Lecture 6

Concurrency Control

- Timestamp-Based Protocols
- Validation-Based Protocols
- Deadlock Handling
- Insert and Delete Operations

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the 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.

Timestamp-Based Protocols (Cont.)

The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.

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

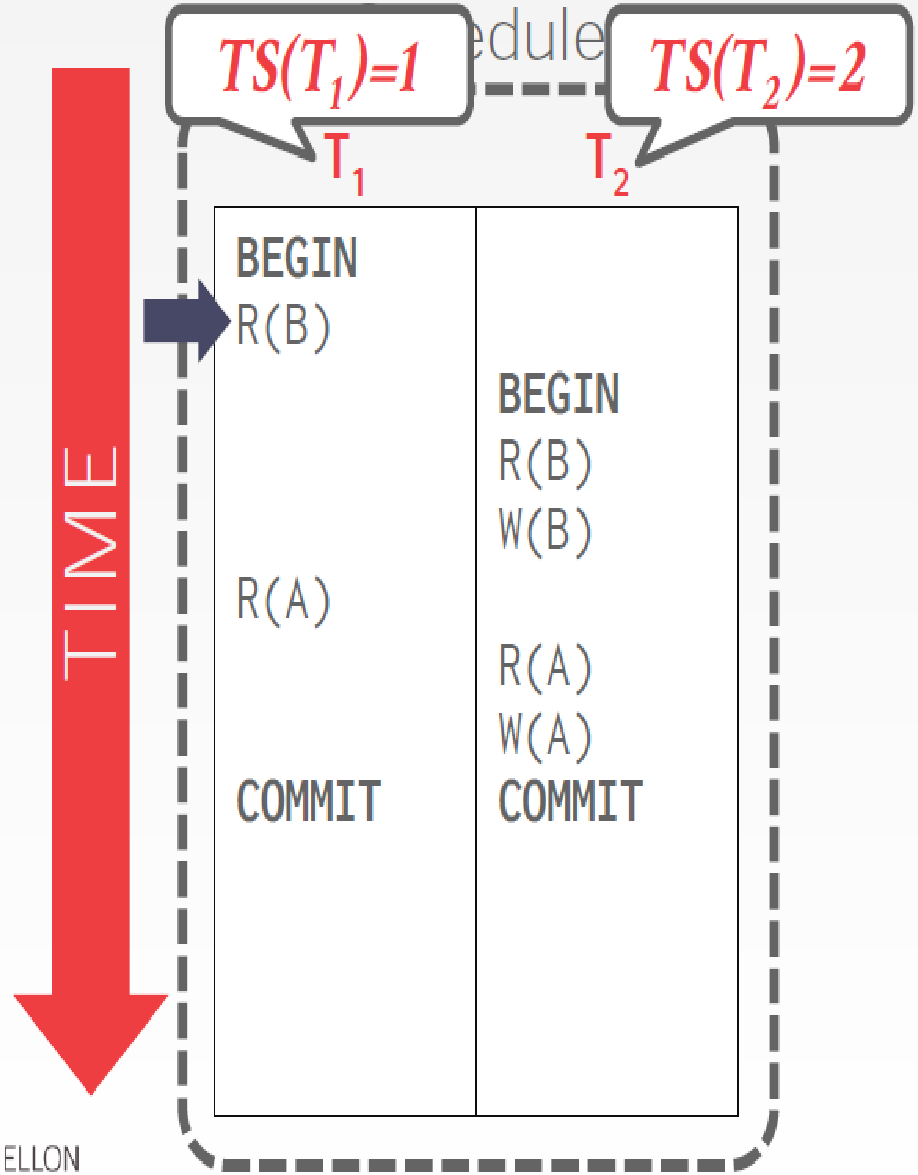
1. If TS(T_i) < **W**-TS(Q),

   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**-TS(Q), then the **read** operation is

   executed, and R-TS(Q) set to the maximum (R-TS(Q) , TS(T_i)).

Example 1



Database

Object	R-TS	W-TS
A	0	0
B	0	0

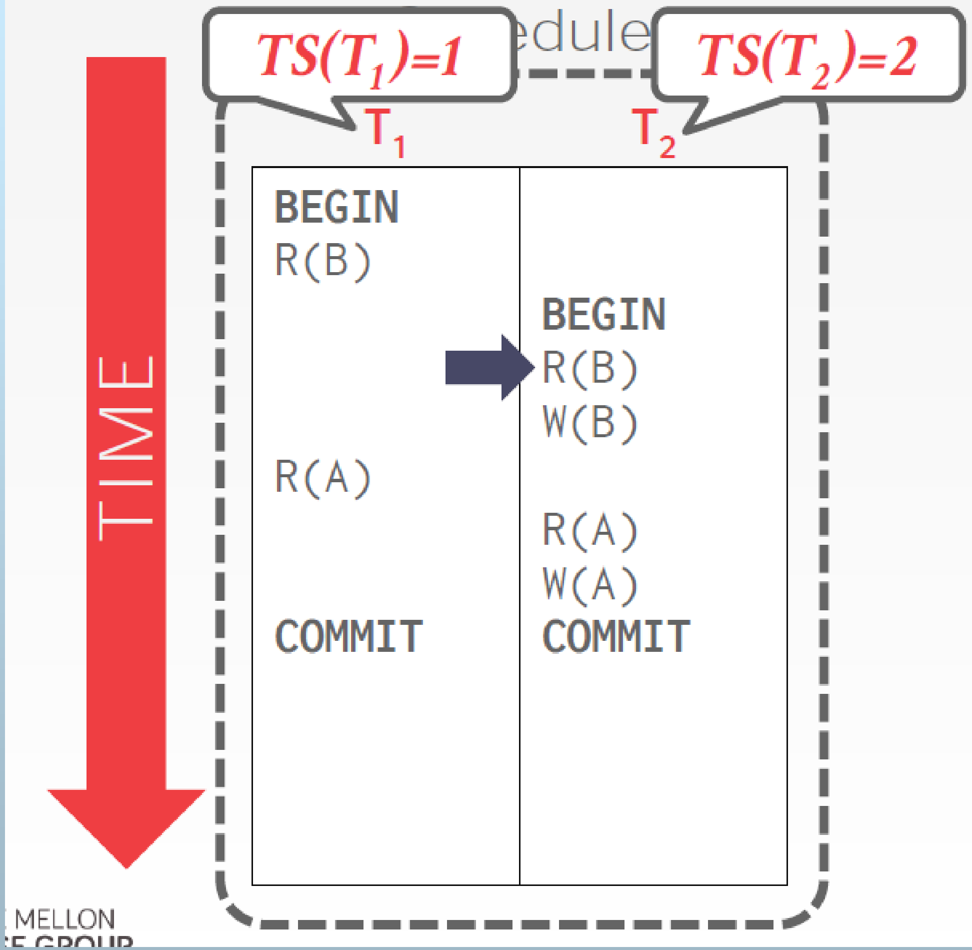
$TS(T_1)=1$ Schedule $TS(T_2)=2$

TIME

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

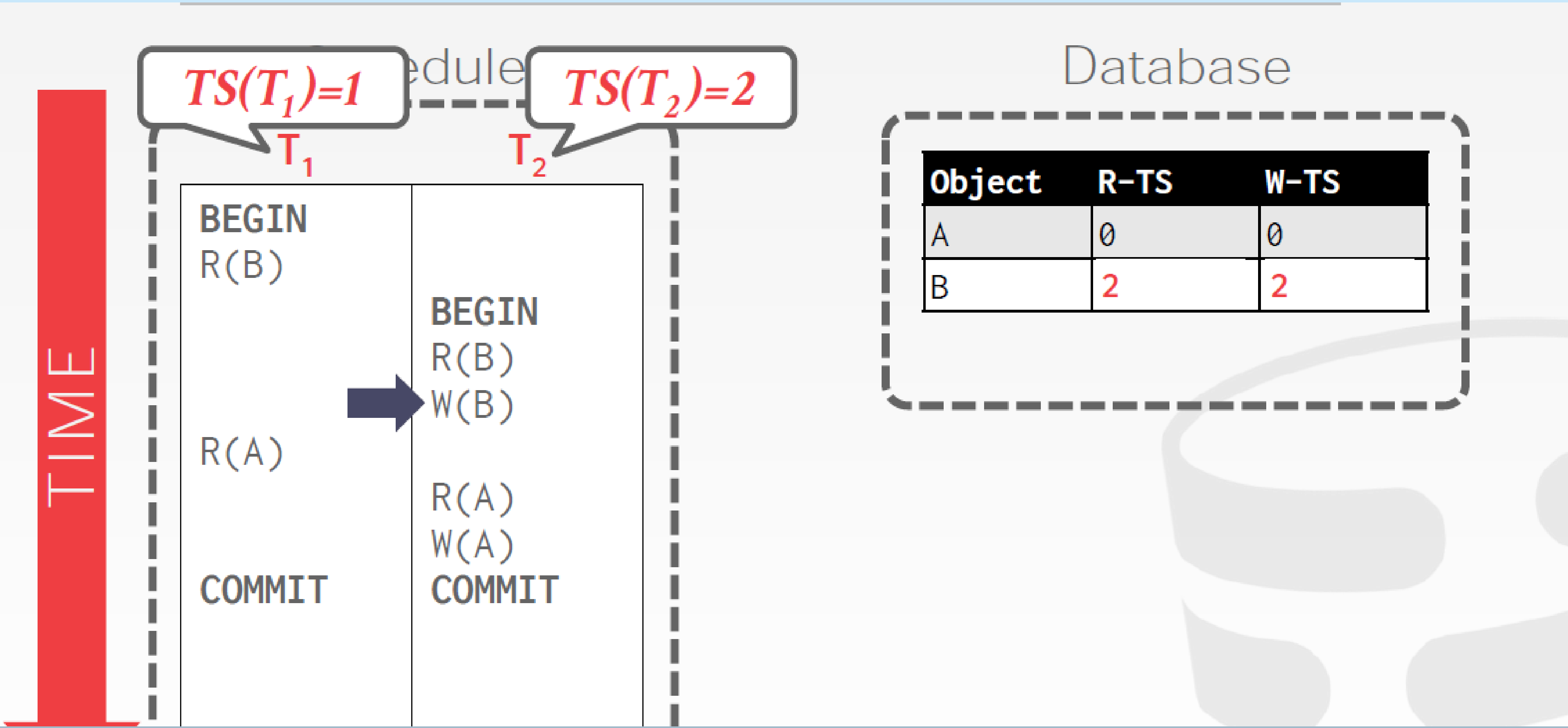
Database

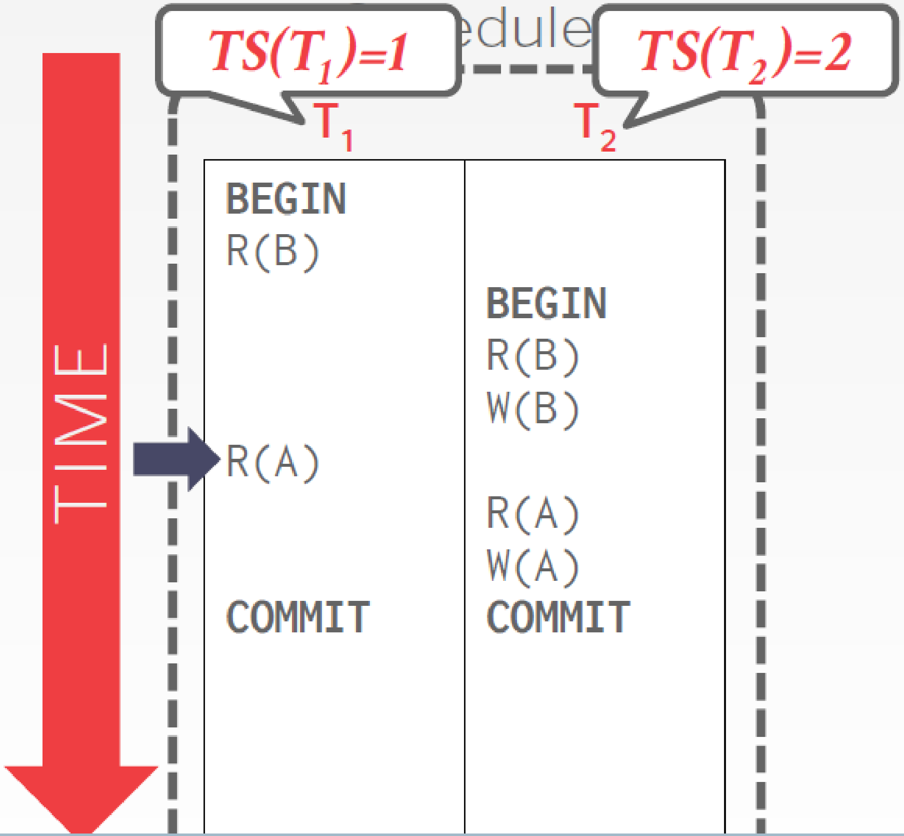
Object	R-TS	W-TS
A	0	0
B	1	0



Database

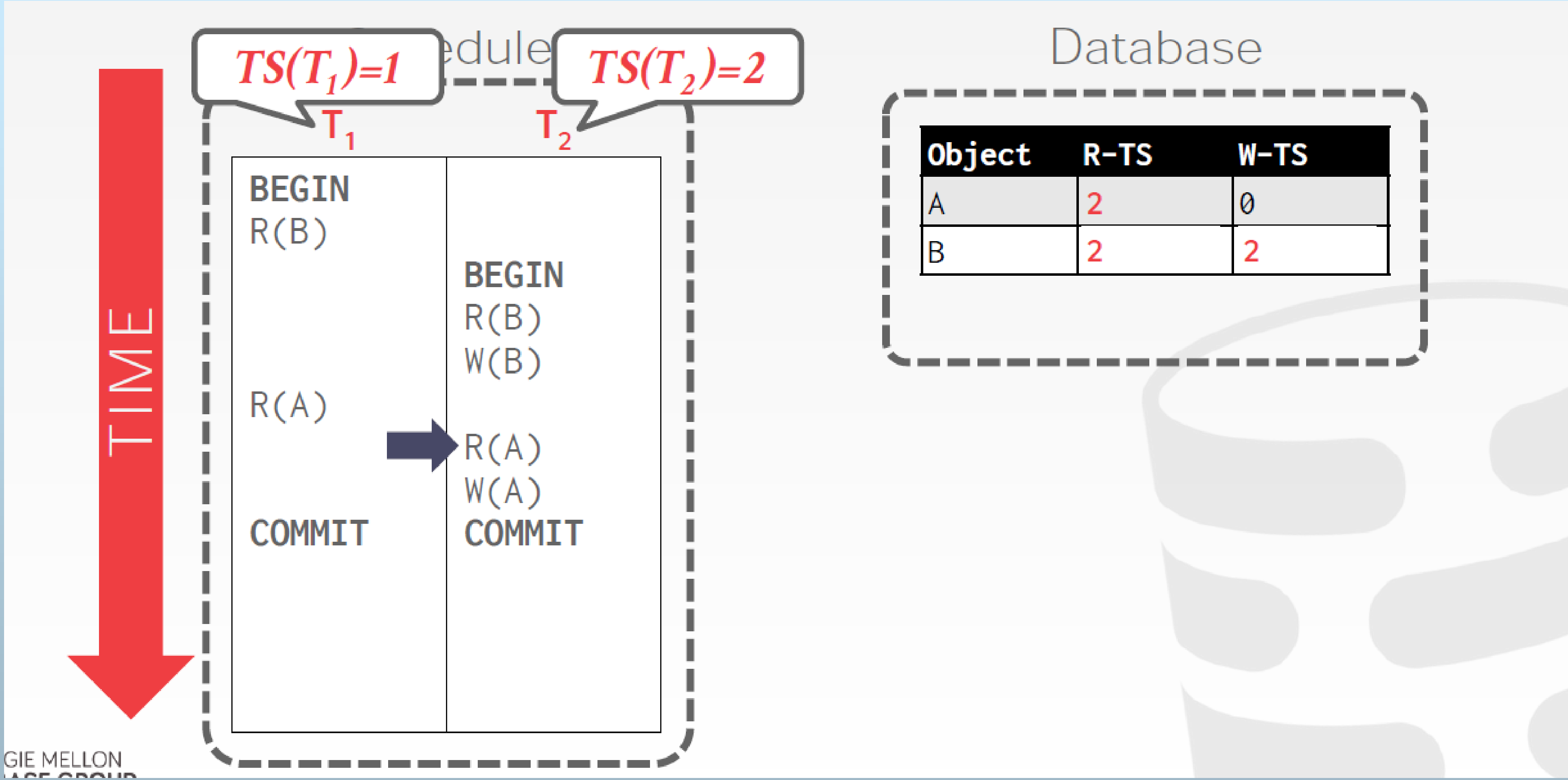
Object	R-TS	W-TS
A	0	0
B	2	0

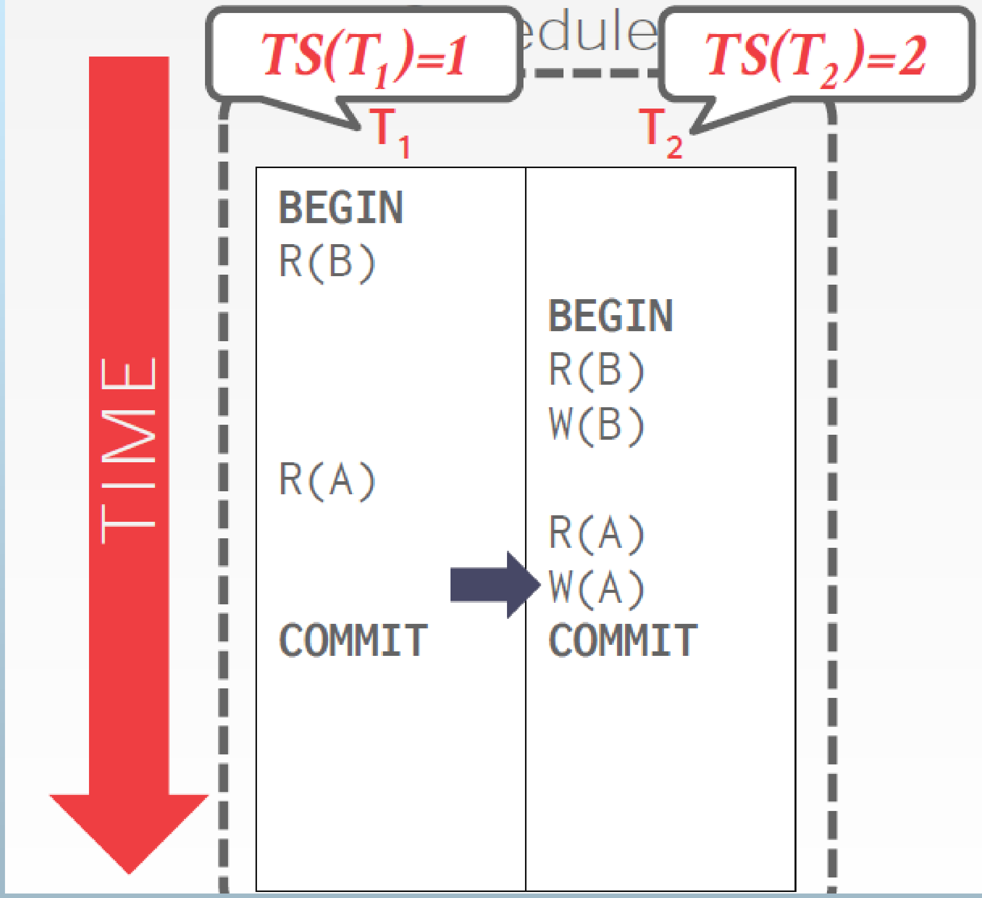

TS(T1)=1
TS(T2)=2
Database
TIME
T1
T2
BEGIN
R(B)
BEGIN
R(B)
W(B)
R(A)
R(A)
W(A)
COMMIT
COMMIT
Object R-TS W-TS
A 0 0
B 2 2



Database

Object	R-TS	W-TS
A	1	0
B	2	2


Database
TS(T1)=1
TS(T2)=2
T1
T2
BEGIN
R(B)
BEGIN
R(B)
W(B)
R(A)
R(A)
W(A)
COMMIT
COMMIT
TIME
Object | R-TS | W-TS
A | 2 | 0
B | 2 | 2
GIE MELLON



Database

Object	R-TS	W-TS
A	2	2
B	2	2

No violations so both txns are safe to commit.

Exercise

T1	T2
R(B)	
	R(B)
	W(B)
R(B)	

object	R-TS	W-TS
B	1	0
B	2	2

TS(T!) =1

R-TS(B)=2

Reject and T1 rollback

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write**(Q).

If $TS(T_i) < R\text{-}TS(Q)$ OR $TS(T_i) < W\text{-}TS(Q)$, then the **write** operation is rejected, and T_i is rolled back.

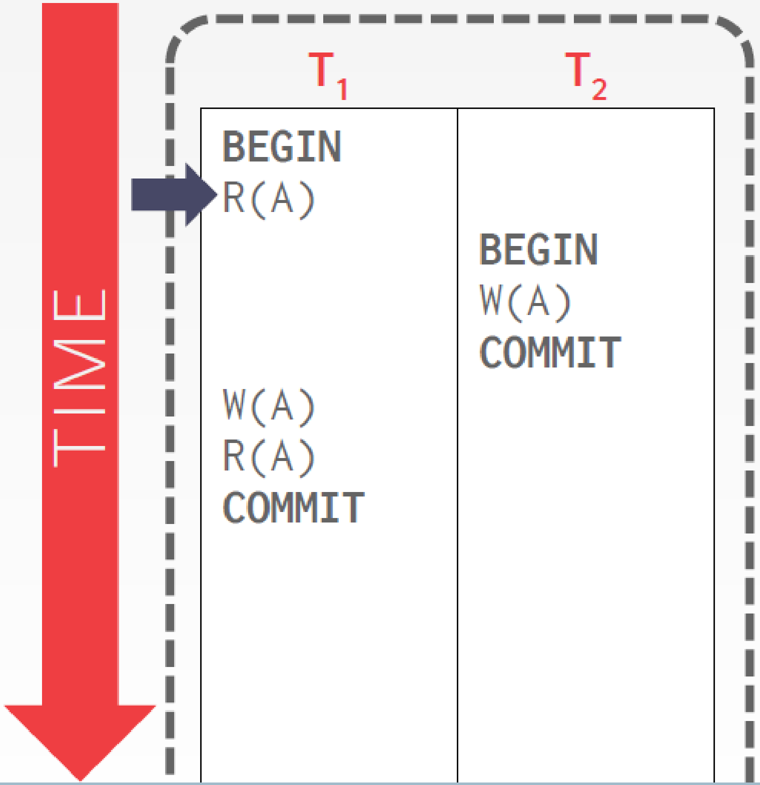
Else, the **write** operation is executed, and $W\text{-}TS(Q)$ is set to $TS(T_i)$.

Notes:

1. Schedules may be under the two-phase locking protocol, but possible are not under the timestamp protocol, and vice versa.
2. Schedules may be under the two protocols simultaneously.

Example 2

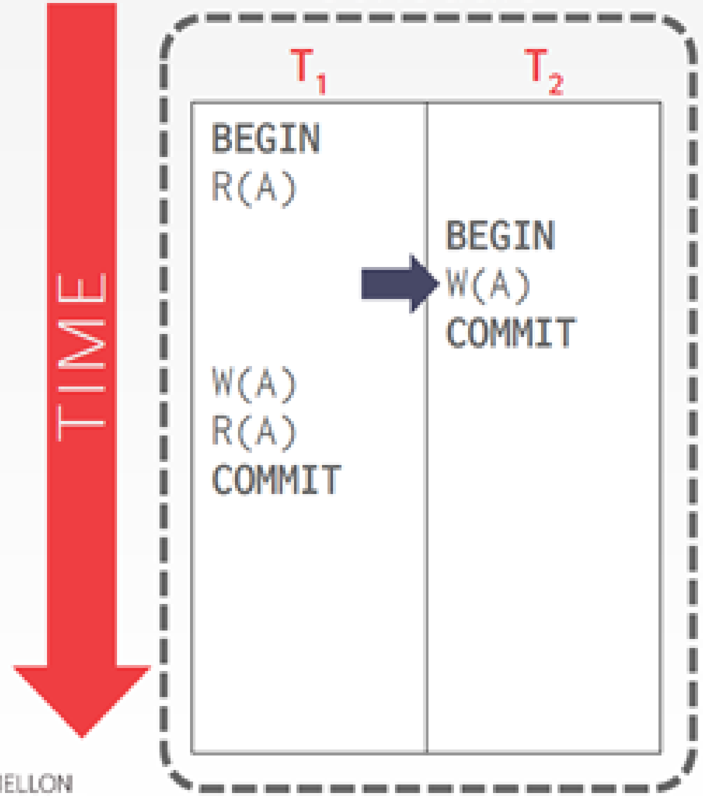
Schedule



Database

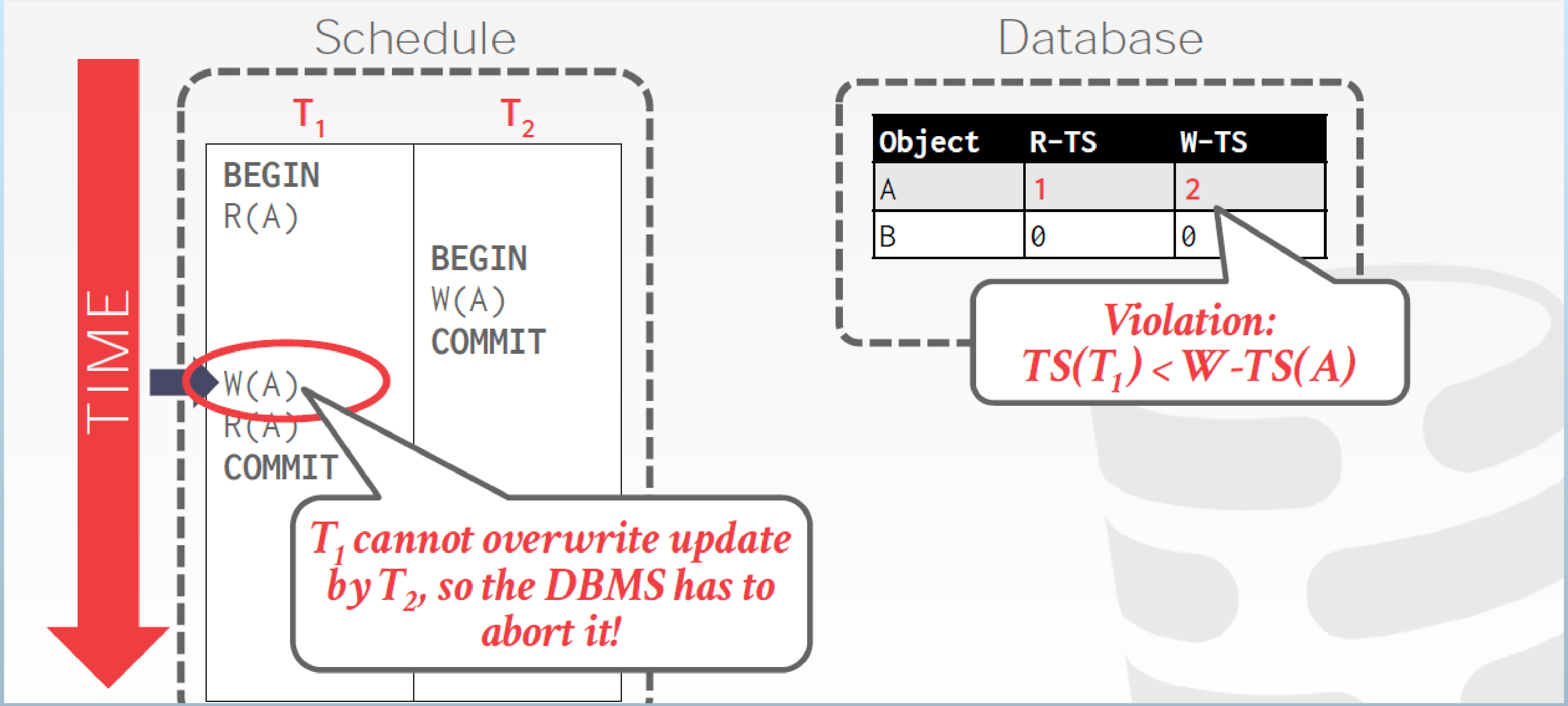
Object	R-TS	W-TS
A	1	0
B	0	0

BASIC T/O – EXAMPLE #2



Database

Object	R-TS	W-TS
A	1	2
B	0	0


Schedule
TIME
T1
T2
BEGIN
R(A)
BEGIN
W(A)
COMMIT
W(A)
R(A)
COMMIT
T1 cannot overwrite update by T2, so the DBMS has to abort it!
Database
Object
R-TS
W-TS
A
1
2
B
0
0
Violation:
TS(T1) < W-TS(A)

Advantages of Timestamp protocol

1. Ensures conflict serializability. This is because conflicting operations are processed in timestamp order.
2. Ensures freedom from deadlock.

There is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which **write** operations may be ignored.
- When T_i attempts to write data item Q, if TS(T_i) < W-TS(Q), then T_i is attempting to write an obsolete value of {Q}. Hence, rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be ignored.
- If TS(Ti) < W-TS(Q), then Ti is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency. Unlike previous protocols, it allows some view-serializable schedules that are not conflict-serializable.

OPTIMISTIC CONCURRENCY CONTROL
(Validation-Based Protocol)

The DBMS creates a private workspace for each transaction.

→ Any object read is copied into workspace.
→ Modifications are applied to workspace.

When transaction commits, the DBMS compares workspace write set to see whether it conflicts with other transactions.

If there are no conflicts, the write set is installed into the "global" database.

OCC PHASES

#1 – Read Phase:

→ Track the read/write sets of txns and store their writes in a private workspace.

#2 – Validation Phase:

→ When a txn commits, check whether it conflicts with other txns.

#3 – Write Phase:

→ If validation succeeds, apply private changes to database. Otherwise abort and restart the txn.

- Note: The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.

Validation-Based Protocol (OCC)

- Each transaction T_i has 3 timestamps
  - **Start**(T_i) : the time when T_i started its execution
  - **Validation**(T_i): the time when T_i entered its validation phase
  - **Finish**(T_i) : the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency. Thus $TS(T_i)$ is given the value of **Validation**(T_i).

OCC works best when the number of conflicts is low. This is when either all of the transactions are read-only or when transactions access disjoint subsets of data. If the database is large and the workload nearly balanced, then there is a low probability of conflict, making OCC a good choice.

Validation Test for Transaction T_j

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - **finish**(T_i) < **start**(T_j)
  - **start**(T_j) < **finish**(T_i) < **validation**(T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j.

  then validation succeeds and T_j can be committed. Otherwise, validation fails and T_j is aborted.
- *Justification*: Either first condition is satisfied, and there is no overlapped execution, or second condition is satisfied and

1. the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
2. the writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i.

Schedule Produced by Validation

- Example of schedule produced using validation

T_{14}	T_{15}
read(*B*)	
	read(*B*) *B*:- *B*-50 **read**(*A*) *A*:- *A*+50
read(*A*) (*validate*) **display** (*A*+*B*)	
	(*validate*) **write** (*B*) **write** (*A*)

$TS(T_{14}) < TS(T_{15})$

the writes to the actual variables are performed only after the validation phase of *T*15

Thus, *T*14 reads the old values of *B* and *A*, and this schedule is serializable.

This validation scheme is called the **optimistic concurrency control** scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end.

In contrast, locking and timestamp ordering are **pessimistic** in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serializable.