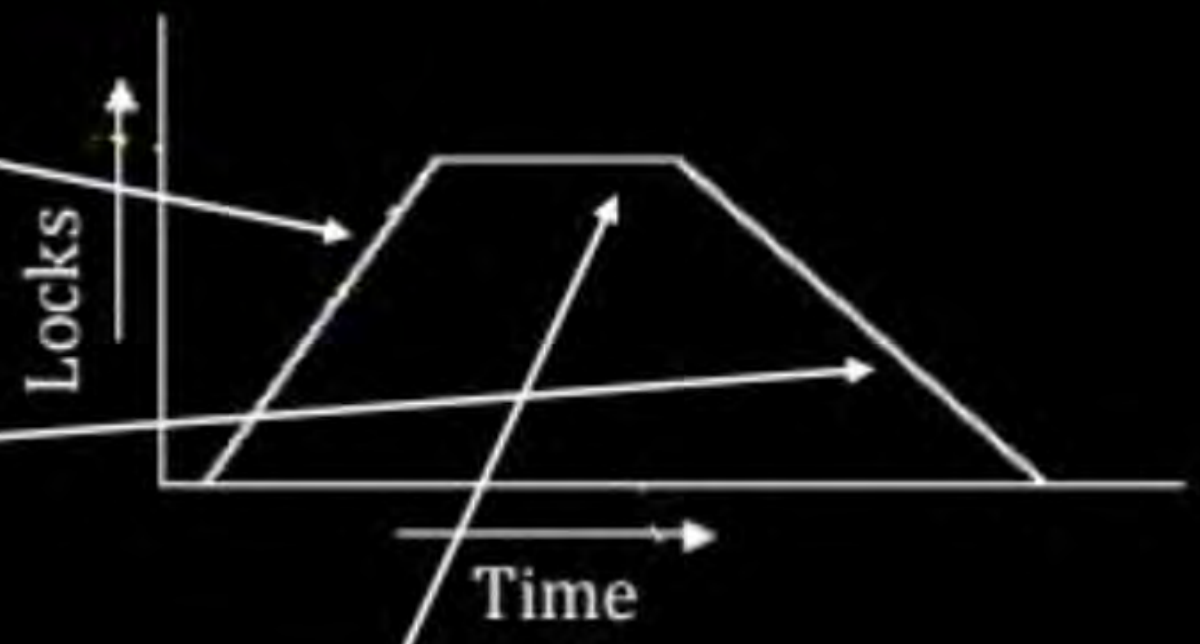
① **Growing Phase** : In Growing Phase Transaction May obtain the Lock But Must Not Release Any Lock.

② **Shrinking Phase** : In Shrinking Phase Transaction Release the lock But Must Not Ask for Any New lock.

Each Transaction finish first its Growing Phase then Shrinking Phase

The Two-Phase Locking Protocol

- A protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - Transaction may obtain locks
  - Transaction may not release lock
- Phase 2: Shrinking Phase
  - Transaction may release locks
  - Transaction may not obtain locks



- The protocol assures serializability. It can be proved that transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock).

Serializablity order (Equivalent Serial schedule) is Determined by Lock POINT.

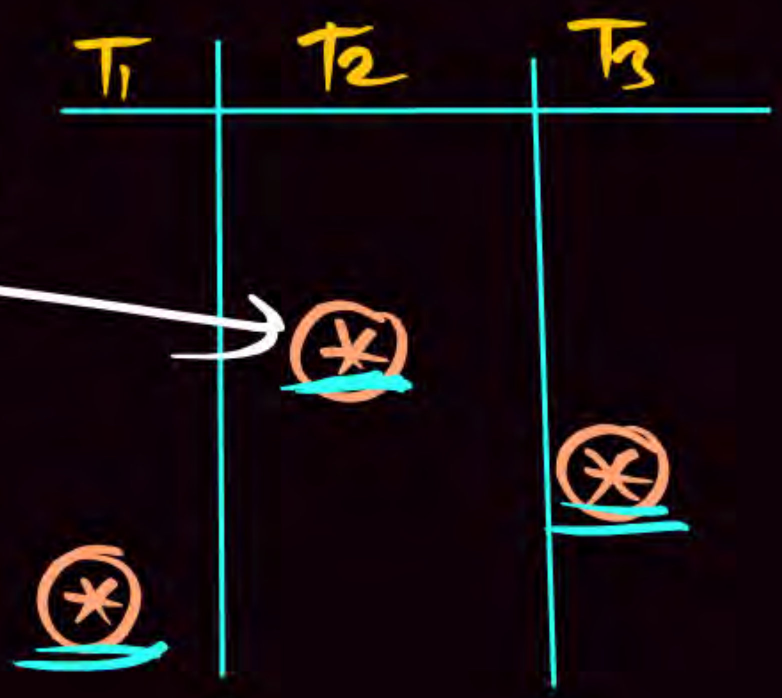
Lock POINT

→ Position of last Lock or first Unlock operation.

or

is a point FROM WHERE Shrinking Phase of the transaction Starts.

position of LAST Lock or First Unlock.



Serializablity order : $\langle T_2 \ T_3 \ T_1 \rangle$

2PL

	T1	T2
1	Read(A)	
	A = A-100	
	Write(A)	
		Read(A)
		Read(C)
	Read(B)	
	B = B+100	
	Write(B)	

	T1	T2	Concurrency control manager
	Lock-X(A)		Grant-X(A, T1)
	Read(A)		
	A = A-100		
	Write(A)		
		Lock-S(A)	X Not allowed
	Lock-X(B)		Grant-X(B, T1)
LP	Unlock-X(A)		Revoke-X(A, T1)
		Lock-S(A)	Grant-S(A, T2)
		Read(A)	
		Lock-S(C)	Grant-S(B, T2)
Lock Point		Unlock-S(A)	
	Read(B)	Read(C)	
	B = B+100	Unlock-S(B)	
	Write(B)		
	Unlock-X(B)		

T1 → T2

2PL

(2)

T1	T2
Read(A)	
A=A-100	
Write(A)	
	Read(A)
	Commit
Commit	

Irrecoverable / Non recoverable Schedule.

T1	T2
Lock-X(A)	
Read(A)	
A=A-100	
write(A)	
unlock-X(A)	
	LOCK-S(A)
	Read(A)
	unlock-S(A)
	Commit
Commit	

Irrecoverable schedule But followed by 2PL.

3

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

T_1	T_2
S(A)	
R(A)	
	X(B)
	W(B)
R(B)	
	X(A)

Denied by T_2

Denied by T_1

Deadlock

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

T_1	T_2
X(A)	
W(A)	
	S(B)
	R(B)
X(B)	
	S(A)

Deny by T_2

Deny by T_1

Deadlock

T_1	T_2	T_3	T_4	T_5
	S(A)			
Denied by T_2 → X(A)				
		S(A)		
	Unlock-S(A)			
Denied by T_3 → X(A)				
			S(A)	
		Unlock-S(A)		
Denied by T_4 → X(A)				
				S(A)
			Unlock-S(A)	
Denied by T_5 → X(A)				
				Unlock-S(A)
Granted → X(A)				

STARVATION.

Important Points about 2PL.

① 2PL Ensure Conflict Serializable Schedule.

↳ Serializablity

OR

If a Schedule is Followed by 2PL then It Ensure Conflict Serializablity

② Serializablity order Determined by **LOCK POINT.**

③ 2PL Not Ensure Recoverablity.

④ 2PL Not Free From Deadlock | Suffer from Deadlock.

⑤ 2PL Suffer from Starvation | Not Free From Starvation.

2 PL

2PL if 2PL Applied

Serializablity ✓ Recoverability ✗

→ Ensure conflict serializable schedule

→ Serializability

But → Not Ensure **Recoverability**

→ Suffer from **Deadlock**

Suffer from **Starvation**

<u>2 PL</u>

Serializable

⇓

Conflict Serializable

But Recoverable Not Ensure.

② STRICT 2PL :

2PL + All Exclusive Lock Taken by the Transaction untill Commit / Rollback.

OR

2PL + All Exclusive Lock Release After Commit / Rollback of the transaction.

T_1	T_2
X(A)	
W(A)	
C/R	
Unlock-X(A)	
	S(A) / X(A)
	R(A) / W(A)

Strict Recoverable

ⓔ

$\overline{T}$

S(A)
X(B)
X(C)
S(D)
⋮
Unlock (A)
Unlock (D)
Commit
Unlock-X(B)
Unlock-X(C)

①

T_1	T_2
W(A)	
	R(A)
C/R	
	Commit

Recoverable

②

T_1	T_2
W(A)	
C/R	
	R(A)

Cascadeless

T_1	T_2
W(A)	
C/R	
	R(A) / W(A)

Strict Recoverable.

STRICT 2PL ⇒ 2PL +

→ Conflict serializable.

↳ Ensure

→ Recoverable schedule

→ Cascadeless

→ Strict Recoverable

→ No Cascading Rollback.

But suffer / Not Free From

↳ Deadlock

↳ Starvation.

③ Rigorious 2PL.

2PL + All Locks (SHARED [S] & Exclusive [X]) Held untill Commit | Rollback of the transaction OR Release After Commit | Rollback.

T_i
S(A)
X(B)
X(C)
S(D)
C / R
U(A)
U(B)
U(C)
U(D)

Rigorious 2PL are → 2PL
→ Strict 2PL.

But suffer from
- Deadlock
- Starvation.

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking does not ensure freedom from deadlocks
- Extensions to basic two-phase locking needed to ensure recoverability of freedom from cascading roll-back
  - (1) Strict two-phase locking: a transaction must hold all its exclusive locks till it commits/aborts.
    - Cascadeless. Strict Recoverable.
    - Ensures recoverability and avoids casecading roll-backs
  - (2) Rigorous two-phase locking: a transaction must hold all locks till commit/abort. SHARED & Exclusive
    - Transactions can be serialized in the order in which they commit.
- Most databases implement rigorous two-phase locking, but refer to if as simply two-phase locking

④ Conservative 2PL : Which State Acquire(taken) all the Locks before transaction Starts (at the beginning) & Release All the Lock After Commit/Rollback.

T_1	T_2
X(A)	
W(A)	
	S(B)
	R(B)
X(B) Not allowed	
W(B)	
	S(A) Not allowed
	R(A)

Deadlock.

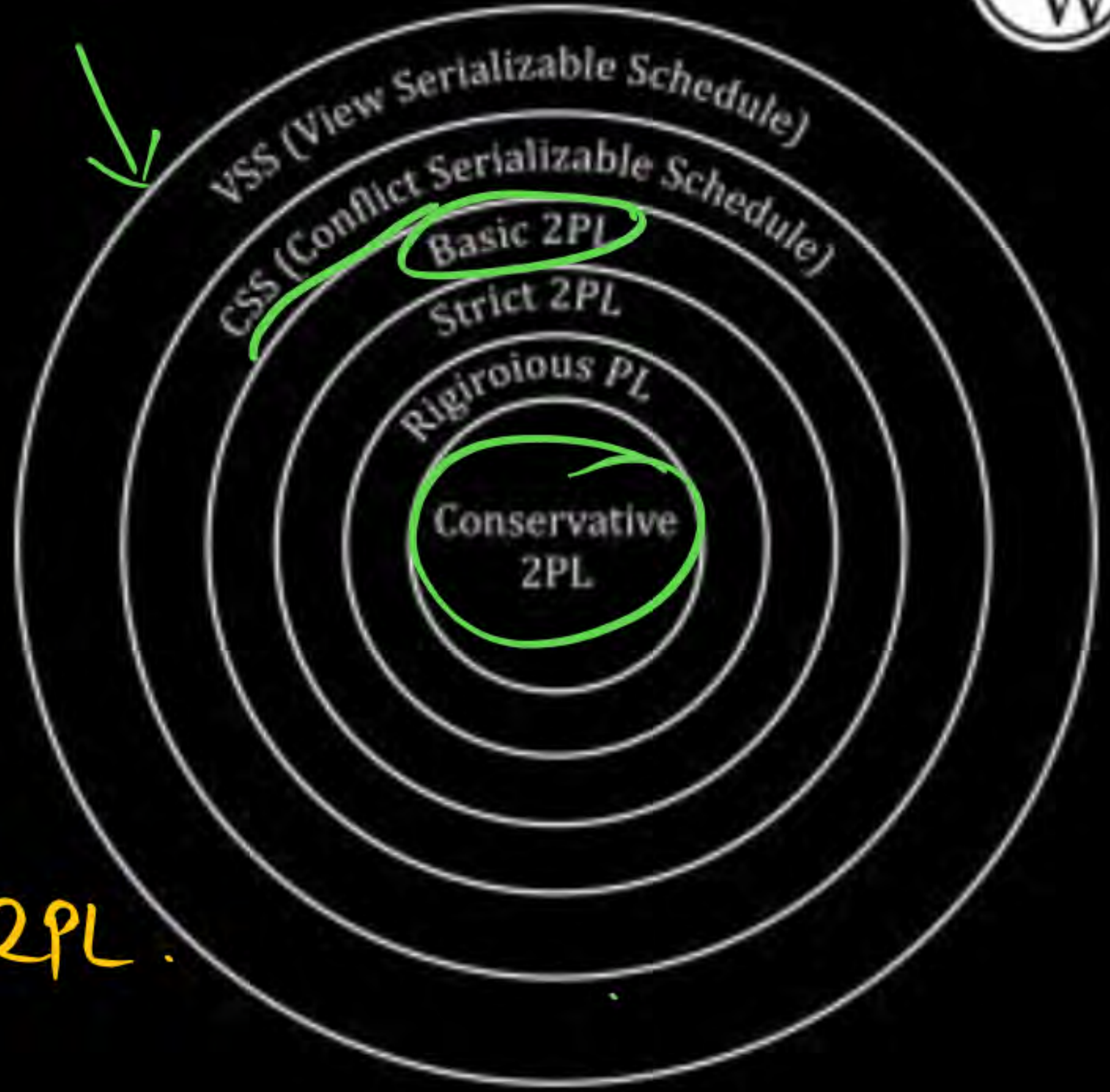
Conservative → 2PL
→ Strict 2PL

No Deadlock

Only suffer from starvation.

1	2	3	4	5
Lock-S(A)	Lock-S(A)	Lock-S(A)	Lock-S(A)	Lock-S(A)
R(A)	R(A)	R(A)	Lock-X(B)	R(A)
Lock-X(B)	Lock-X(B)	Lock-X(B)	R(A)	Unlock(A)
R(B)	Unlock(A)	R(B)	R(B)	Lock-X(B)
Unlock(A)	R(B)	W(B)	W(B)	R(B)
W(B)	W(B)	Commit	Commit	W(B)
Unlock(B)	Commit	Unlock(A)	Unlock(A)	Unlock(B)
Commit	Unlock(B)	Unlock(B)	Unlock(B)	Commit
2PL Only	· Strict 2PL · 2PL	Rigorious 2PL Strict 2PL 2PL.	Conservative 2PL Rigorious 2PL Strict 2PL 2PL.	Not 2PL.

T_i	T_j
R(A)	
	W(P)
R(B) W(B)	



TIMESTAMP BASED CONCURRENCY CONTROL

Timestamp Based Protocol :

Unique Time Stamp value is assigned to each Transaction, when Transaction Enter into the System

$TS(T_1) = 10$

$TS(T_2) = 30$

$TS(T_3) = 20$

$TS(T_4) = 40$

$\underset{T_1}{10} \rightarrow \underset{T_3}{(20)} \rightarrow \underset{T_2}{(30)} \rightarrow \underset{T_4}{(40)}$

OLDER Transaction

Serializability order $= \langle T_1\ T_3\ T_2\ T_4 \rangle$

Younger Transaction.

Timestamp-Based Protocols

- Each transaction T_i is issued a timestamp $TS(T_i)$ when it enters the system.
  - Each transaction has a unique timestamp
  - Newer transaction have timestamp strictly greater than earlier ones
  - Timestamp could be based on a logical counter
    - Real time may not be unique
    - Can use (wall-clock time, logical counter) to ensure
- Timestamp-based protocols manage concurrent execution such that **time-stamp order = serializability order**
- Several alternative protocols based on timestamps

If Transaction T_j enter, After the transaction T_i

then

Time Stamp of T_j

$TS(T_j) > TS(T_i)$

$$TS(T_j) > TS(T_i)$$

$$T_i \longrightarrow T_j$$

T_i: OLDER Transaction

T_j: Younger (Newest) Transaction

Serializability order $\langle T_i \ T_j)$

T_i followed by T_j

There are 2 Type of Time Stamp.

1. ① Transaction Time Stamp $TS(T_i)$: Fixed

2. ② **Data Item Time stamp:**
   - $RTS(A)$
   - $WTS(A)$

   {variable}

RTS(A): Read Time Stamp on Data Item A

WTS(A): Write Time stamp on Data Item A.

① Read-TimeStamp [RTS(A)] : Denote the Highest Transaction Timestamp which perform R(A) operation successfully.

T_1 (10)	T_2 (20)	T_3 (30)
R(A)		
	R(A)	
		R(A)

Initially RTS(A) = 0;

RTS(A) = 10; max(0, 10)

RTS(A) = 20; max(10, 20)

RTS(A) = 30; max(20, 30)

$\boxed{\text{RTS(A) : 30}}$

② Write Time Stamp (WTS(A)) : Denote the Highest Transaction Time stamp that Perform write(A) operation successfully.

10 T_1	20 T_2	30 T_3
W(A)		
	W(A)	
		W(A)

$$\boxed{WTS(A) = 30}$$

WTS(A) = 0 (initially)

WTS(A) = 10 max(0, 10)

WTS(A) = 20 max(10, 20)

WTS(A) = 30 max(20, 30)

Timestamp-Based Protocols

The **timestamp ordering (TSO) protocol**

- Maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed write (Q) successfully.
  - **R-timestamp** (Q) is the largest time-stamp of any transaction that executed **read** (Q) successfully.
- Imposes rules on read and write operations to ensure that
  - any conflicting operations are executed in timestamp order
  - out of order operations cause transaction rollback

Conflict operation

T_i

Read (A) ⟶ Write (A)

T_i

Write (A) ⟶ Read (A)

Write (A) ⟶ Write (A)

$T_i \Rightarrow$ Read(Q)

$\boxed{\text{If } TS(T_i) < WTS(Q);}$ Reject

T_i: Write(Q)

$TS(T_i) < RTS(Q)$; Reject

$TS(T_i) < WTS(Q)$; Reject

I: T_i – Read(Q) (Transaction T_i Issue R(Q) Operation)

(i) If $TS(T_i) < WTS(Q)$: Read operation Reject & T_i Rollback.

(ii) If $TS(T_i) \geq WTS(Q)$: Read operation is allowed

and Set Read – $TS(Q) = \max[RTS(Q), TS(T_i)]$

II: T_i – Write(Q) (Transaction T_i Issue Write(Q) Operation)

(i) If $TS(T_i) < RTS(Q)$: Write operation Reject & T_i Rollback.

(ii) If $TS(T_i) < WTS(Q)$: Write operation Reject & T_i Rollback.

(iii) Otherwise execute write (Q) operation

Set Read $WTS(Q) = TS(T_i)$

Read

(i) If $TS(T_i) < WTS(Q)$

2010 | 2022 | 2023 | 2024

(10) T_1 | (20) T_2 | (30) T_3 | (40) T_4

$W(Q)$

$T_i \equiv T_2$

$\Downarrow$

$TS(T_2) = 20$

$$WTS(Q) = 30$$

$$TS(T_i) = TS(T_2) = 20$$

$T_i = T_2 \Rightarrow TS(T_2) = 2021$

Write(Q)

$TS(T_i) < RTS(Q)$

year.

2020	2021	2023
T_1	T_2	T_3
	W(Q)	
		R(Q)

$RTS(Q) = 2023.$

$TS(T_2) = 2021$

$TS(T_i) < WTS(Q)$

2020	2021	2023
T_1	T_2	T_3
	W(Q)	
		W(Q)

$WTS(Q) = 2023$

$TS(T_2) = 2021$

Timestamp-Based Protocols (Cont.)

- Suppose a transaction T_i issues a **read** (Q)

1. If $TS(T_i) \leq$ **W**-timestamp (Q), then T_i needs to read a value of Q that was already overwritten.
   - Hence, the **read** operation is rejected, and T_i is rolled back.
2. If $TS(T_i) \geq$ **W**-timestamp (Q), then the **read** operation is executed, and R-timestamp(Q) is set to.

   max(R-timestamp (Q), TS (T_i)).

Timestamp-Based Protocols (Cont.)

- Suppose a transaction T_i issues **write**(Q)

1. If $TS(T_i) < \mathbf{R}$-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that the value would never be produced.
   - Hence, the **write** operation is rejected, and T_i is rolled back.
2. If $TS(T_i) < \mathbf{W}$-timestamp (Q), then T_i is attempting to write an obsolete value of Q.
   - Hence, this **write** operation is rejected, and T_i is rolled back.
3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to $TS(T_i)$.

(1)

$T_1(10)$	$T_2(20)$
R(A)	
	W(A)
W(A)	

$TS(T_1) < TS(T_2)$

$T_1 \rightarrow T_2$

(2)

$T_1(10)$	$T_2(20)$
R(A)	
	R(A)
W(A)	

$TS(T_1) < TS(T_2)$

$T_1 \rightarrow T_2$

(3)

$T_1(10)$	$T_2(20)$
R(A)	
	W(A)
R(B)	
	W(B)

$TS(T_1) < TS(T_2)$

$T_1 \rightarrow T_2$

(4)

$T_1(10)$	$T_2(20)$
	r(A)
r(B)	
W(A)	

$TS(T_1) < TS(T_2)$

$T_1 \rightarrow T_2$

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if $TS(T_i) < W\text{-timestamp}(Q)$, then T_i is attempting to write an obsolete value of {Q}.
  - Rather than rolling back T_i as the timestamp ordering protocol would have don, this {**write**} operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.
  - Allows some view-serializable schedules that are not conflict-serializable.

Thomas Write Rule (View Serializability)

1. $TS(T_i) < RTS(Q)$: Rollback
2. $TS(T_i) < WTS(Q)$: Write operation is Ignored and No Roll back

Same as TSP

Time Stamp Protocol: Ensure serializability deadlock free but starvation possible

Deadlock Prevention Algorithm

(1) Wait-Die (2) Wound-wait

Older Younger

$T_1(10)$	$T_2(20)$
R(A)	
	W(A)
W(A)	

$TS(T_1) < TS(T_2)$

$T_1 \rightarrow T_2$

T_1	T_2
R(A)	
W(A)	
	W(A)

$T_1 \rightarrow T_2$

THANK YOU!