Concurrency Control and Recovery Techniques, NOSQL Management

Dr. Jyotismita Chaki

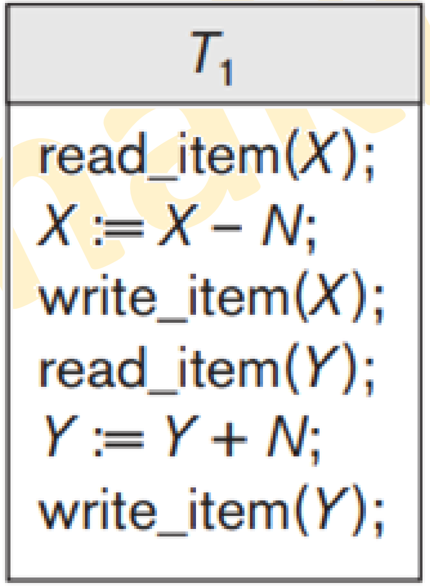
Why concurrency control

- Assume that two people who go to electronic kiosks at the same time to buy a movie ticket for the same movie and the same show time.
- However, there is only one seat left in for the movie show in that particular theatre.
- Without concurrency control in DBMS, it is possible that both moviegoers will end up purchasing a ticket.
- However, concurrency control method does not allow this to happen.
- Both moviegoers can still access information written in the movie seating database.
- But concurrency control only provides a ticket to the buyer who has completed the transaction process first.

Why concurrency control

- Several problems can occur when concurrent transactions execute in an uncontrolled manner.
- Referring to a much simplified airline reservations database in which a record is stored for each airline flight.
- Each record includes the number of reserved seats on that flight as a named (uniquely identifiable) data item, among other information.
- Figure (a) shows a transaction T1 that transfers N reservations from one flight whose number of reserved seats is stored in the database item named X to another flight whose number of reserved seats is stored in the database item named Y.
- Figure (b) shows a simpler transaction T2 that just reserves M seats on the first flight (X) referenced in transaction T1.

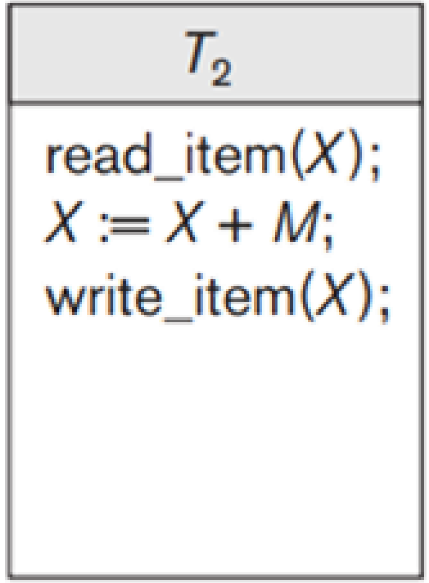
(a)



T_1

```
read_item(X);
X := X − N;
write_item(X);
read_item(Y);
Y := Y + N;
write_item(Y);
```

(b)



T_2

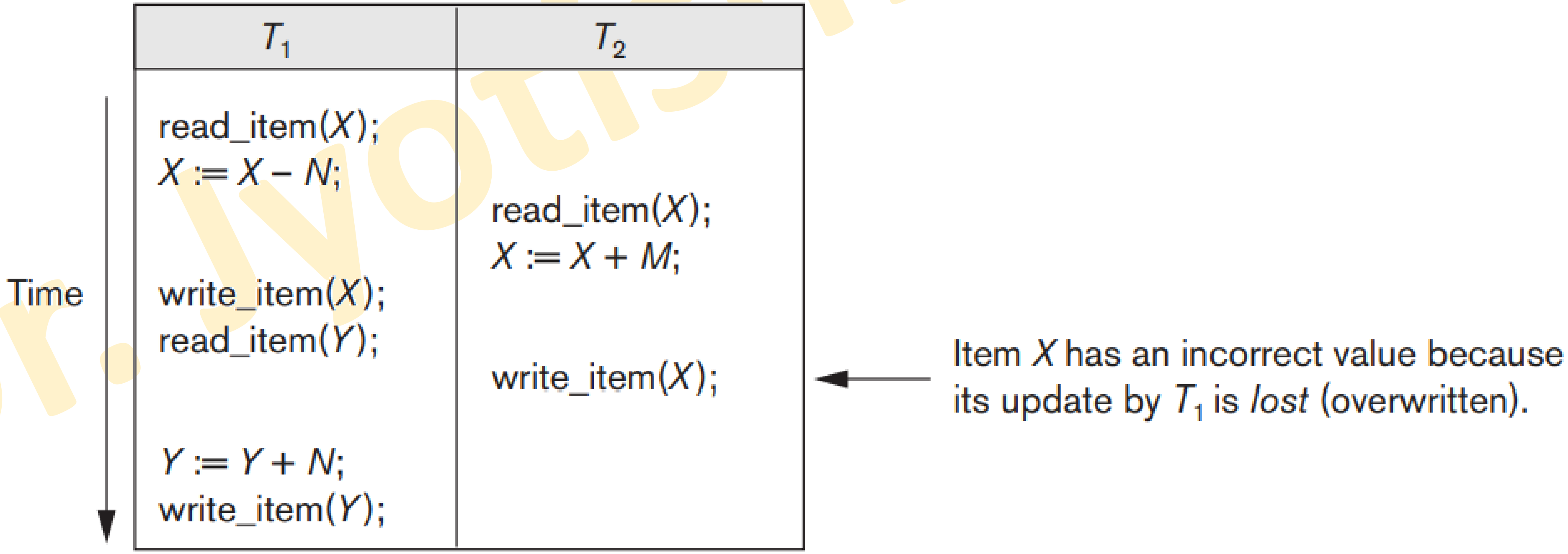
```
read_item(X);
X := X + M;
write_item(X);
```

Why concurrency control

- When a database access program is written, it has the flight number, the flight date, and the number of seats to be booked as parameters; hence, the same program can be used to execute many different transactions, each with a different flight number, date, and number of seats to be booked.
- For concurrency control purposes, a transaction is a particular execution of a program on a specific date, flight, and number of seats. In Figure (a) and (b), the transactions T1 and T2 are specific executions of the programs that refer to the specific flights whose numbers of seats are stored in data items X and Y in the database.

Why Concurrency control: Problems

- The types of problems we may encounter with these two simple transactions if they run concurrently.
    - **The Lost Update Problem**: This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect.



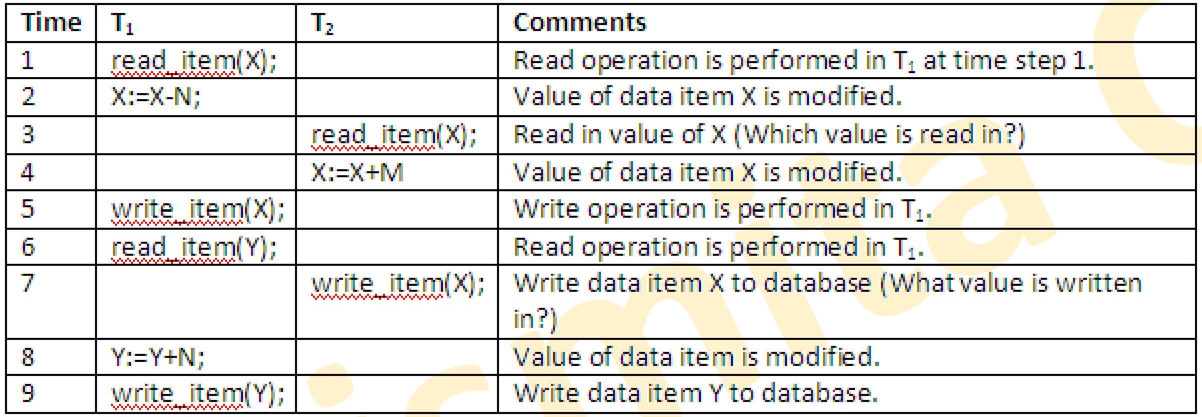
T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

write_item(X);
read_item(Y);	

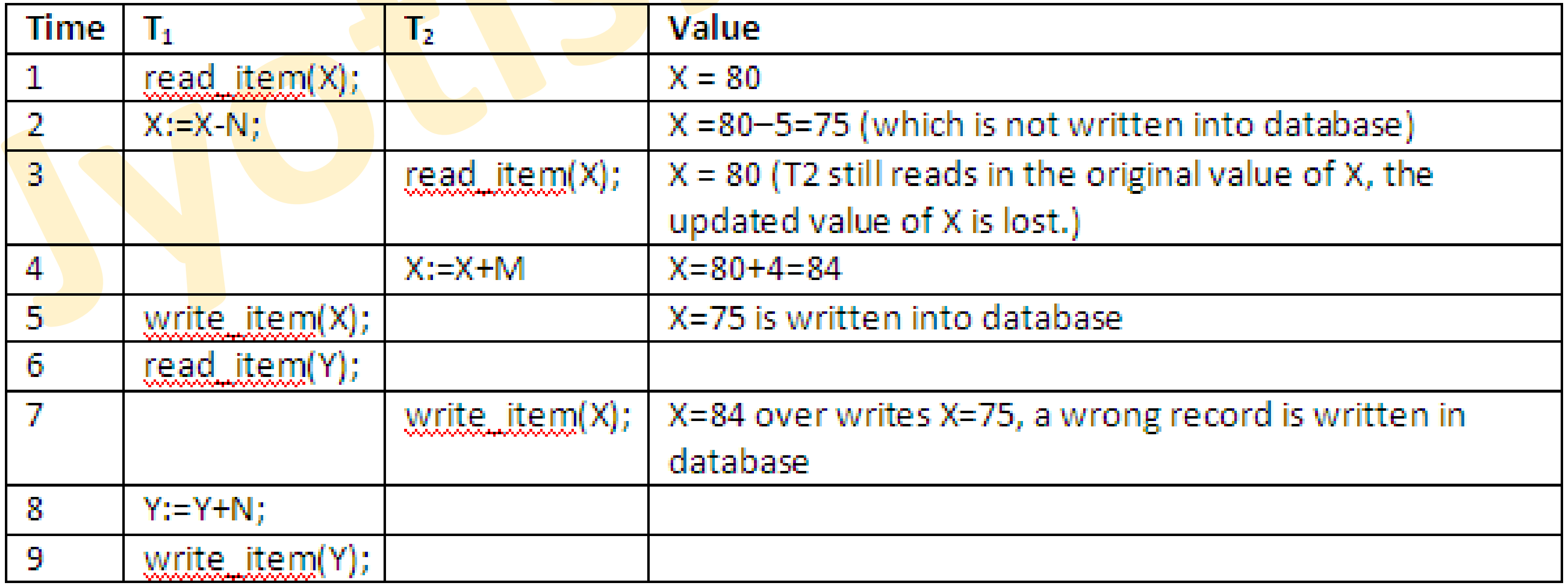
	write_item(X);
$Y := Y + N$; write_item(Y);	

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

Why Concurrency control: Problems: The Lost Update Problem Example



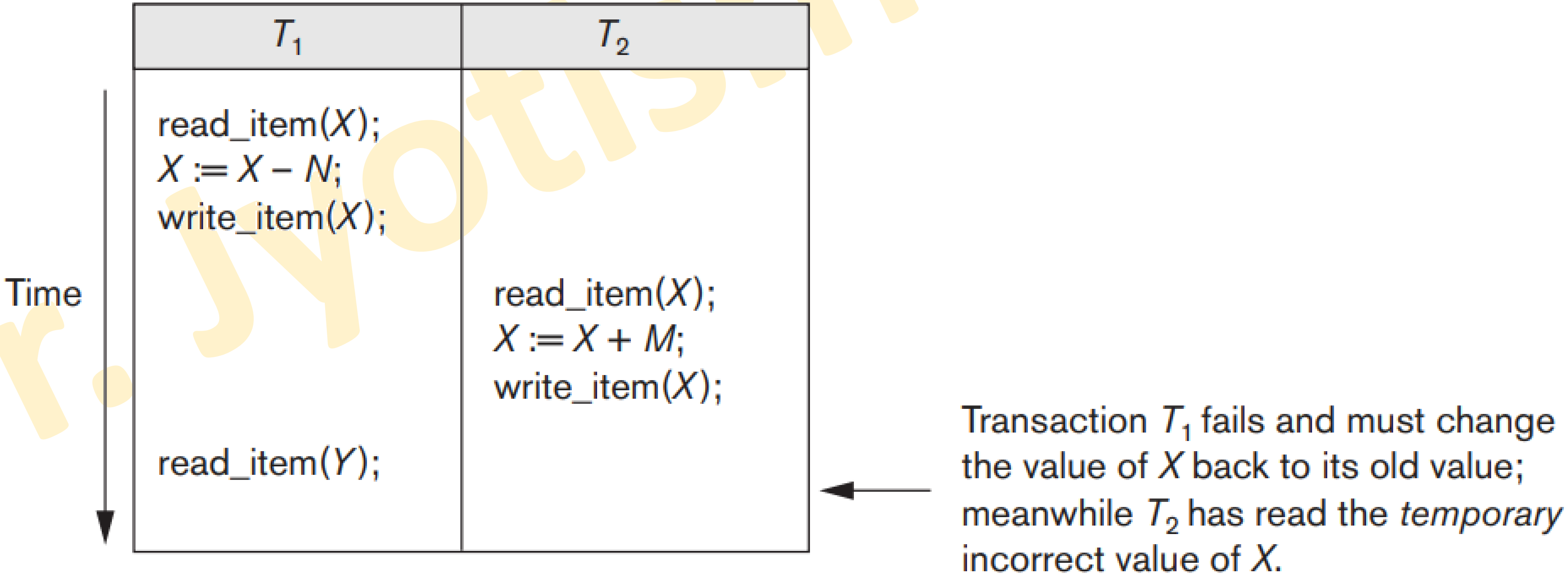
Time	T_1	T_2	Comments
1	read_item(X);		Read operation is performed in T_1 at time step 1.
2	X:=X-N;		Value of data item X is modified.
3		read_item(X);	Read in value of X (Which value is read in?)
4		X:=X+M	Value of data item X is modified.
5	write_item(X);		Write operation is performed in T_1.
6	read_item(Y);		Read operation is performed in T_1.
7		write_item(X);	Write data item X to database (What value is written in?)
8	Y:=Y+N;		Value of data item is modified.
9	write_item(Y);		Write data item Y to database.



Time	T_1	T_2	Value
1	read_item(X);		X = 80
2	X:=X-N;		X =80–5=75 (which is not written into database)
3		read_item(X);	X = 80 (T2 still reads in the original value of X, the updated value of X is lost.)
4		X:=X+M	X=80+4=84
5	write_item(X);		X=75 is written into database
6	read_item(Y);		
7		write_item(X);	X=84 over writes X=75, a wrong record is written in database
8	Y:=Y+N;		
9	write_item(Y);		

Why Concurrency control: Problems

- The types of problems we may encounter with these two simple transactions if they run concurrently.
  - **The Temporary Update (or Dirty Read) Problem**: This problem occurs when one transaction updates a database item and then the transaction fails for some reason. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value.



T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

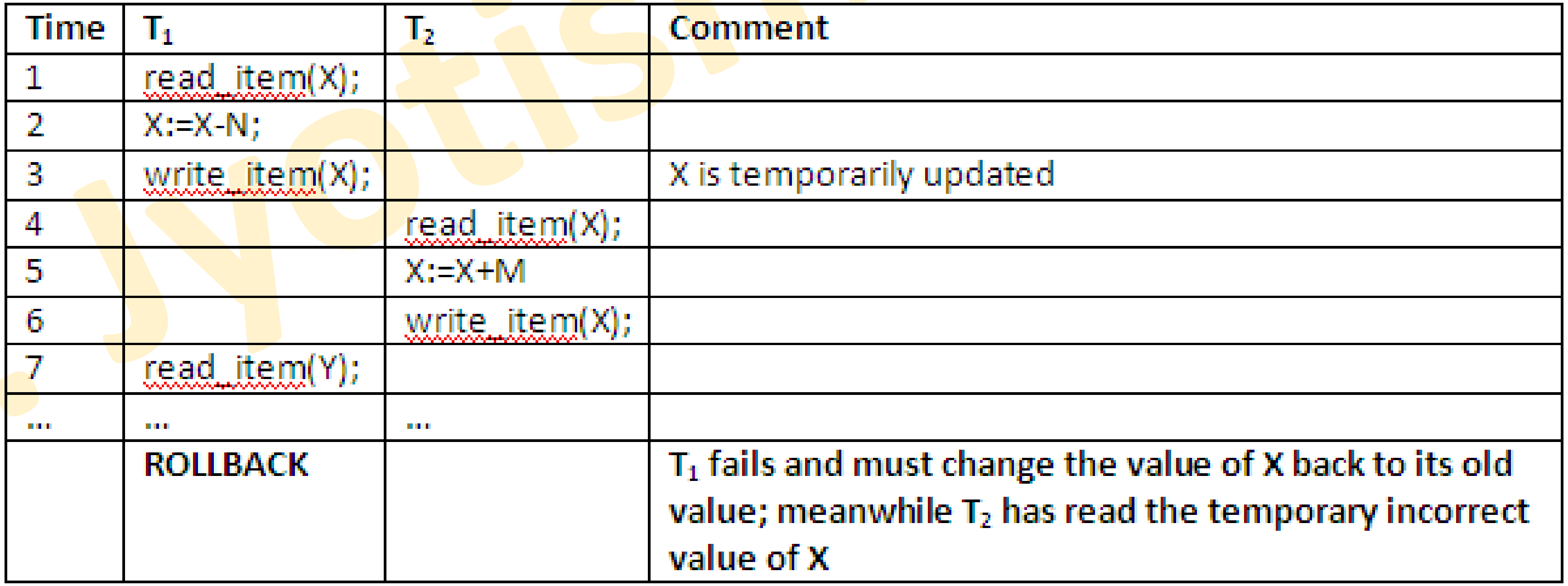
	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

Why Concurrency control: Problems: The Temporary Update (or Dirty Read) Problem: Example

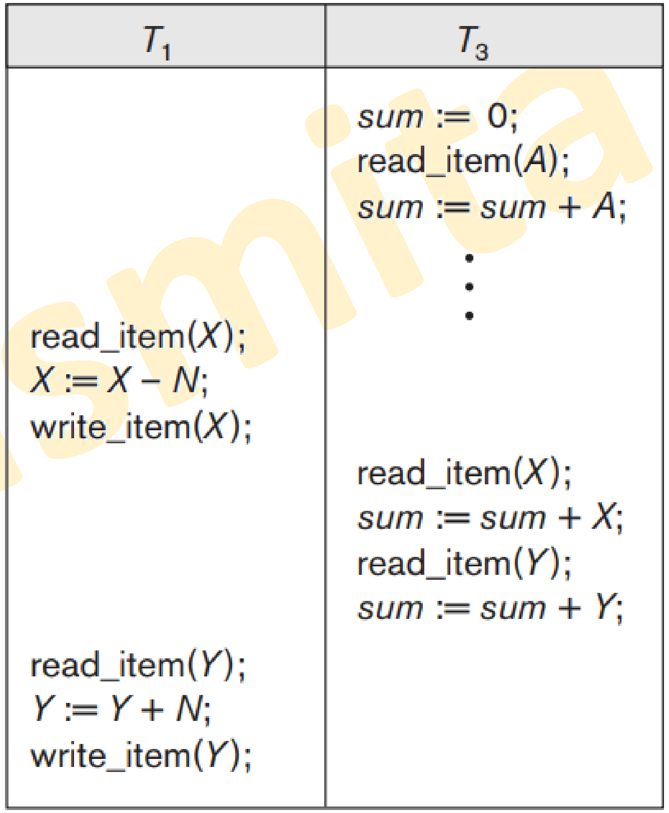
- The figure below shows an example where T1 updates item X and then fails before completion, so the system must change X back to its original value. Before it can do so, however, transaction T2 reads the 'temporary' value of X, which will not be recorded permanently in the database because of the failure of T1.
- The value of item X that is read by T2 is called dirty data, because it has been created by a transaction that has not been completed and committed yet; hence this problem is also known as the dirty read problem.
- Since the dirty data read in by T2 is only a temporary value of X, the problem is sometimes called temporary update too.



Time	T_1	T_2	Comment
1	read_item(X);		
2	X:=X-N;		
3	write_item(X);		X is temporarily updated
4		read_item(X);	
5		X:=X+M	
6		write_item(X);	
7	read_item(Y);		
…	…	…	
	ROLLBACK		**T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the temporary incorrect value of X**

Why Concurrency control: Problems

- The types of problems we may encounter with these two simple transactions if they run concurrently.
  - **The Incorrect Summary Problem**: If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated.



T_1	T_3
	$sum := 0$;
	read_item(A);
	$sum := sum + A$;
	⋮
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	$sum := sum + X$;
	read_item(Y);
	$sum := sum + Y$;
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Why Concurrency control: Problems: The Incorrect Summary Problem Example

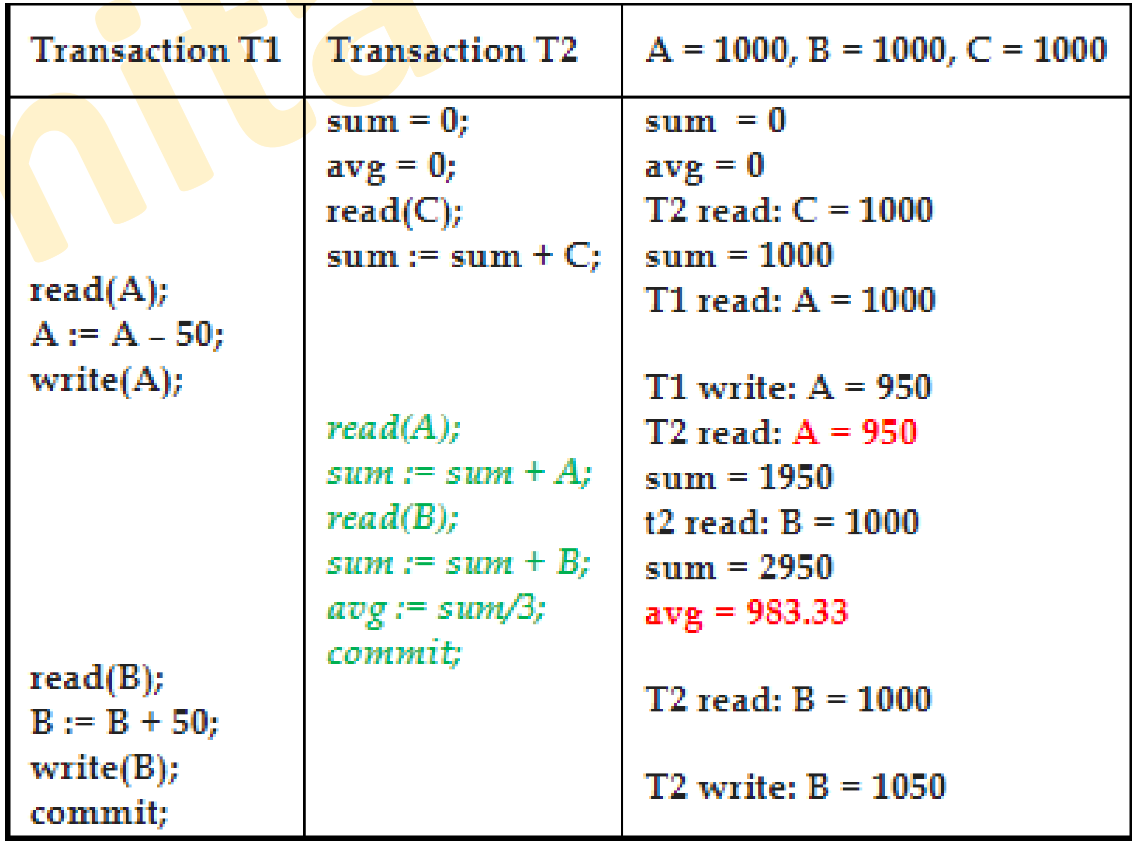
- Consider the schedule S1 given below, in which, transaction T1 transfers money from account A to account B and in the mean time, transaction T2 calculates the sum of 3 accounts namely, A, B, and C. The third column shows the account balances and calculated values after every instruction is executed.

Transaction T2 reads the value of account A after A is updated and reads B before B is updated. [The portion that violates in T2 is highlighted in green color]. Hence, the aggregate operation is end up with an inconsistent result.

If all the instructions in T1 are executed before T2 starts, then A will be 950, B will be 1050 and average value will be 1000.

If all the instructions in T1 are executed after T2 finishes, then A will be 950, B will be 1050 and average value will be 1000.

But, due to this interleaved execution, the final value of A is 950, B is 1050, and average is 983.33 which is wrong.



Transaction T1	Transaction T2	A = 1000, B = 1000, C = 1000
	sum = 0;	sum = 0
	avg = 0;	avg = 0
	read(C);	T2 read: C = 1000
	sum := sum + C;	sum = 1000
read(A);		T1 read: A = 1000
A := A – 50;		
write(A);		T1 write: A = 950
	read(A);	T2 read: A = 950
	sum := sum + A;	sum = 1950
	read(B);	t2 read: B = 1000
	sum := sum + B;	sum = 2950
	avg := sum/3;	avg = 983.33
	commit;	
read(B);		T2 read: B = 1000
B := B + 50;		
write(B);		T2 write: B = 1050
commit;		

Need for locking

- Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items.
- A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it.
- Generally, there is one lock for each data item in the database.
- Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

Lock Based Protocols: Binary Locks

- Several types of locks are used in concurrency control.
  - **Binary Locks**:
    - A **binary lock** can have two **states** or **values**: **locked** and **unlocked** (or 1 and 0, for simplicity). A distinct lock is associated with each database item X.
    - If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1.
    - We refer to the current value (or state) of the lock associated with item X as **lock(X)**.
    - Two operations, **lock_item** and **unlock_item**, are used with binary locking.
    - A transaction requests access to an item X by first issuing a lock_item(X) operation.
    - If LOCK(X) = 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the transaction locks the item) and the transaction is allowed to access item X.
    - When the transaction is through using the item, it issues an unlock_item(X) operation, which sets LOCK(X) back to 0 (unlocks the item) so that X may be accessed by other transactions.
    - Hence, a binary lock enforces mutual exclusion on the data item.

Lock Based Protocols: Binary Locks

- If the simple binary locking scheme described here is used, every transaction must obey the following rules:
  - A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.
  - A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.
  - A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X.
  - A transaction T will not issue an unlock_item(X) operation unless it already holds the lock on item X.

Lock Based Protocols: Shared/Exclusive (or Read/Write) Locks

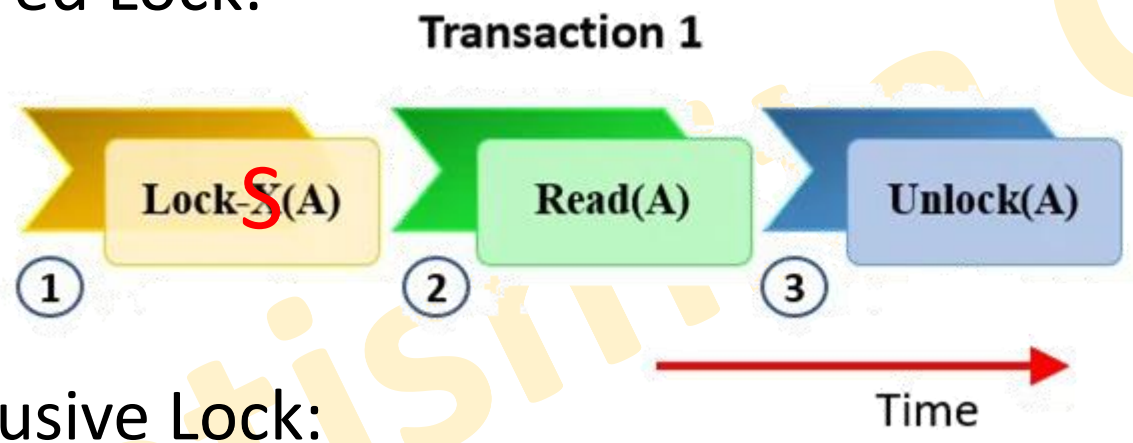
- The preceding binary locking scheme is too restrictive for database items because at most one transaction can hold a lock on a given item.
- We should allow several transactions to access the same item X if they all access X for reading purposes only.
- This is because read operations on the same item by different transactions are not conflicting.
- However, if a transaction is to write an item X, it must have exclusive access to X.
- For this purpose, a different type of lock, called a **multiple-mode lock**, is used.
- In this scheme—called **shared/exclusive** or **read/write** locks—there are three locking operations: read_lock(X), write_lock(X), and unlock(X).

Lock Based Protocols: Shared/Exclusive (or Read/Write) Locks

- A lock associated with an item X, LOCK(X), now has three possible states: *read-locked*, *write-locked*, or *unlocked*.
- A **read-locked item** is also called **share-locked (**Often represented as **lock-S())** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked (**Often represented as **lock-X())** because a single transaction exclusively holds the lock on the item.
- Multiple read locks can exist at the same time.
- The write transaction should wait for read locks to finish reading.
- If you have the write lock before the read lock, the write lock will block other transactions to read or write the same table. If you have the read lock before the write lock, the read lock will block the write transactions until the reading transaction finishes.

Lock Based Protocols: Shared/Exclusive (or Read/Write) Locks

- Example of Shared Lock:



Transaction 1

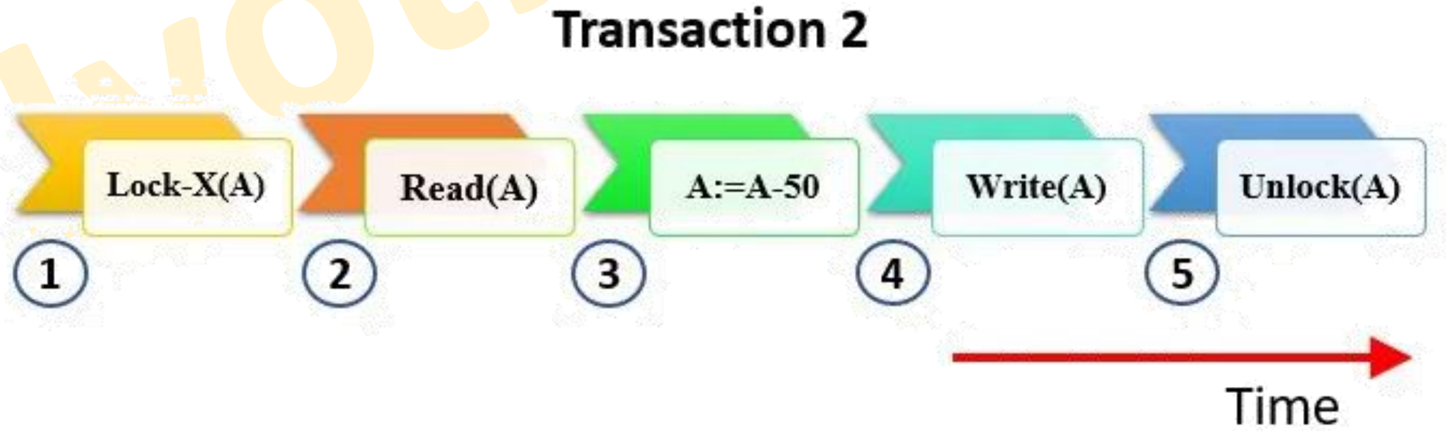
1 Lock-~~X~~S(A)

2 Read(A)

3 Unlock(A)

Time

- Example of Exclusive Lock:



Transaction 2

1 Lock-X(A)

2 Read(A)

3 A:=A-50

4 Write(A)

5 Unlock(A)

Time

Lock Based Protocols: Shared/Exclusive (or Read/Write) Locks

- When we use the shared/exclusive locking scheme, the system must enforce the following rules:
  1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
  2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.
  3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.
  4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.
  5. A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X.
  6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Lock Based Protocols: Shared/Exclusive (or Read/Write) Locks: Conversion (Upgrading, Downgrading) of Locks

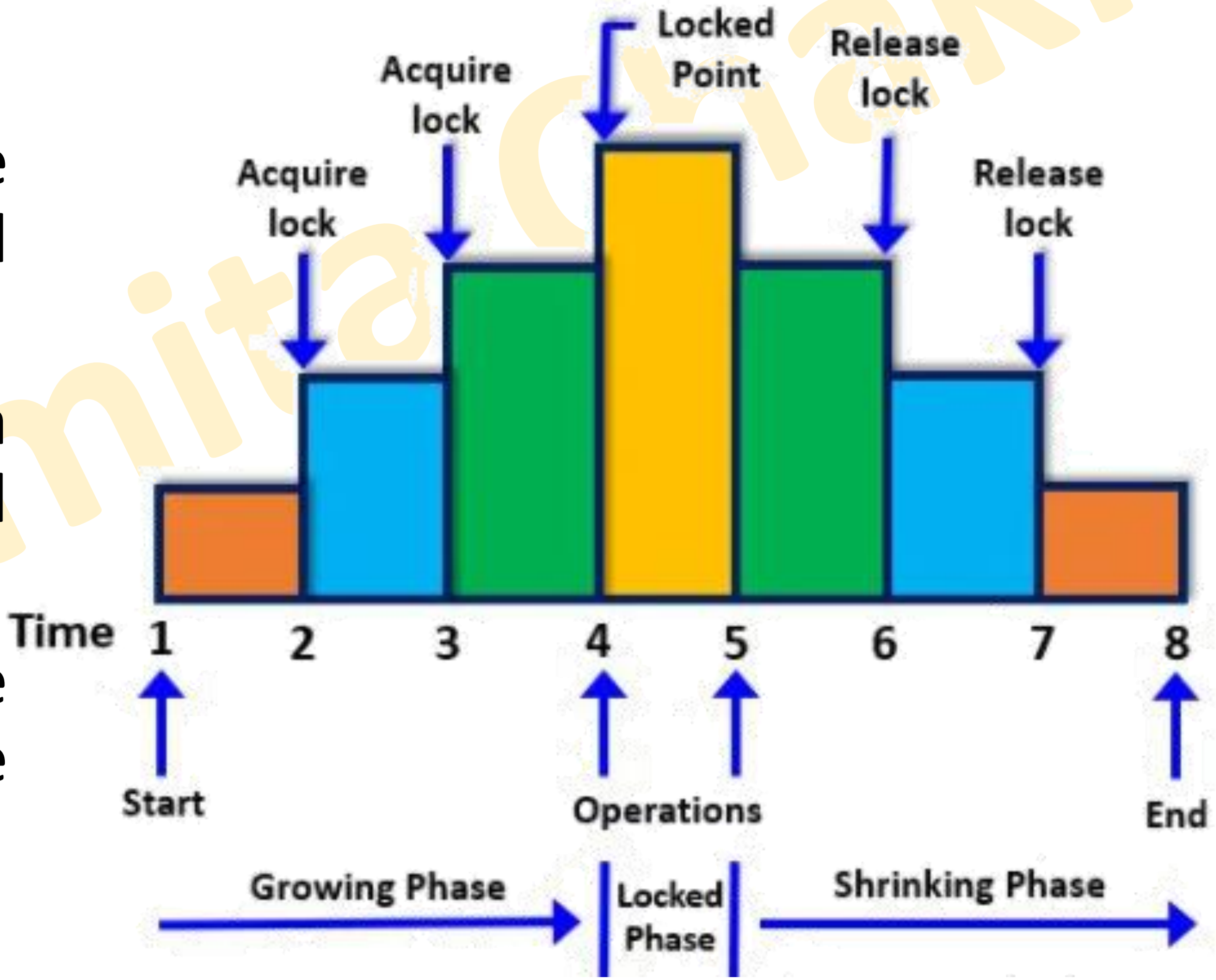
- It is desirable to relax conditions 4 and 5 in the preceding list in order to allow **lock conversion**; that is, a transaction that already holds a lock on item X is allowed under certain conditions to **convert** the lock from one locked state to another.
- For example, it is possible for a transaction T to issue a read_lock(X) and then later to **upgrade** the lock by issuing a write_lock(X) operation.
- If T is the only transaction holding a read lock on X at the time it issues the write_lock(X) operation, the lock can be upgraded; otherwise, the transaction must wait.
- It is also possible for a transaction T to issue a write_lock(X) and then later to **downgrade** the lock by issuing a read_lock(X) operation.

Two-Phase Locking: Guaranteeing Serializability

- Using binary locks or read/write locks in transactions, as described earlier, does not guarantee serializability of schedules on its own.
- A transaction is said to follow the **two-phase locking protocol** if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction.
- Such a transaction can be divided into two phases: an **expanding** or **growing (first) phase**, during which new locks on items can be acquired but none can be released; and a **shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired.
- If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase.

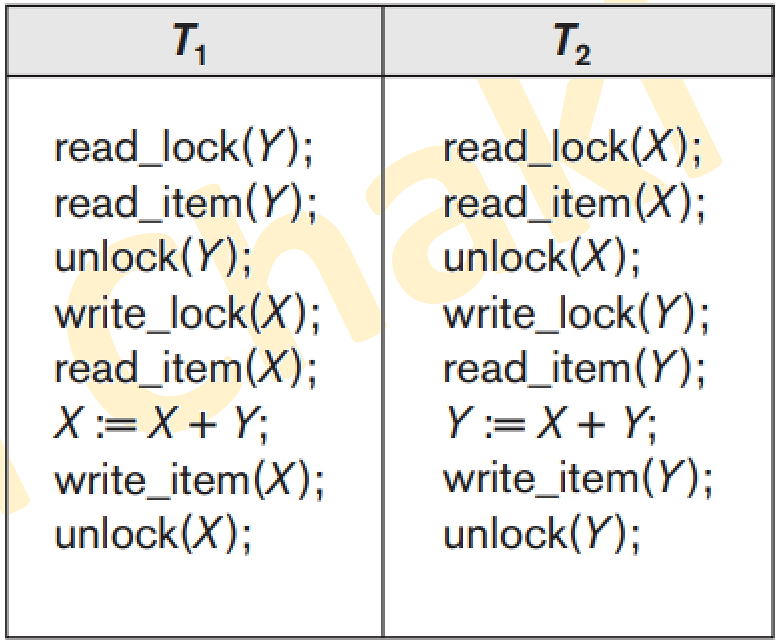
Two-Phase Locking: Guaranteeing Serializability

- The above phases in a DBMS are determined by something called a ‘**Lock Point**’.
- **Lock point** is the point where a transaction has achieved its final lock.
- It is also the point where the growing phase ends and the shrinking phase begins.



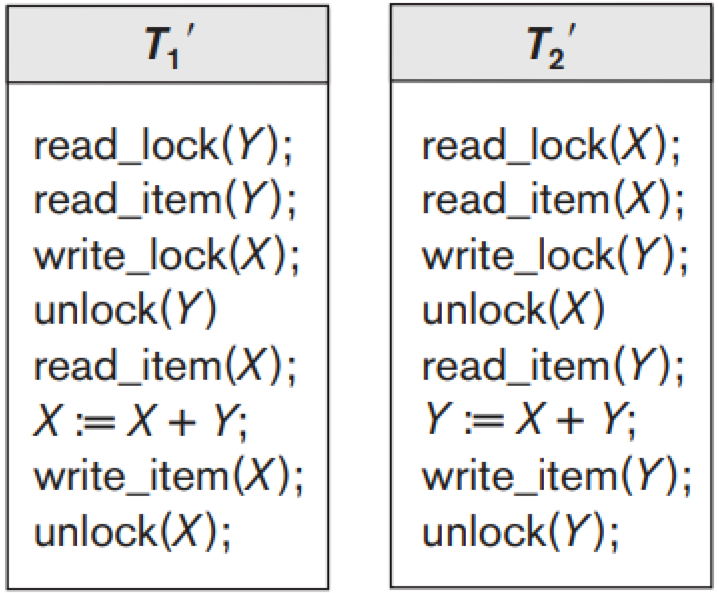
Two-Phase Locking: Guaranteeing Serializability

- Transactions T1 and T2 in top Figure do not follow the two-phase locking protocol because the write_lock(X) operation follows the unlock(Y) operation in T1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T2. [*Initial values: X=20, Y=30, Result serial schedule T1 followed by T2: X=50, Y=80, Result of serial schedule T2 followed by T1: X=70, Y=50*]
- If we enforce two-phase locking, the transactions can be rewritten as T1′ and T2′, as shown in bottom Figure



T_1	T_2

read_lock(*Y*);
read_item(*Y*);
unlock(*Y*);
write_lock(*X*);
read_item(*X*);
$X := X + Y$;
write_item(*X*);
unlock(*X*);
	read_lock(*X*);
	read_item(*X*);
	unlock(*X*);
	write_lock(*Y*);
	read_item(*Y*);
	$Y := X + Y$;
	write_item(*Y*);
	unlock(*Y*);



T_1'

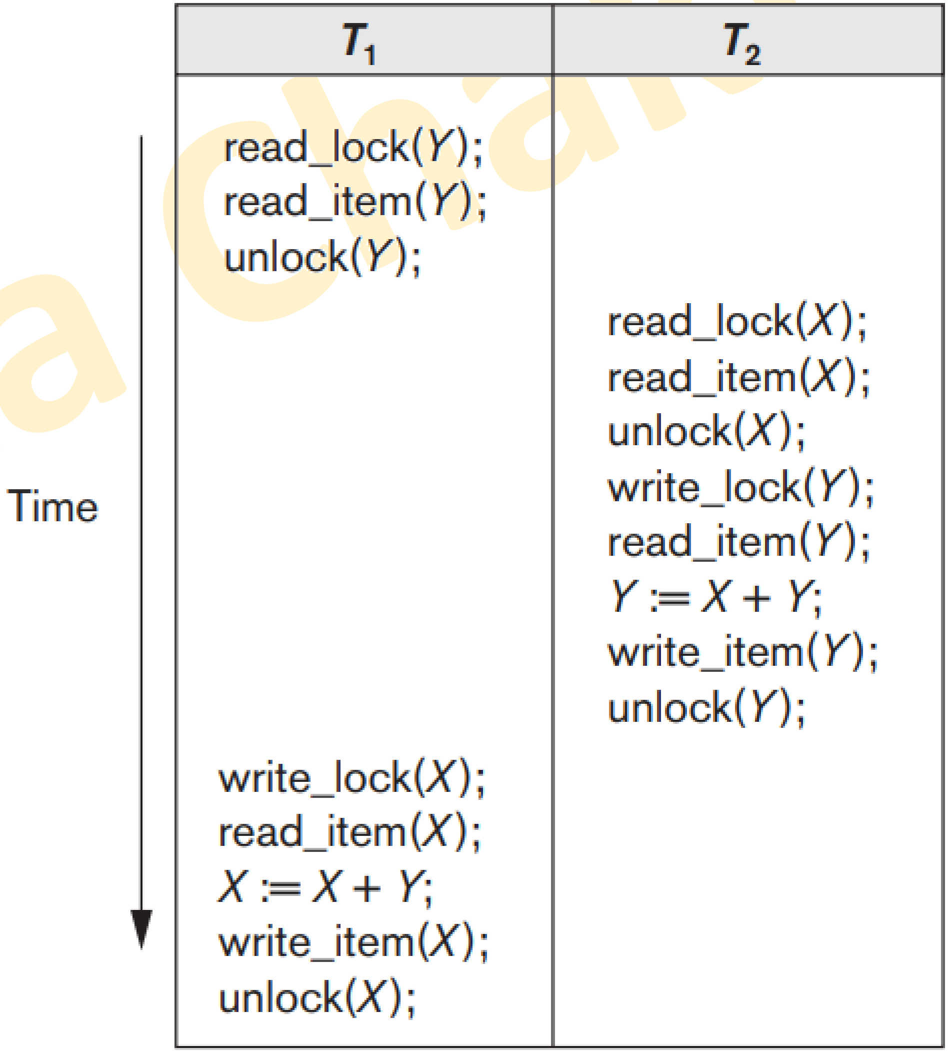
read_lock(*Y*);
read_item(*Y*);
write_lock(*X*);
unlock(*Y*)
read_item(*X*);
$X := X + Y$;
write_item(*X*);
unlock(*X*);

T_2'

read_lock(*X*);
read_item(*X*);
write_lock(*Y*);
unlock(*X*)
read_item(*Y*);
$Y := X + Y$;
write_item(*Y*);
unlock(*Y*);

Two-Phase Locking: Guaranteeing Serializability

- The schedule shown in this Figure is not permitted for T1′ and T2′ (with their modified order of locking and unlocking operations) under the rules of locking described because T1′ will issue its write_lock(X) before it unlocks item Y; consequently, when T2′ issues its read_lock(X), it is forced to wait until T1′ releases the lock by issuing an unlock (X) in the schedule.



T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);	

	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	Y := X + Y;
	write_item(Y);
	unlock(Y);

write_lock(X);
read_item(X);
X := X + Y;
write_item(X);
unlock(X);	

Two-Phase Locking: Guaranteeing Serializability

- It can be proved that, if every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable, obviating the need to test for serializability of schedules.
- The locking protocol, by enforcing two-phase locking rules, also enforces serializability.
- Although the two-phase locking protocol guarantees serializability (that is, every schedule that is permitted is serializable), it does not permit all possible serializable schedules (that is, some serializable schedules will be prohibited by the protocol).

Two-Phase Locking: Solution of lost update problem

Time	T1	T2	A
t1		Write_lock(A)	100
t2	Write_lock(A)	Read_item(A)	100
t3	Wait	A:= A+100	100
t4	Wait	Write_item(A)	200
t5	Wait	Unlock(A)	200
t6	Wait	Commit	200
t7	Read_item(A)		200
t8	A:= A-10		200
t9	Write_item(A)		190
t10	Unlock(A)		190
t11	commit		190

Two-Phase Locking: Solution of incorrect summary problem

Time	T1	T2	A = 100	B = 50	C = 25
t1	Write_lock(A)	1/ Sum=0	100	50	25
t2	Write_lock(C)		100	50	25
t3	Read_item(A)	Read_lock(A)	100	50	25
t4	A:= A-10	Wait	100	50	25
t5	Write_item(A)	Wait	90	50	25
t6	Read_item(C)	Wait	90	50	25
t7	C:= C+10	Wait	90	50	25
t8	Write_item(C)	Wait	90	50	35
t9	Unlock(A, C)	Wait	90	50	35
t10	Commit	Wait	90	50	35
t11		Read_lock(B)	90	50	35
t12		Read_lock(C)	90	50	35
t13		Read_item(A)	90	50	35
t14		Sum:= Sum+A	90	50	35
t15		Read_lock(B)	90	50	35
t16		Sum:= Sum+B	90	50	35
t17		Read_item(C)	90	50	35
t18		Sum:= Sum+C	90	50	35
t19		Unlock(A, B, C)	90	50	35
t20		Commit	90	50	35

Two-Phase Locking: Variations

- The technique just described is known as **basic 2PL**.
- A variation known as **conservative 2PL (or static 2PL)** requires a transaction to lock all the items it accesses before the transaction begins execution, by predeclaring its read-set and write-set.
- The most popular variation of 2PL is **strict 2PL**, which guarantees strict schedules. In this variation, a transaction T does not release any of its exclusive (write) locks until after it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability.
- A more restrictive variation of strict 2PL is **rigorous 2PL**, which also guarantees strict schedules. In this variation, a transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL.

Two-Phase Locking: Example

- **2 PL:** There is growing and shrinking phase.
- **Strict 2 PL:** There is Lock-x(B) and it is unlocked before commit so no strict 2 PL.
- **Rigorous:** If it is not strict 2 PL then it can't be Rigorous.
- **Conservative:** If it is not strict 2 PL then it can't be conservative.

T1
Lock-s(A)
Read(A)
Lock-x(B)
Read(B)
Unlock(A)
Write(B)
Unlock(B)

Two-Phase Locking: Example

- **2 PL:** There is growing and shrinking phase so it is 2 PL.
- **Strict 2 PL:** Exclusive locks are unlocked after commit. So yes it is.
- **Rigorous:** We have taken Lock-s(A) and we have unlocked it before commit. So no rigorous.
- **Conservative:** We have not taken all the locks at first then start the transaction so no conservative.

T1
Lock-s(A)
Read(A)
Lock-x(B)
Unlock(A)
Read(B)
Write(B)
commit
Unlock(B)

Two-Phase Locking: Example

- **2 PL:** There is growing and shrinking phase so it is 2 PL.
- **Strict 2 PL:** Exclusive locks are unlocked after commit. So yes it is.
- **Rigorous:** We have unlocked all the locks after commit so it is rigorous.
- **Conservative:** We have not taken all the locks at first then start the transaction so no conservative.

T1
Lock-s(A)
Read(A)
Lock-x(B)
Read(B)
Write(B)
commit
Unlock(B)
Unlock(A)

Two-Phase Locking: Example

- **2 PL:** There is growing and shrinking phase so it is 2 PL.
- **Strict 2 PL:** Exclusive locks are unlocked after commit. So yes it is.
- **Rigorous:** We have unlocked all the locks after commit so it is rigorous.
- **Conservative:** We have taken all the locks at first then start the transaction so yes it is conservative.

T1
Lock-s(A)
Lock-x(B)
Read(B)
Write(B)
Read(A)
commit
Unlock(A)
Unlock(B)

Two-Phase Locking: Example

- **2 PL:** There is no growing and shrinking phase so it is not 2 PL.
- **Strict 2 PL:** Because it is not 2 P L so not either of it.
- **Rigorous:** Because it is not 2 P L so not either of it.
- **Conservative:** Because it is not 2 P L so not either of it.

T1
Lock-s(A)
Read(A)
Unlock(A)
Lock-x(B)
Read(B)
Write(B)
Unlock(B)
Unlock(A)
Commit

Two-Phase Locking: Variations

- The difference between strict and rigorous 2PL:
  - the former holds write-locks until it commits, whereas the latter holds all locks (read and write).
  - Also, the difference between conservative and rigorous 2PL is that the former must lock all its items before it starts, so once the transaction starts it is in its shrinking phase; the latter does not unlock any of its items until after it terminates (by committing or aborting), so the transaction is in its expanding phase until it ends.

Two-Phase Locking: Example

- Consider the following two transactions:
  - T31: read(A); read(B); if A = 0 then B := B + 1; write(B).
  - T32: read(B); read(A); if B = 0 then A := A + 1; write(A).
- Add lock and unlock instructions to transactions T31 and T32, so that they observe the two-phase locking protocol.
- Lock and unlock instructions:
  - T31: lock-S(A) read(A) lock-X(B) read(B) if A = 0 then B := B + 1 write(B) unlock(A) unlock(B)
  - T32: lock-S(B) read(B) lock-X(A) read(A) if B = 0 then A := A + 1 write(A) unlock(B) unlock(A)

Concurrency control based on timestamp

- The use of locking, combined with the 2PL protocol, guarantees serializability of schedules.
- The serializable schedules produced by 2PL have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire.
- If a transaction needs an item that is already locked, it may be forced to wait until the item is released.
- Some transactions may be aborted and restarted because of the deadlock problem.
- A different approach to concurrency control involves using transaction timestamps to order transaction execution for an equivalent serial schedule.

Concurrency control based on timestamp: Timestamps

- **Timestamp** values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the transaction start time.
- We will refer to the timestamp of transaction T as **TS(T)**. Concurrency control techniques based on timestamp ordering do not use locks; hence, **deadlocks cannot occur**.
- Timestamps can be generated in several ways.
    - One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, … in this scheme.
    - A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time.
    - Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

Concurrency control based on timestamp: Timestamp Ordering Algorithm

- The idea for this scheme is to enforce the equivalent serial order on the transactions based on their timestamps.
- A schedule in which the transactions participate is then serializable, and the only equivalent serial schedule permitted has the transactions in order of their timestamp values.
- This is called **timestamp ordering (TO)**.
- This differs from 2PL, where a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols.
- In timestamp ordering, however, the schedule is equivalent to the particular serial order corresponding to the order of the transaction timestamps.
- The algorithm allows interleaving of transaction operations, but it must ensure that for each pair of conflicting operations in the schedule, the order in which the item is accessed must follow the timestamp order.

Time	T1	T2	T3
1:00	Begin Transaction		
2:00		Begin Transaction	
3:00			Begin Transaction

TS(T1) = 10, TS(T2) = 20, TS (T3) = 30

Concurrency control based on timestamp: Timestamp Ordering Algorithm

- To do this, the algorithm associates with each database item X two timestamp (TS) values:
  1. **read_TS(X).** The read timestamp of item X is the largest timestamp among all the timestamps of transactions that have successfully read item X—that is, read_TS(X) = TS(T), where T is the youngest transaction that has read X successfully.

Time	T1	T2	T3
1:00	Begin Transaction		
2:00	R(X)		
2:15		Begin Transaction	
3:00		R(X)	
3:15			Begin Transaction
3:30			R (X)

read_TS(X) = 30

Concurrency control based on timestamp: Timestamp Ordering Algorithm

- To do this, the algorithm associates with each database item X two timestamp (TS) values:

  2. **write_TS(X).** The write timestamp of item X is the largest of all the timestamps of transactions that have successfully written item X—that is, write_TS(X) = TS(T), where T is the youngest transaction that has written X successfully. Based on the algorithm, T will also be the last transaction to write item X

Time	T1 (TS = 10)	T2 (TS = 20)	T3 (TS = 15)
1:00	Begin Transaction		
2:00	W(X)		
2:15			Begin Transaction
3:00			W (X)
3:15		Begin Transaction	
3:30		W(X)	

write_TS(X) = 20

Concurrency control based on timestamp: Basic Timestamp Ordering

- Whenever a transaction T issues a write_item(X) operation, the following check is performed:
  a) If TS(Ti) < Read_TS(X) then the operation is rejected, Ti abort.
  b) If TS(Ti) < Write_TS(X) then the operation is rejected and Ti is rolled back otherwise the operation is executed.
  c) If the condition in part (a) and (b) does not occur, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

Time	T1 (TS = 10)	T2 (TS = 20)
2:00		R(X)
2:15	W(X)	

a) T1 abort

Time	T1 (TS = 10)	T2 (TS = 20)
2:00		W(X)
2:15	W(X)	

b) T1 abort

Concurrency control based on timestamp: Basic Timestamp Ordering

- Whenever a transaction T issues a read_item(X) operation, the following check is performed:
  a) If Write_TS(X) >TS(Ti) then the operation is rejected.
  b) If Write_TS(X) <= TS(Ti) then the operation is executed.

Time	T1 (TS = 10)	T2 (TS = 20)
2:00		W(X)
2:15	R(X)	

a) T1 abort

- Whenever the basic TO algorithm detects two conflicting operations that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it.
- The schedules produced by basic TO are hence guaranteed to be conflict serializable.
- Deadlock does not occur with timestamp ordering.

Concurrency control based on timestamp: Basic Timestamp Ordering

T1 (TS = 10)	T2 (TS = 20)	T3 (TS = 30)
R(X)		
	R(Y)	
W(Z)		
		R(Y)
R(Z)		
	W(Y)	
		W(X)

	X	Y	Z
Read_TS	0	0	0
Write_TS	0	0	0

$R_{T1}(X)$

	X	Y	Z
Read_TS	10	0	0
Write_TS	0	0	0

$R_{T2}(Y)$

	X	Y	Z
Read_TS	10	20	0
Write_TS	0	0	0

$W_{T1}(Z)$

	X	Y	Z
Read_TS	10	20	0
Write_TS	0	0	10

$R_{T3}(Y)$

	X	Y	Z
Read_TS	10	30	0
Write_TS	0	0	10

$R_{T1}(Z)$

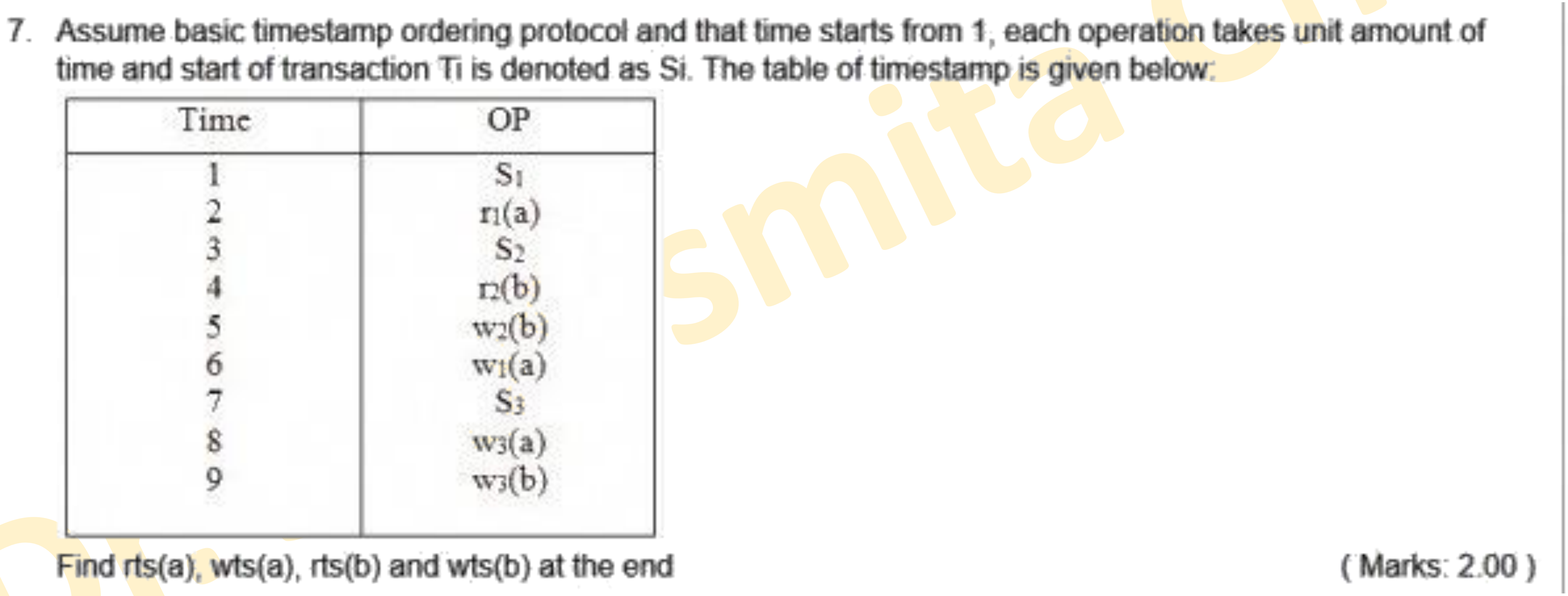
	X	Y	Z
Read_TS	10	30	10
Write_TS	0	0	10

$W_{T2}(Y)$: T2 abort/roll back. After T3 again T2 will start with a new timestamp

$W_{T3}(X)$

	X	Y	Z
Read_TS	10	30	10
Write_TS	30	0	10

Concurrency control based on timestamp: Basic Timestamp Ordering



7. Assume basic timestamp ordering protocol and that time starts from 1, each operation takes unit amount of time and start of transaction Ti is denoted as Si. The table of timestamp is given below:

Time	OP
1	S_1
2	$r_1(a)$
3	S_2
4	$r_2(b)$
5	$w_2(b)$
6	$w_1(a)$
7	S_3
8	$w_3(a)$
9	$w_3(b)$

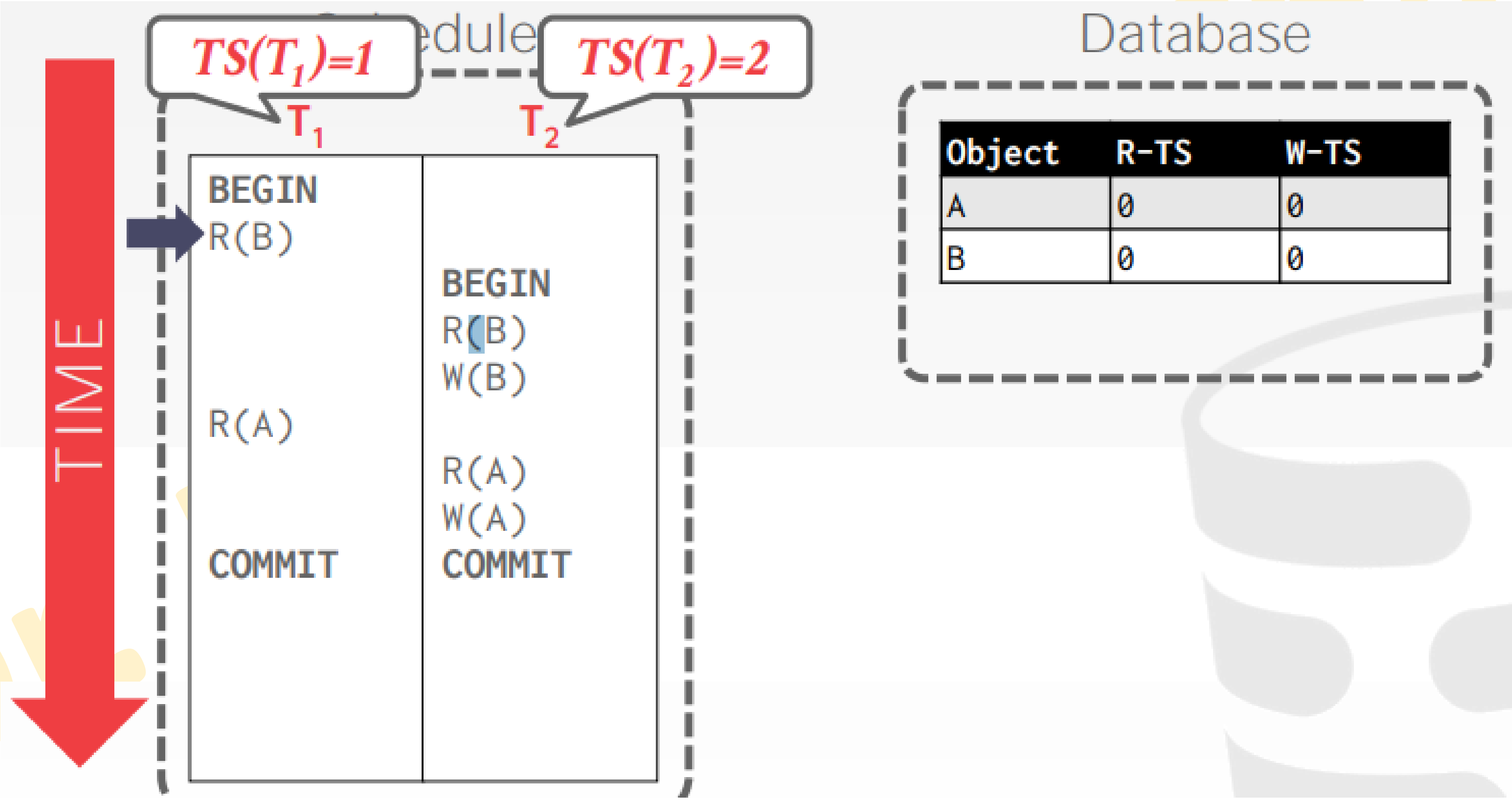
Find rts(a), wts(a), rts(b) and wts(b) at the end

(Marks: 2.00)

Concurrency control based on timestamp: Basic Timestamp Ordering

- T1 starts at TS =1
- T2 starts at TS = 3
- T3 starts at TS =7.
- While giving the TS for any read or write always look for youngest.
- **RTS(a)** = a is first read by T1 hence RTS(a) =1. (Read(a) is never done anywhere again hence it is youngest)
- **WTS(a)** = a is first written by T1 hence WTS(a) = 1. But again written by T3 which has higher TS (youngest) Hence final TS of WTS(a) = 7
- **RTS(b)** = b is first read by T2 hence RTS(b) =3. (Read(b) is never done anywhere again hence it is youngest)
- **WTS(b)** = b is first written by T2 hence WTS(b) = 3. But again written by T3 which has higher TS (youngest) Hence final TS of W(b) = 7

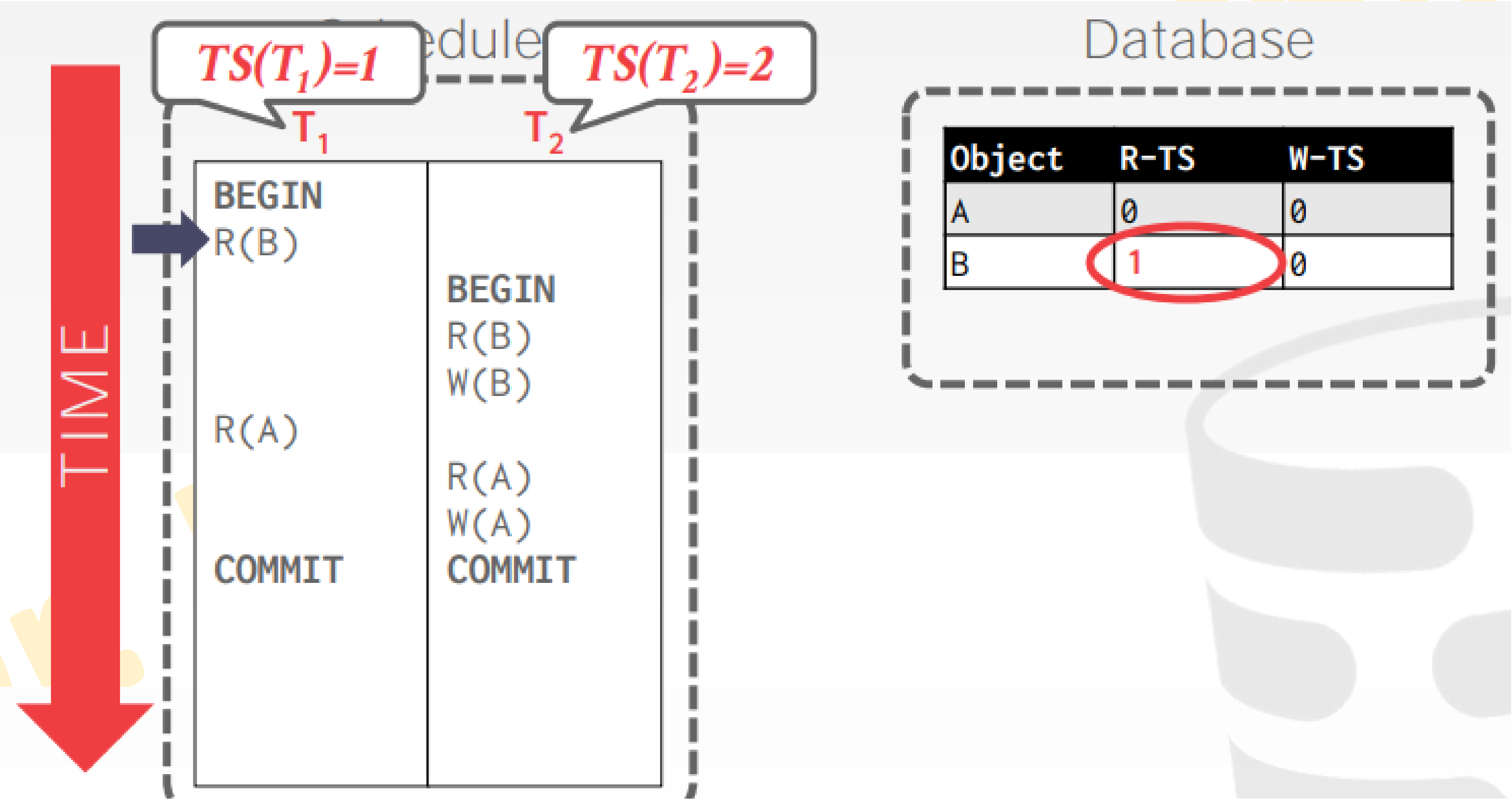
Concurrency control based on timestamp: Basic Timestamp Ordering



T_1	T_2
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	0	0

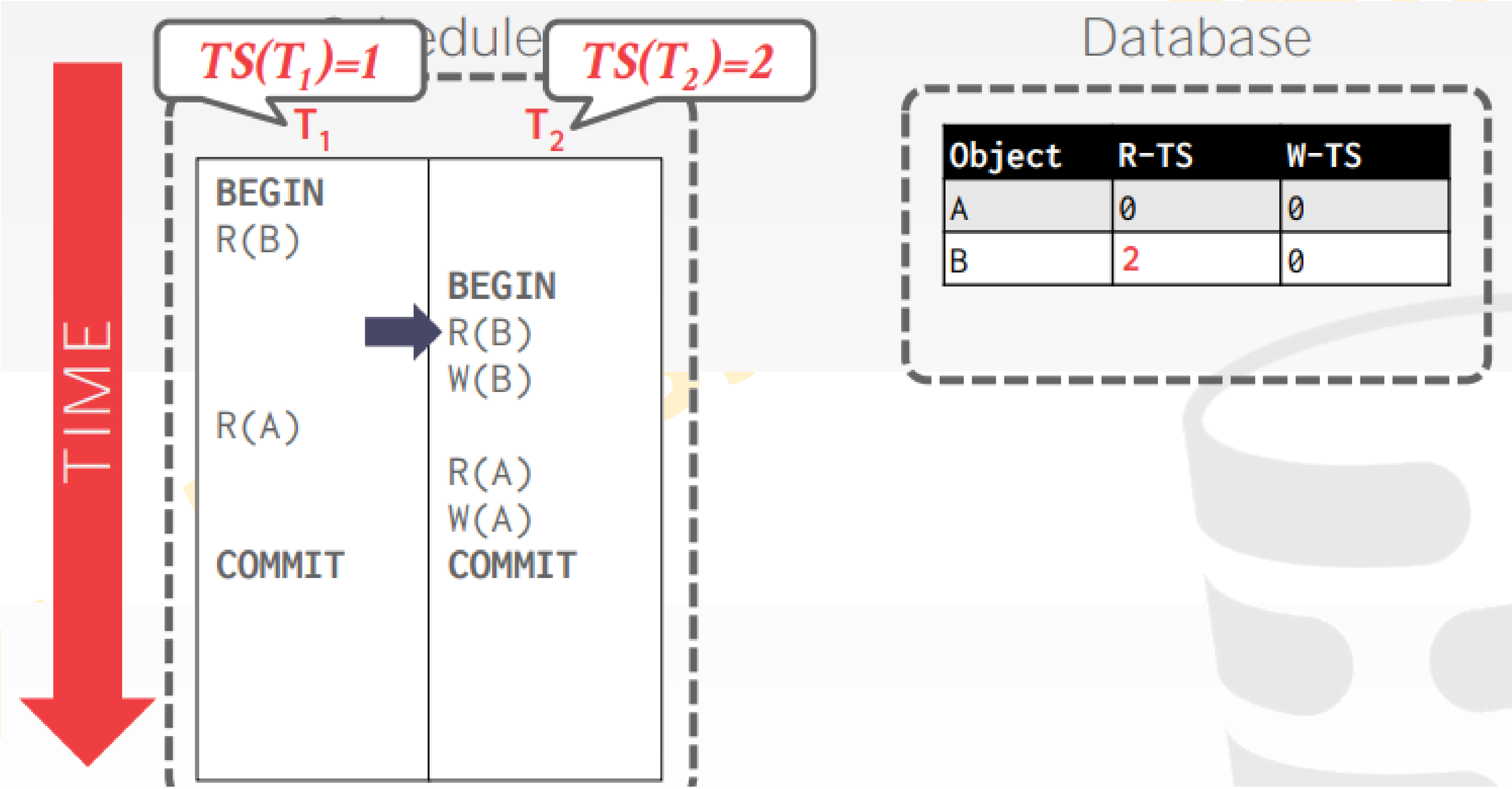
Concurrency control based on timestamp: Basic Timestamp Ordering



T_1	T_2
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	1	0

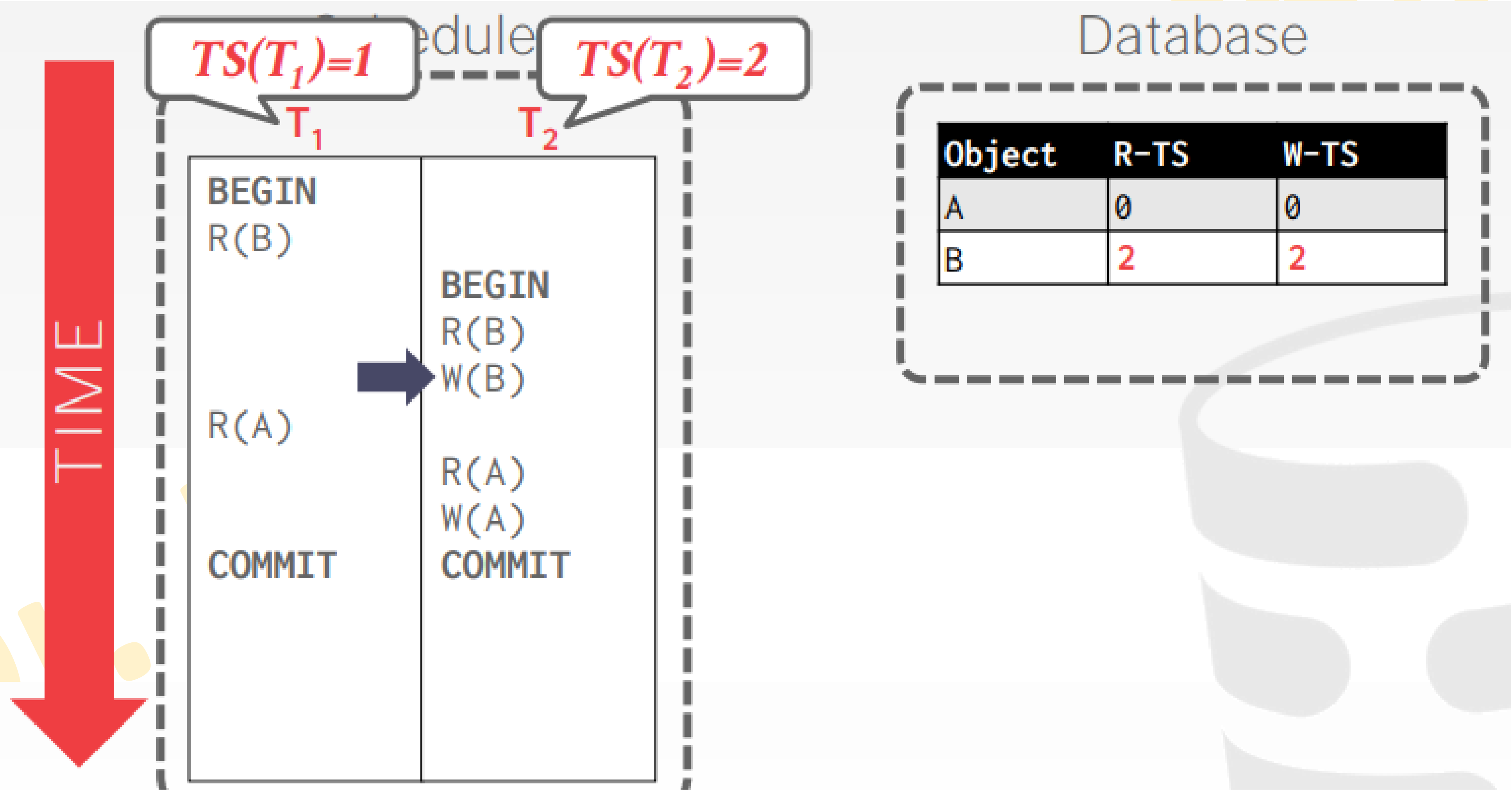
Concurrency control based on timestamp: Basic Timestamp Ordering



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	0

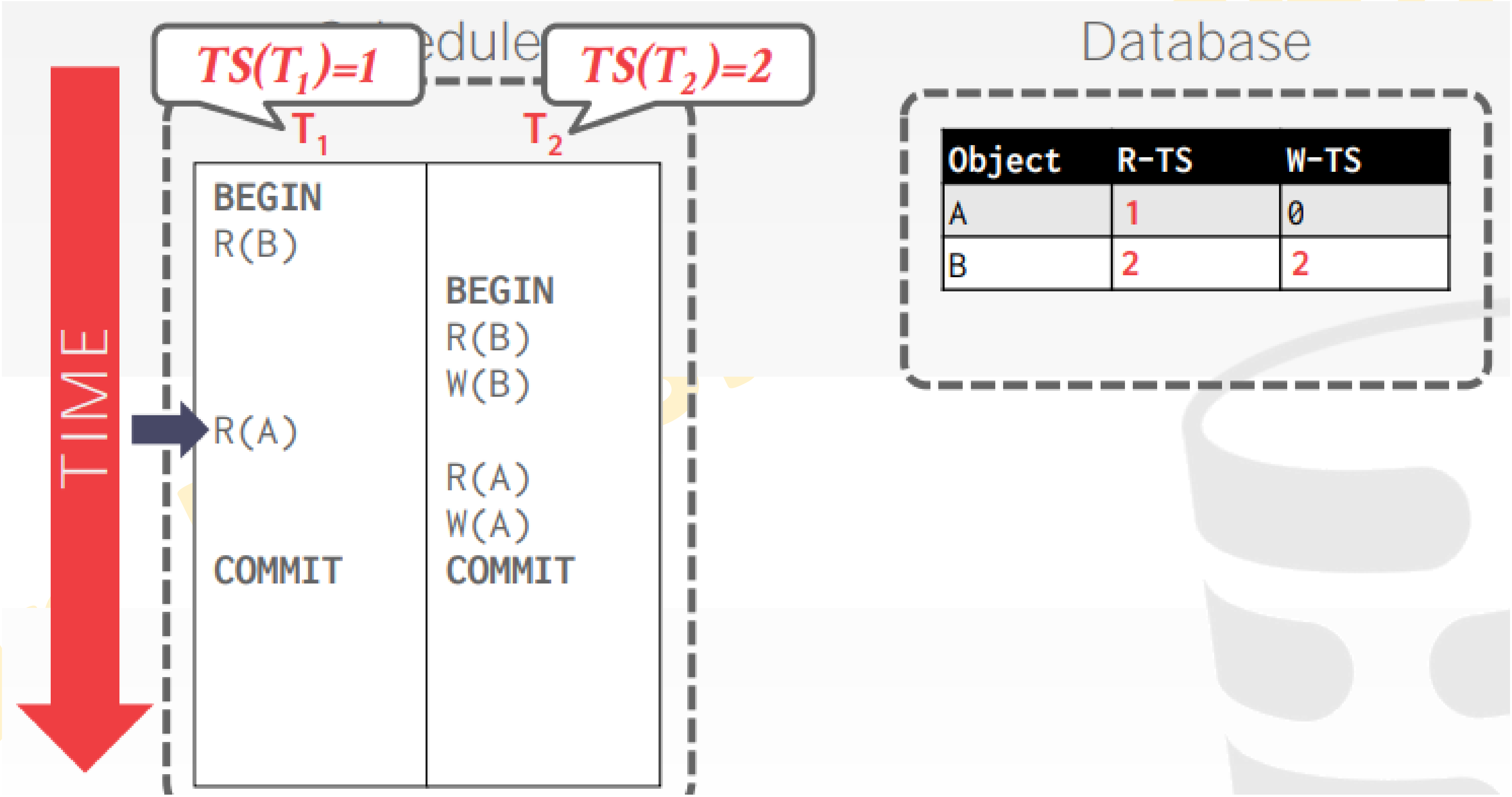
Concurrency control based on timestamp: Basic Timestamp Ordering



T_1	T_2
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

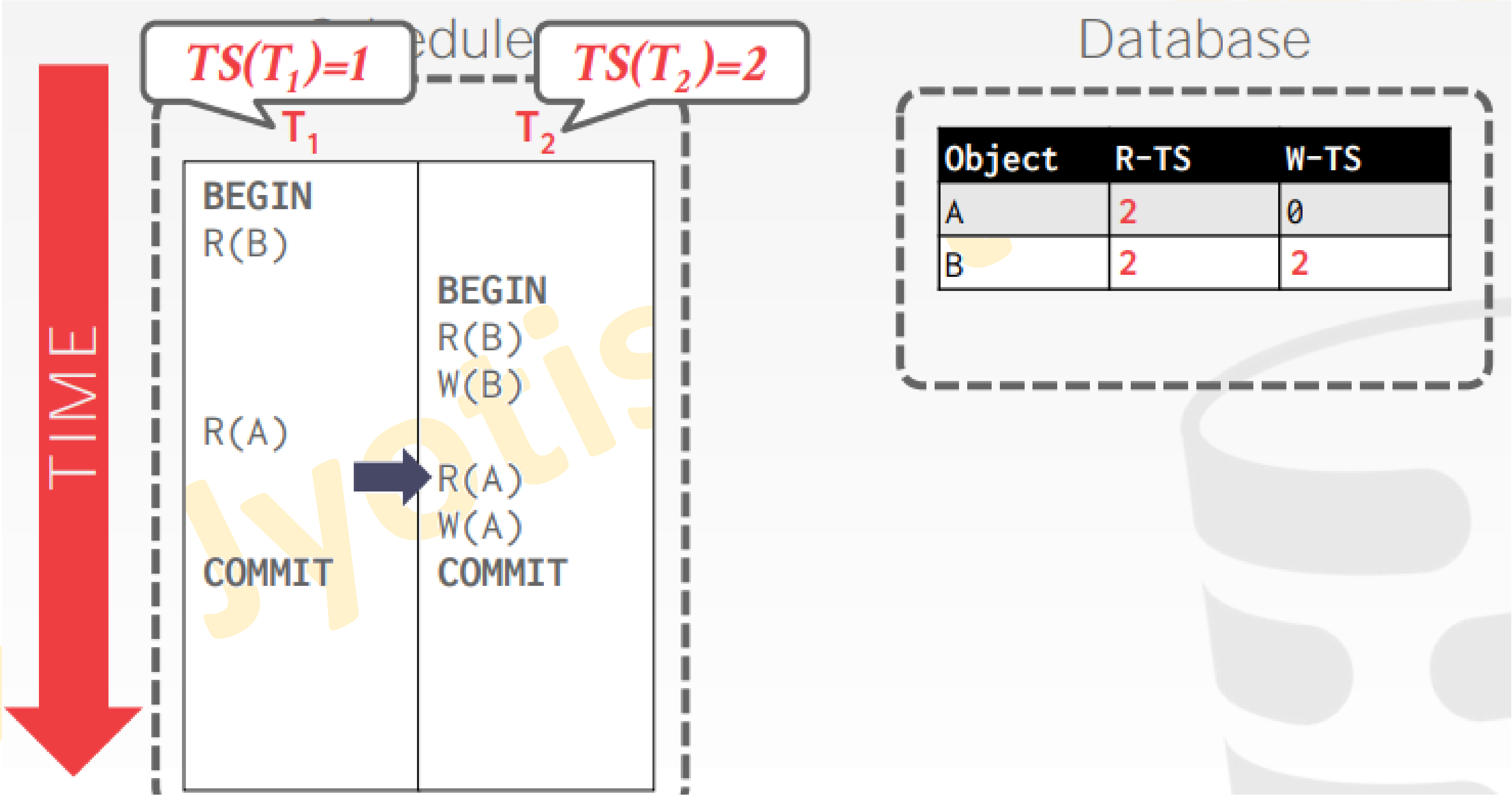
Concurrency control based on timestamp: Basic Timestamp Ordering



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

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

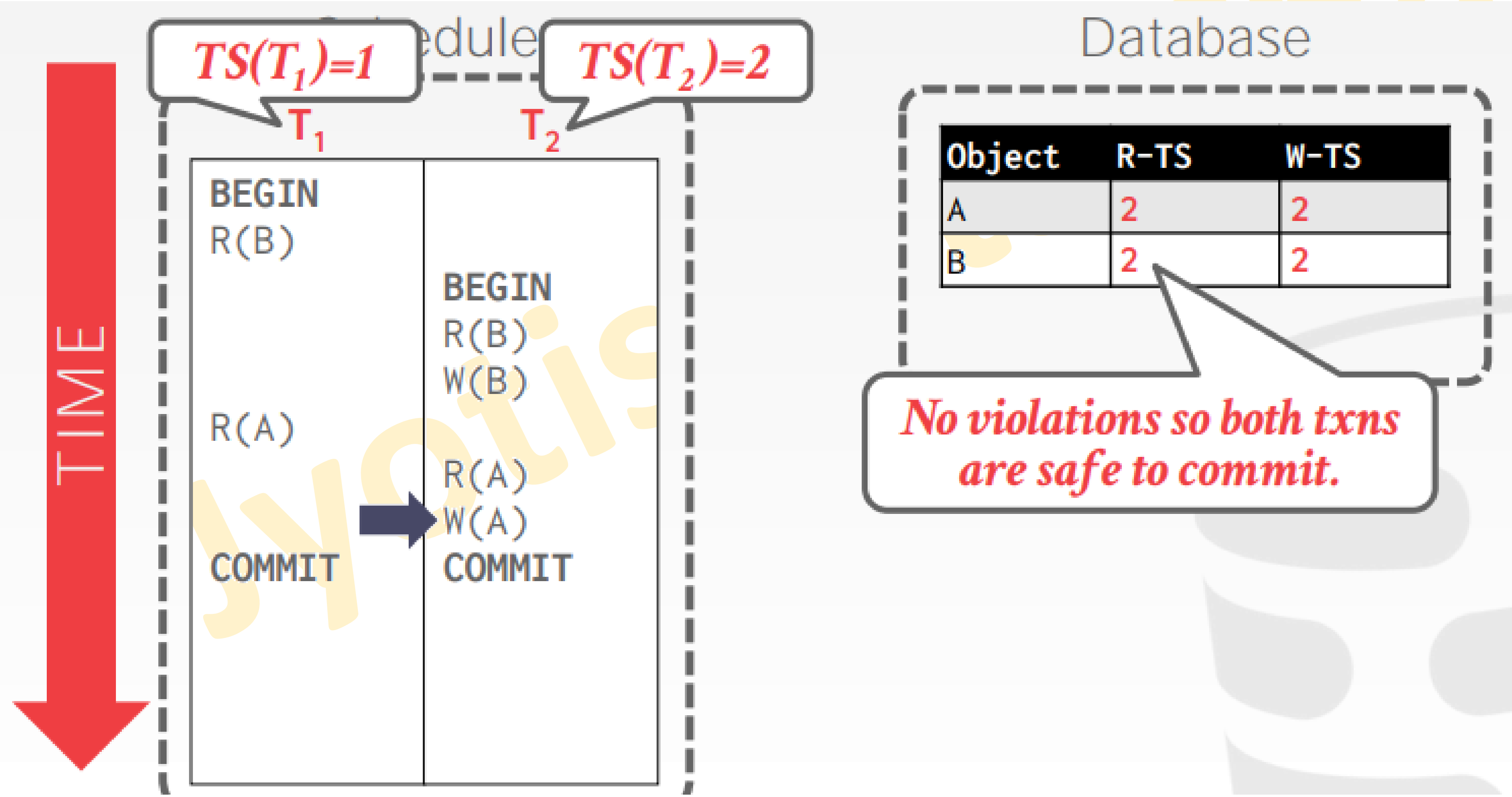
Concurrency control based on timestamp: Basic Timestamp Ordering



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

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

Concurrency control based on timestamp: Basic Timestamp Ordering

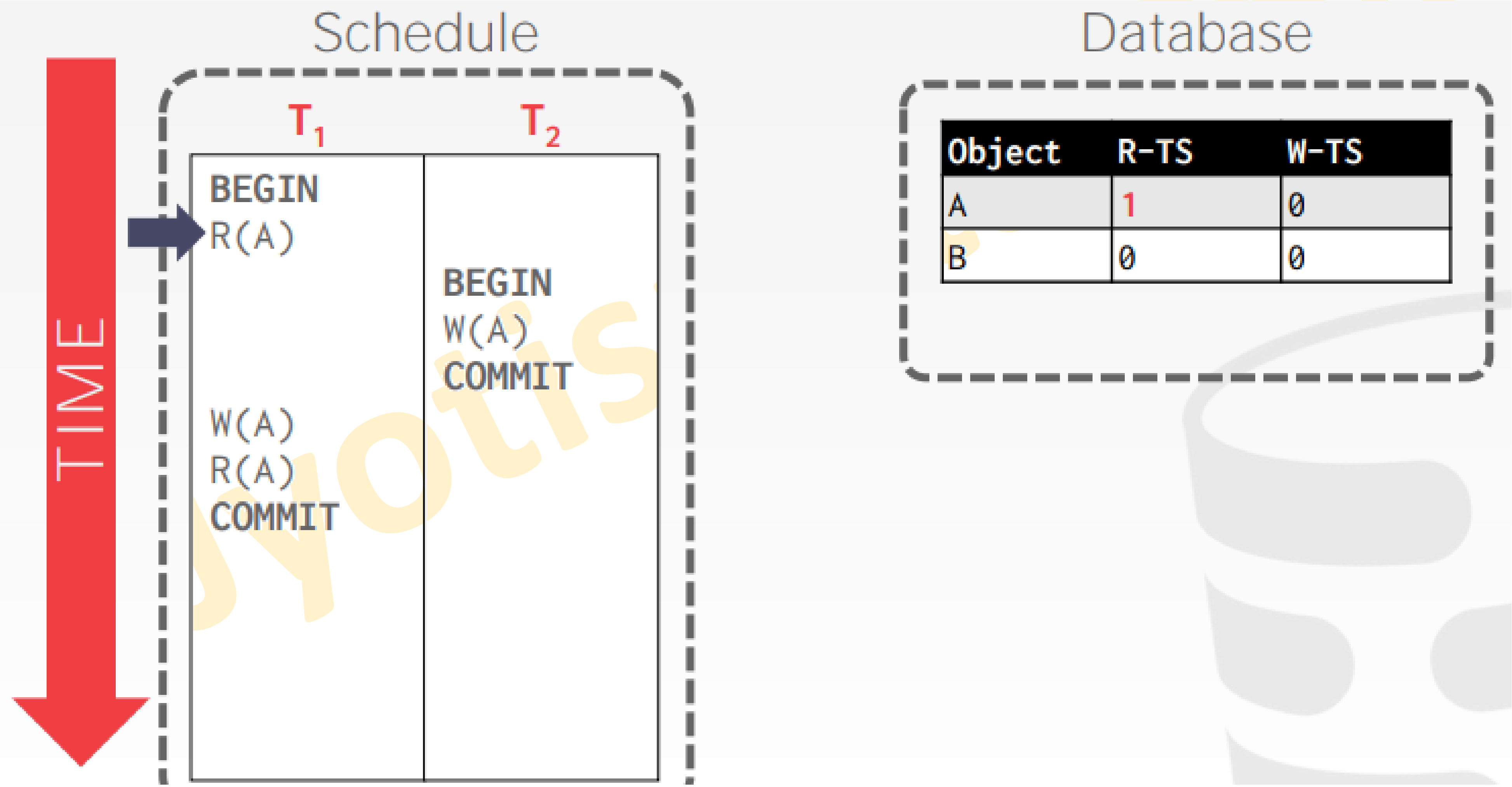


T_1	T_2
BEGIN	
R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
	W(A)
COMMIT	COMMIT

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

No violations so both txns are safe to commit.

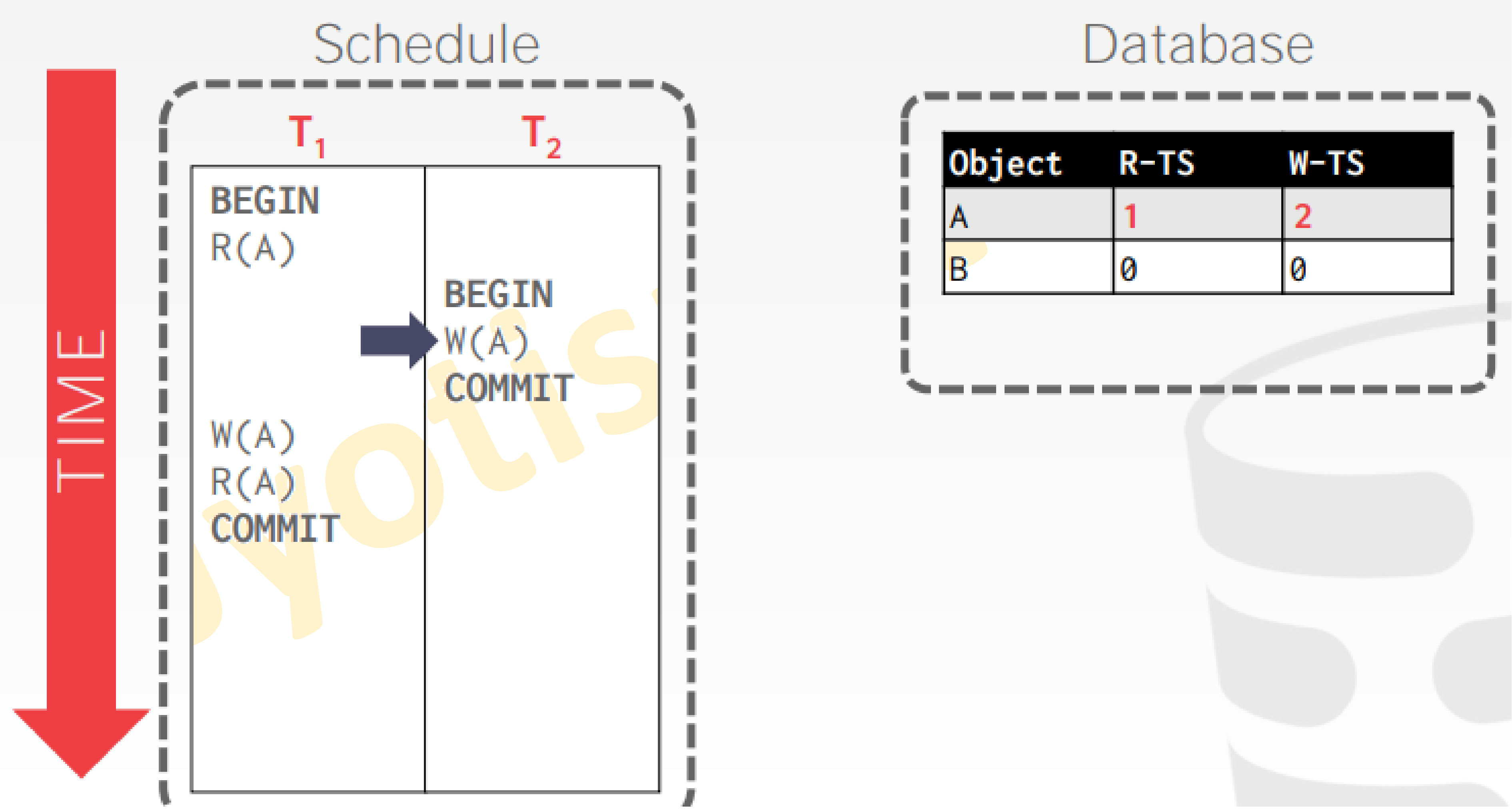
Concurrency control based on timestamp: Basic Timestamp Ordering



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

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

Concurrency control based on timestamp: Basic Timestamp Ordering



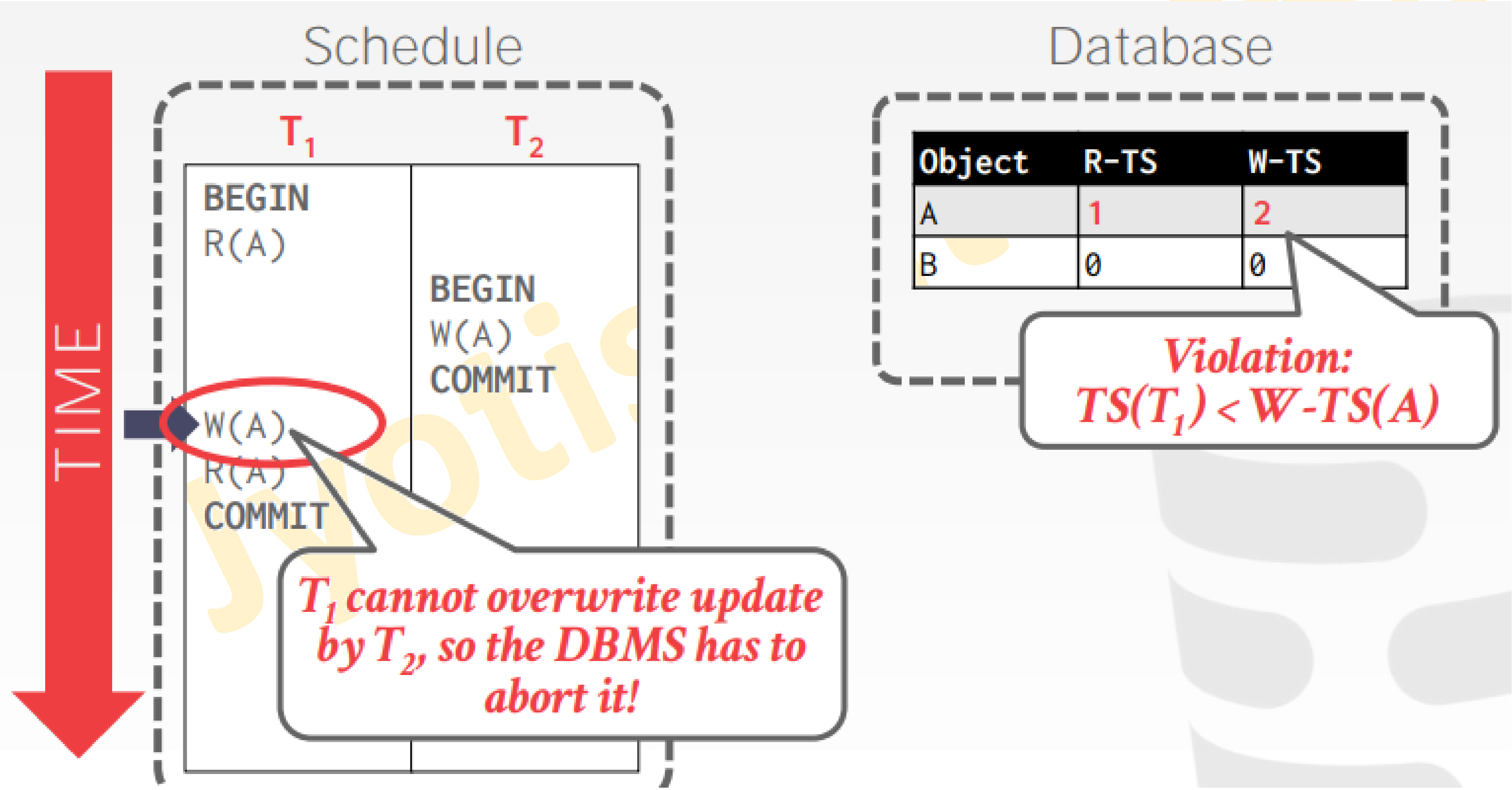
Schedule

T1	T2
BEGIN	
R(A)	
	BEGIN
	W(A)
	COMMIT
W(A)	
R(A)	
COMMIT	

Database

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

Concurrency control based on timestamp: Basic Timestamp Ordering



T_1	T_2

BEGIN
R(A)	

	BEGIN
	W(A)
	COMMIT

W(A)
R(A)
COMMIT	

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

Violation: $TS(T_1) < W\text{-}TS(A)$

T_1 cannot overwrite update by T_2, so the DBMS has to abort it!

Concurrency control based on timestamp: Strict Timestamp Ordering

- A variation of basic TO called strict TO ensures that the schedules are both strict (for easy recoverability) and (conflict) serializable.
- In this variation, a transaction T issues a read_item(X) or write_item(X) such that TS(T) > write_TS(X) has its read or write operation delayed until the transaction T′ that wrote the value of X (hence TS(T′) = write_TS(X)) has committed or aborted.
- To implement this algorithm, it is necessary to simulate the locking of an item X that has been written by transaction T′ until T′ is either committed or aborted.
- This algorithm does not cause deadlock, since T waits for T′ only if TS(T) > TS(T′).

Concurrency control based on timestamp: Thomas’s Write Rule

- A modification of the basic TO algorithm, known as Thomas’s write rule, does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for the write_item(X) operation as follows:
  1. If read_TS(X) > TS(T), then abort and roll back T and reject the operation.
  2. If write_TS(X) > TS(T), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of X. Thus, we must ignore the write_item(X) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).
  3. If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

Concurrency control based on timestamp: Thomas’s Write Rule

Time	T1	T2
t1	Begin transaction	
t2	Read (A)	
t3	A:= A+20	Begin transaction
t4		Read (A)
t5	Write (A)	
t6		A:= A+30
t7		Write (A)
t8		B:= 100
t9		Write (B)
t10		Commit
t11	Begin transaction	
t12	Read (A)	
t13	A:= A+20	
t14	Write (A)	
t15	Commit	

- T2 (t4) → dirty read problem.
- To get rid from the problem first T2 will execute, then T1 will restart (t11) [according to Thomas’s write rule 1]

Concurrency control based on timestamp: Thomas's Write Rule

Time	T1	T2
t1	Begin transaction	
t2	Read (A)	
t3	A:= A+10	
t4	Write (A)	
t5		Begin transaction
t6		Read (B)
t7		B:= B+100
t8		Write (B)
t9		Read (C)
t10		C:= C+200
t11		Write (C)
t12		Commit
t13	C:= 50	
t14	Write (C)	
t15	Commit	

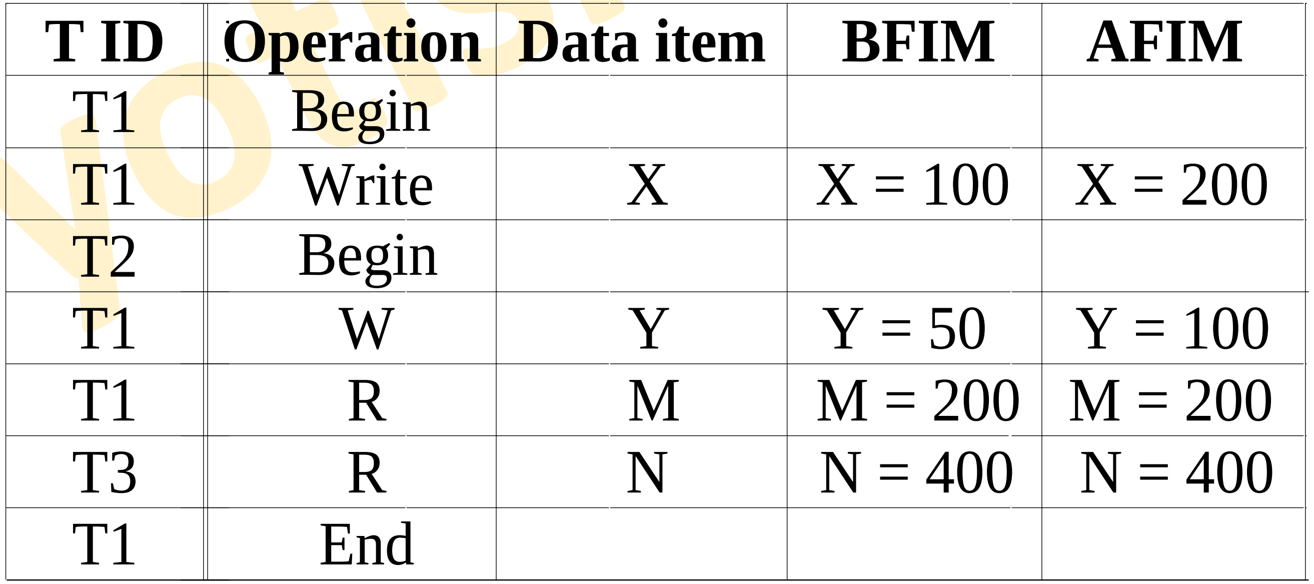
- T2 is younger than T1
- T2 first updates the value of C.
- Thus according to Thomas's write rule (2) the updated value of C done by T1 (at t13) will be cancelled out

Recovery concepts

- Recovery from transaction failures usually means that the database is restored to the most recent consistent state before the time of failure.
- To do this, the system must keep information about the changes that were applied to data items by the various transactions.
- This information is typically kept in the **system log**.
- A typical strategy for recovery may be summarized informally as follows:
  - If there is extensive damage to a wide portion of the database due to catastrophic failure, such as a disk crash, the recovery method restores a past copy of the database that was backed up to archival storage (typically tape or other large capacity offline storage media) and reconstructs a more current state by reapplying or redoing the operations of committed transactions from the backed-up log, up to the time of failure.
  - When the database on disk is not physically damaged, and a noncatastrophic failure has occurred, the recovery strategy is to identify any changes that may cause an inconsistency in the database.

Recovery concepts

- For recovery from any type of failure data values prior to modification (BFIM - BeFore Image) and the new value after modification (AFIM – AFter Image) are required.
- These values and other information is stored in a sequential file called system log.



T ID	Operation	Data item	BFIM	AFIM
T1	Begin			
T1	Write	X	X = 100	X = 200
T2	Begin			
T1	W	Y	Y = 50	Y = 100
T1	R	M	M = 200	M = 200
T3	R	N	N = 400	N = 400
T1	End			

Recovery concepts

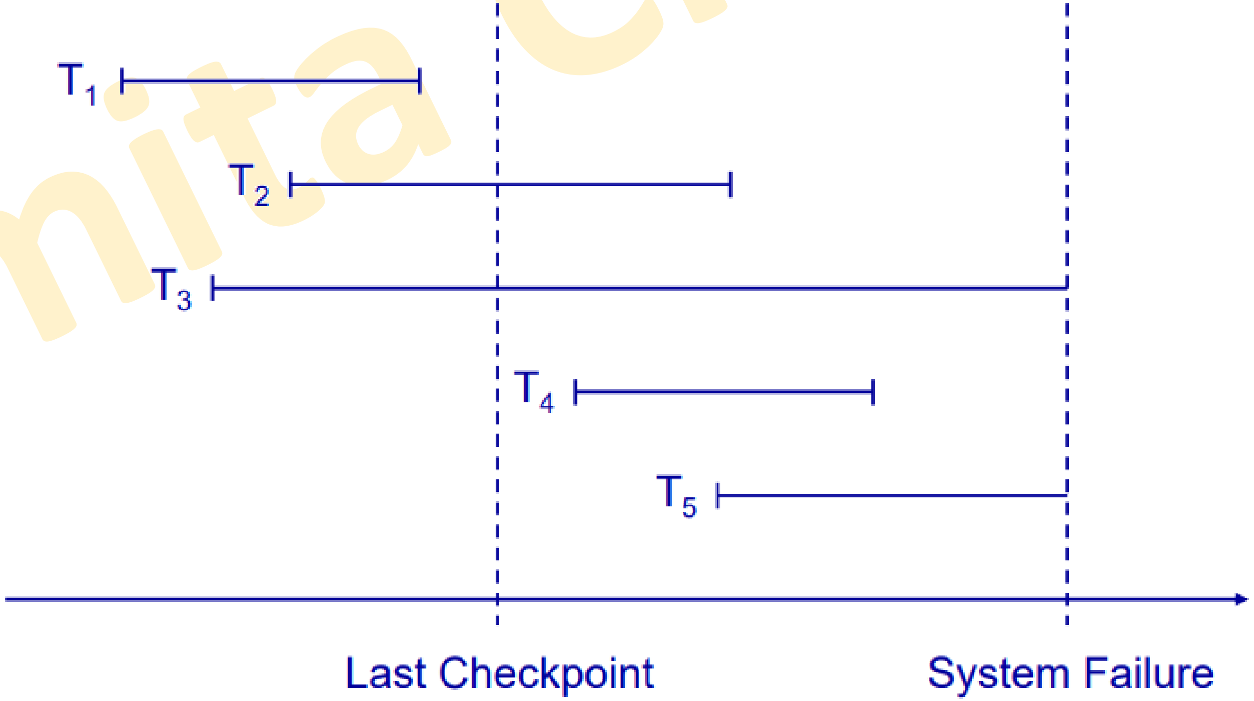
- To maintain atomicity, a transaction’s operations are redone or undone.
    - **Undo**: Restore all BFIMs on to disk (Remove all AFIMs).
    - **Redo**: Restore all AFIMs on to disk.
- Database recovery is achieved either by performing only **Undos** or only **Redos** or by a combination of the two.

Recovery concepts: Checkpointing

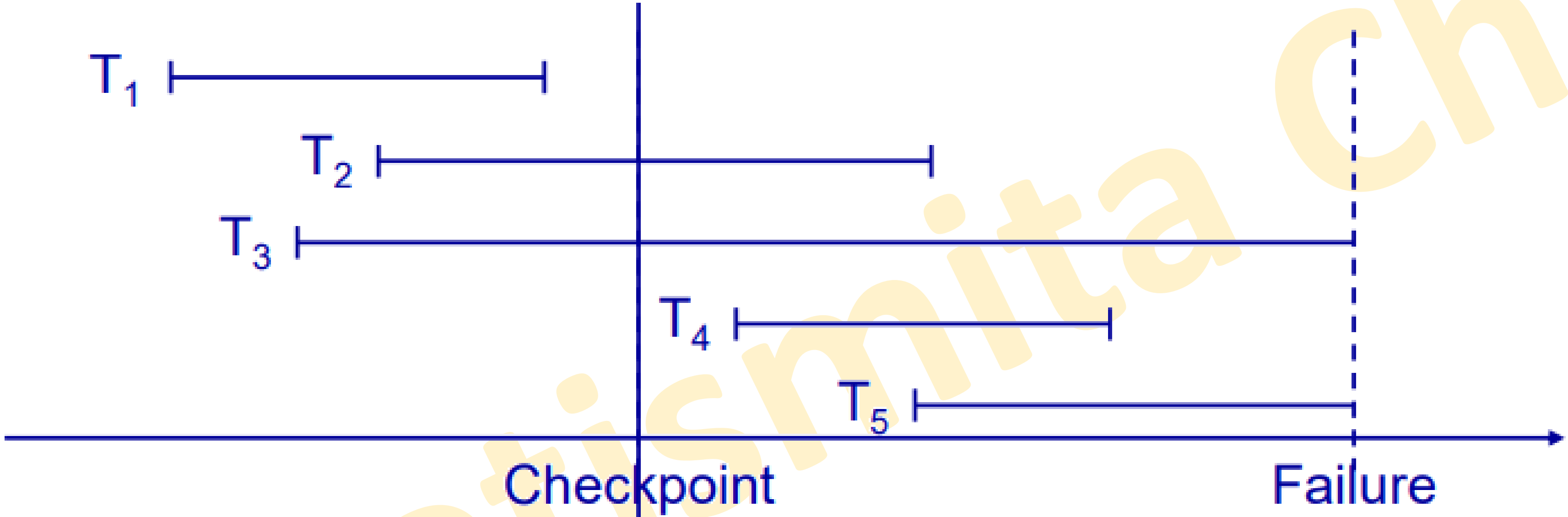
- From time to time the database flushes its buffer to database disk to minimize the task of recovery.
- Following steps defines a **checkpoint** operation:
    1. Suspend execution of transactions temporarily.
    2. Force write modified buffer data to disk (unless a no-UNDO).
    3. Write a [checkpoint] record to the log, save the log to disk.
    4. Resume normal transaction execution.
- During recovery redo or undo is required to transactions appearing after [checkpoint] record.

Recovery concepts: Checkpointing

- Any transaction that was running at the time of failure needs to be undone and restarted.
- Any transactions that committed since the last checkpoint need to be redone.
- Transactions of type T1 need no recovery.
- Transactions of type T3 or T5 need to be undone and restarted.
- Transactions of type T2 or T4 need to be redone.



Recovery concepts: Checkpointing



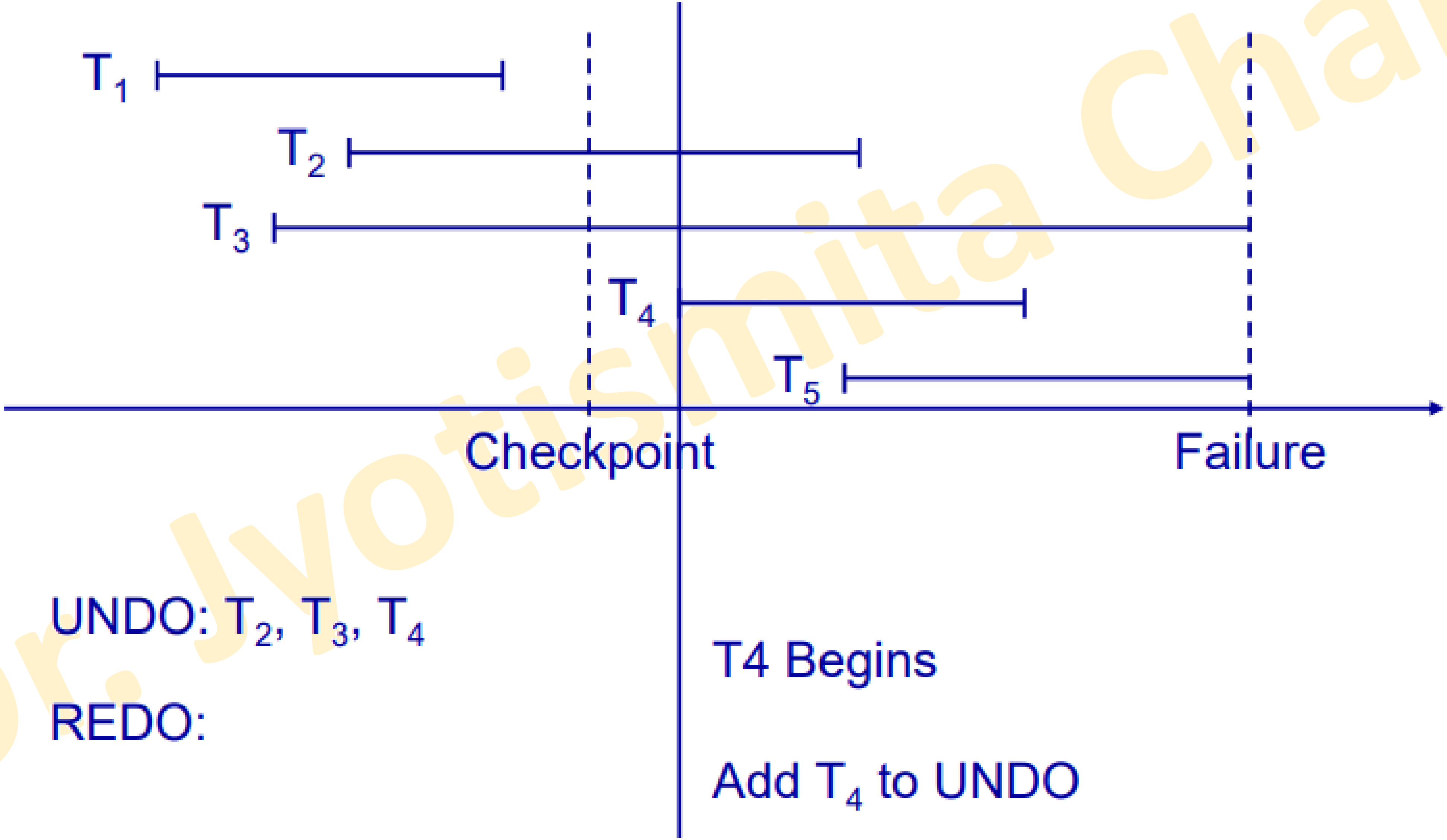
UNDO: T_2, T_3

REDO:

Last Checkpoint

Active transactions: T_2, T_3

Recovery concepts: Checkpointing



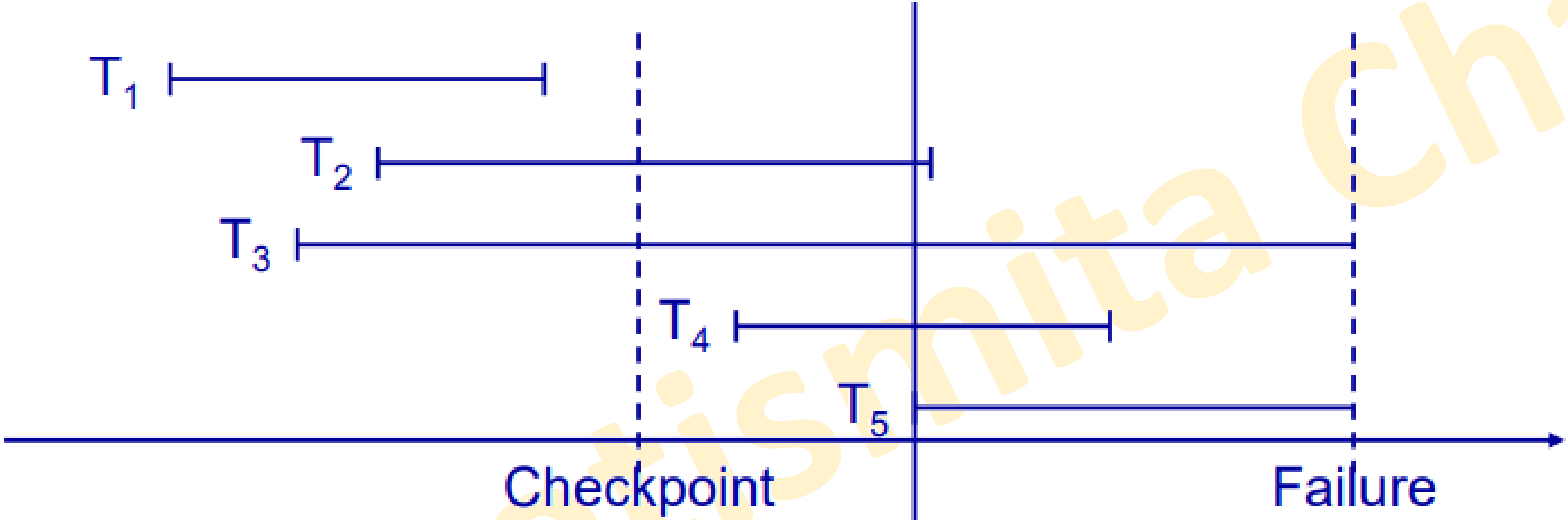
UNDO: T_2, T_3, T_4

REDO:

T4 Begins

Add T_4 to UNDO

Recovery concepts: Checkpointing



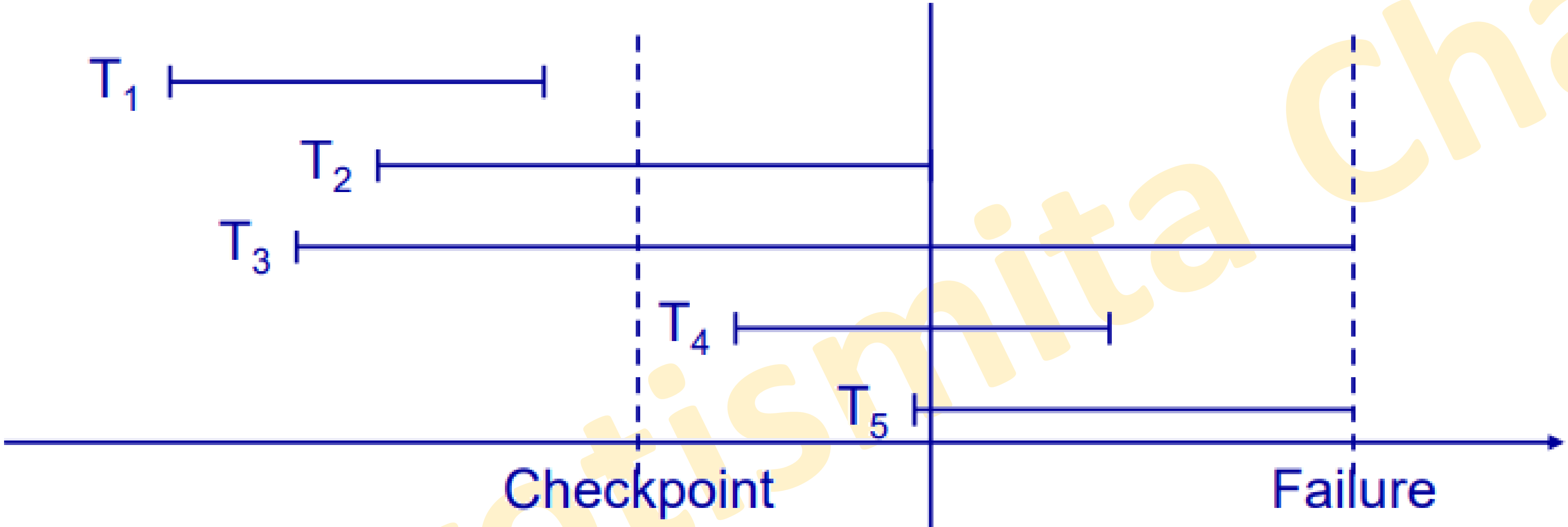
UNDO: T_2, T_3, T_4, T_5

REDO:

T_5 begins

Add T_5 to UNDO

Recovery concepts: Checkpointing



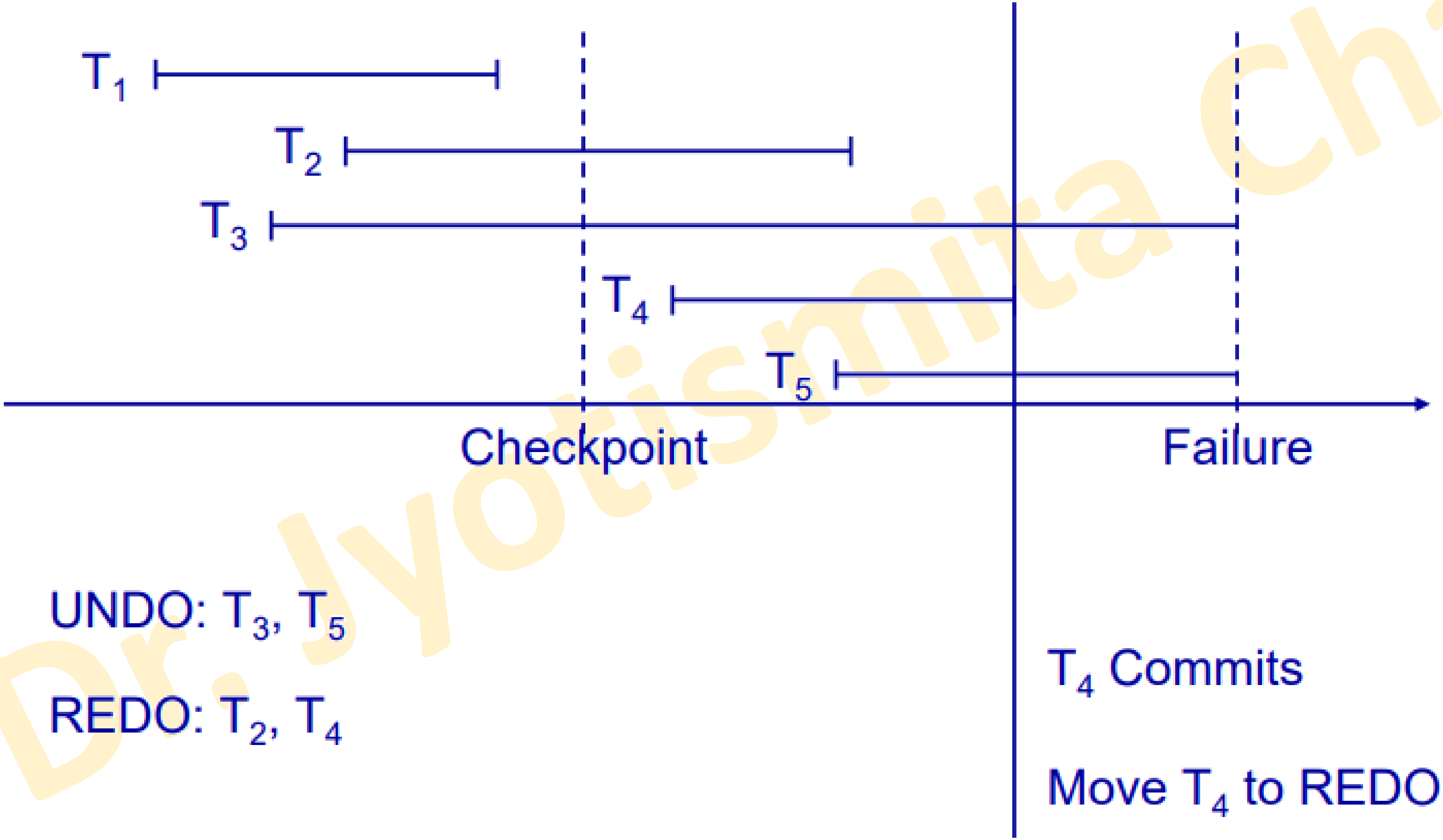
UNDO: T_3, T_4, T_5

REDO: T_2

T_2 Commits

Move T_2 to REDO

Recovery concepts: Checkpointing



UNDO: T_3, T_5

REDO: T_2, T_4

T_4 Commits

Move T_4 to REDO

Recovery based on deferred update

- Conceptually, we can distinguish two main policies for recovery from noncatastrophic transaction failures: **deferred update** and **immediate update**.
- The **deferred update** techniques do not physically update the database on disk until after a transaction commits; then the updates are recorded in the database.
- Before reaching commit, all transaction updates are recorded in the local transaction workspace or in the main memory buffers that the DBMS maintains.
- Before commit, the updates are recorded persistently in the log file on disk, and then after commit, the updates are written to the database from the main memory buffers.
- If a transaction fails before reaching its commit point, it will not have changed the database on disk in any way, so UNDO is not needed.
- It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their effect may not yet have been recorded in the database on disk.
- Hence, deferred update is also known as the **NO-UNDO/REDO** algorithm.

Recovery based on deferred update

- Two tables are required for implementing this protocol.
  - **Active table**: All active transactions are entered in this table.
  - **Commit table**: Transactions to be committed are entered in this table.
- These tables are filled by scanning through the log from the last checkpoint during recovery.
- During recovery
  - All transactions of the **commit** table are redone in the order they were written to log, and
  - All transactions of **active** tables are ignored.

Recovery based on deferred update: Single Transaction

- A = 100, B = 200

New value

T1
Read (A)
A:= A+100
Write (A)
Read (B)
B:= B+200
Write (B)
Commit

System Log
<start_transaction, T1>
<write_item, T1, A, 200>
<write_item, T1, B, 400>
<Commit, T1>

- After commit, if T1 fails → REDO
- After Recovery of T1: Updated value A = 200, B = 400

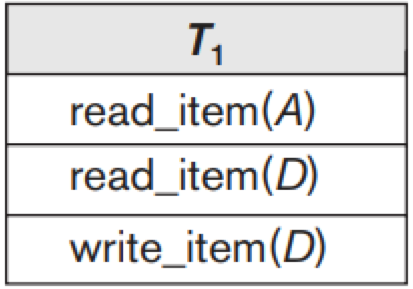
T1
Read (A)
A:= A+100
Write (A)
Read (B)
B:= B+200
Write (B)

System Log
<start_transaction, T1>
<write_item, T1, A, 200>
<write_item, T1, B, 400>

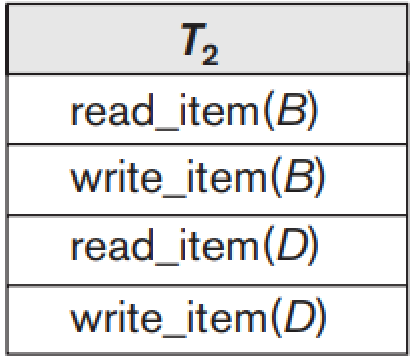
- If T1 fails before Commit → no REDO
- After recovery of T1: Value of A = 100, B = 200

- After commit, the data stored in HD will be updated

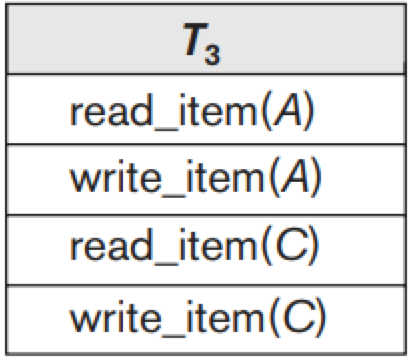
Recovery based on deferred update: Multiple Transaction



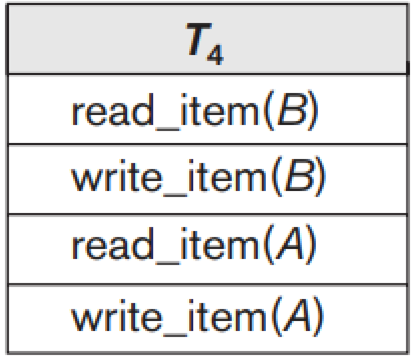
T_1
read_item(A)
read_item(D)
write_item(D)



T_2
read_item(B)
write_item(B)
read_item(D)
write_item(D)

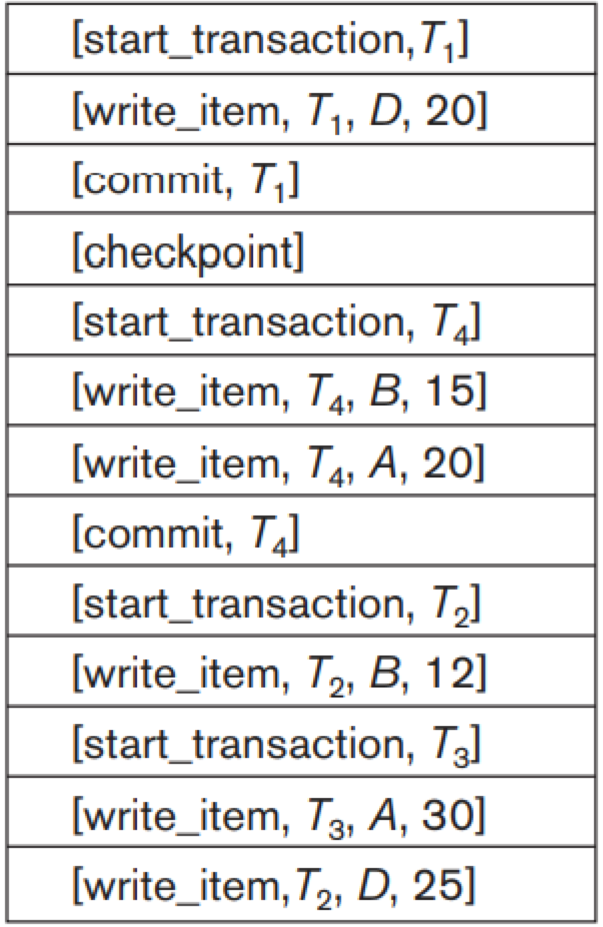


T_3
read_item(A)
write_item(A)
read_item(C)
write_item(C)



T_4
read_item(B)
write_item(B)
read_item(A)
write_item(A)

The READ and WRITE operations of four transactions.



[start_transaction, T_1]
[write_item, T_1, D, 20]
[commit, T_1]
[checkpoint]
[start_transaction, T_4]
[write_item, T_4, B, 15]
[write_item, T_4, A, 20]
[commit, T_4]
[start_transaction, T_2]
[write_item, T_2, B, 12]
[start_transaction, T_3]
[write_item, T_3, A, 30]
[write_item, T_2, D, 25]

← System crash

System log at the point of crash

T2 and T3 are ignored because they did not reach their commit points. T4 is redone because its commit point is after the last system checkpoint.

Recovery techniques based on immediate update

- In the **immediate update** techniques, the database may be updated by some operations of a transaction before the transaction reaches its commit point.
- However, these operations must also be recorded in the log on disk by force-writing before they are applied to the database on disk, making recovery still possible.
- If a transaction fails after recording some changes in the database on disk but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back.
- In the general case of immediate update, both undo and redo may be required during recovery. This technique, known as the **UNDO/REDO** algorithm, requires both operations during recovery and is used most often in practice.
- A variation of the algorithm where all updates are required to be recorded in the database on disk before a transaction commits requires undo only, so it is known as the **UNDO/NO-REDO** algorithm.

Recovery techniques based on immediate update

- Theoretically, we can distinguish two main categories of immediate update algorithms:
  1. If the recovery technique ensures that all updates of a transaction are recorded in the database on disk before the transaction commits, there is never a need to REDO any operations of committed transactions. This is called the UNDO/NO-REDO recovery algorithm. In this method, all updates by a transaction must be recorded on disk before the transaction commits, so that REDO is never needed.
  2. If the transaction is allowed to commit before all its changes are written to the database, we have the most general case, known as the UNDO/REDO recovery algorithm. This is also the most complex technique, but the most commonly used in practice.

Recovery techniques based on immediate update

- A = 100, B = 200

Old value

New value

Before commit (immediate after writing to main memory), the data stored in HD will be updated

T1
Read (A)
A:= A+100
Write (A)
Read (B)
B:= B+200
Write (B)
Commit

System Log
<start_transaction, T1>
<write_item, T1, A, 100, 200>
<write_item, T1, B, 200, 400>
<Commit, T1>

- After commit, if T1 fails → REDO
- After Recovery of T1: Updated value A = 200, B = 400

T1
Read (A)
A:= A+100
Write (A)
Read (B)
B:= B+200
Write (B)

System Log
<start_transaction, T1>
<write_item, T1, A, 100, 200>
<write_item, T1, B, 200, 400>

- If T1 fails before Commit → UNDO
- After recovery of T1 (old value is fetched from the log): Value of A = 100, B = 200

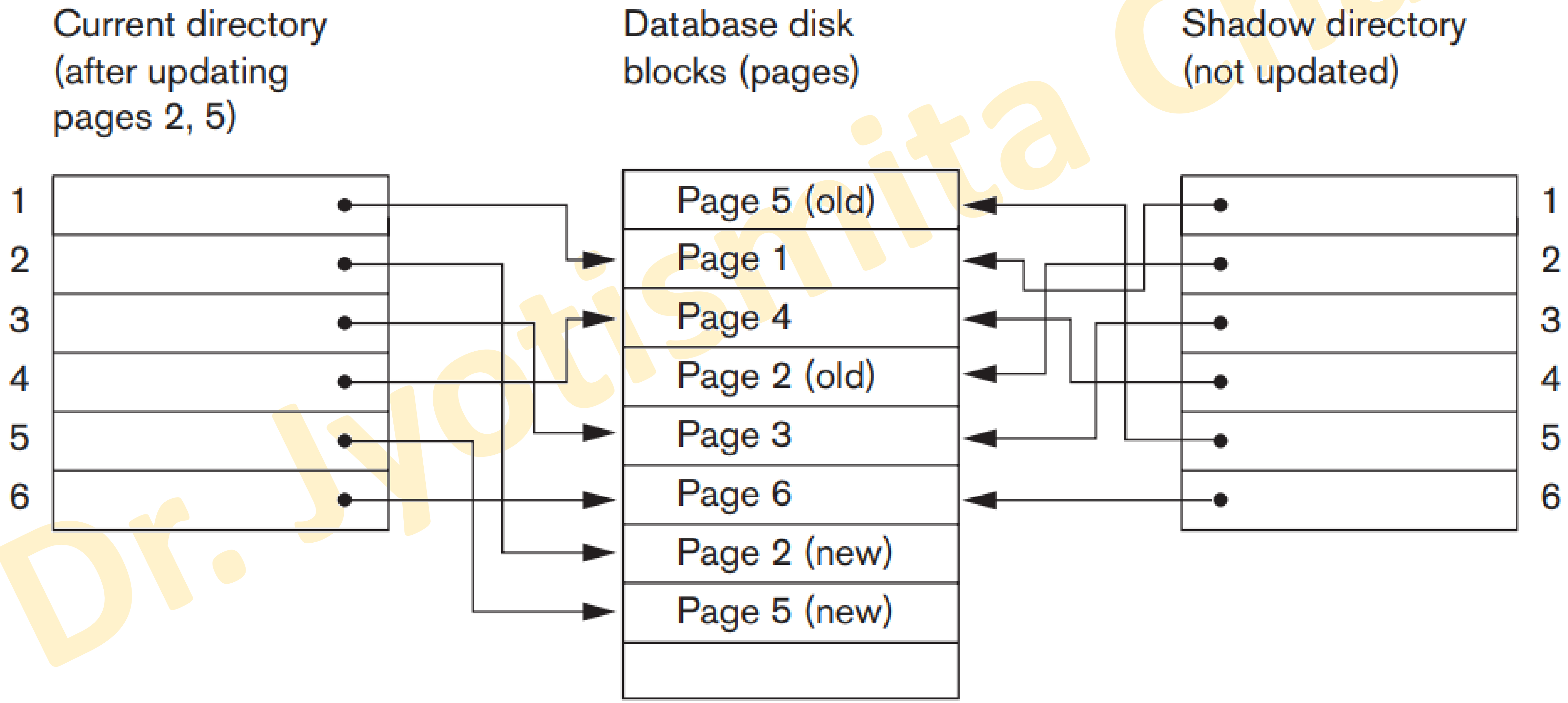
Shadow paging

- This recovery scheme does not require the use of a log in a single-user environment.
- In a multiuser environment, a log may be needed for the concurrency control method.
- Shadow paging considers the database to be made up of a number of fixedsize disk pages (or disk blocks)—say, n—for recovery purposes.
- A directory with n entries is constructed, where the i-th entry points to the i-th database page on disk.
- The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it.
- When a transaction begins executing, the current directory—whose entries point to the most recent or current database pages on disk—is copied into a shadow directory.
- The shadow directory is then saved on disk while the current directory is used by the transaction.

Shadow paging

- During transaction execution, the shadow directory is never modified.
- When a write_item operation is performed, a new copy of the modified database page is created, but the old copy of that page is not overwritten.
- Instead, the new page is written elsewhere—on some previously unused disk block.
- The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block.
- For pages updated by the transaction, two versions are kept.
- The old version is referenced by the shadow directory and the new version by the current directory.

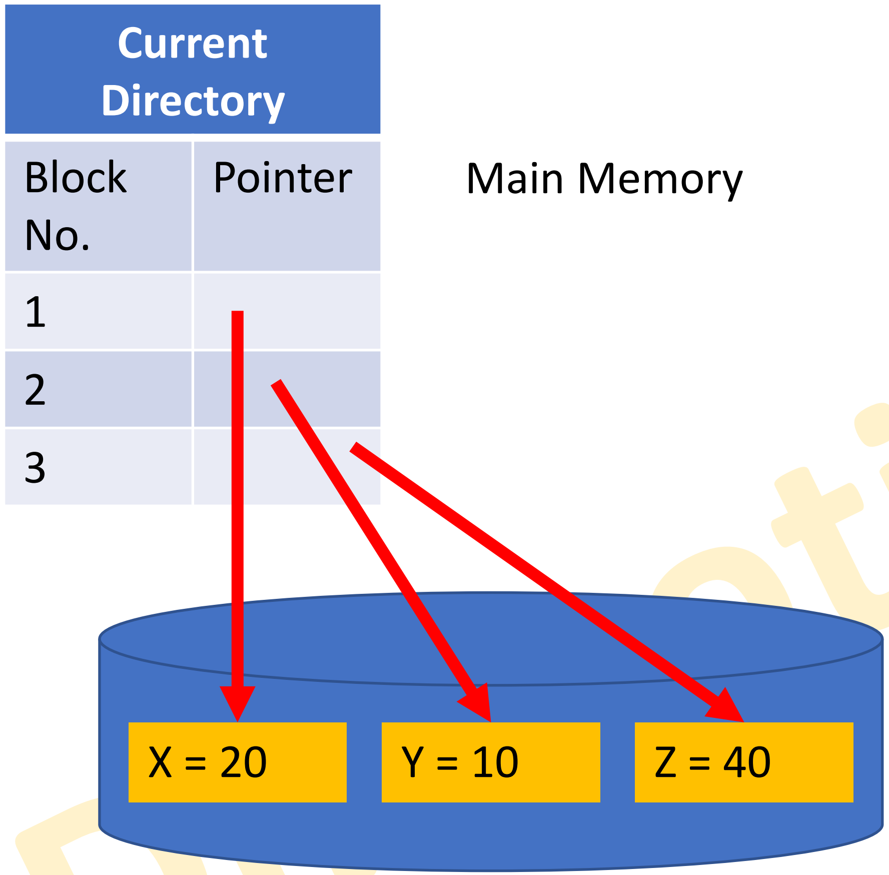
Shadow paging



Shadow paging

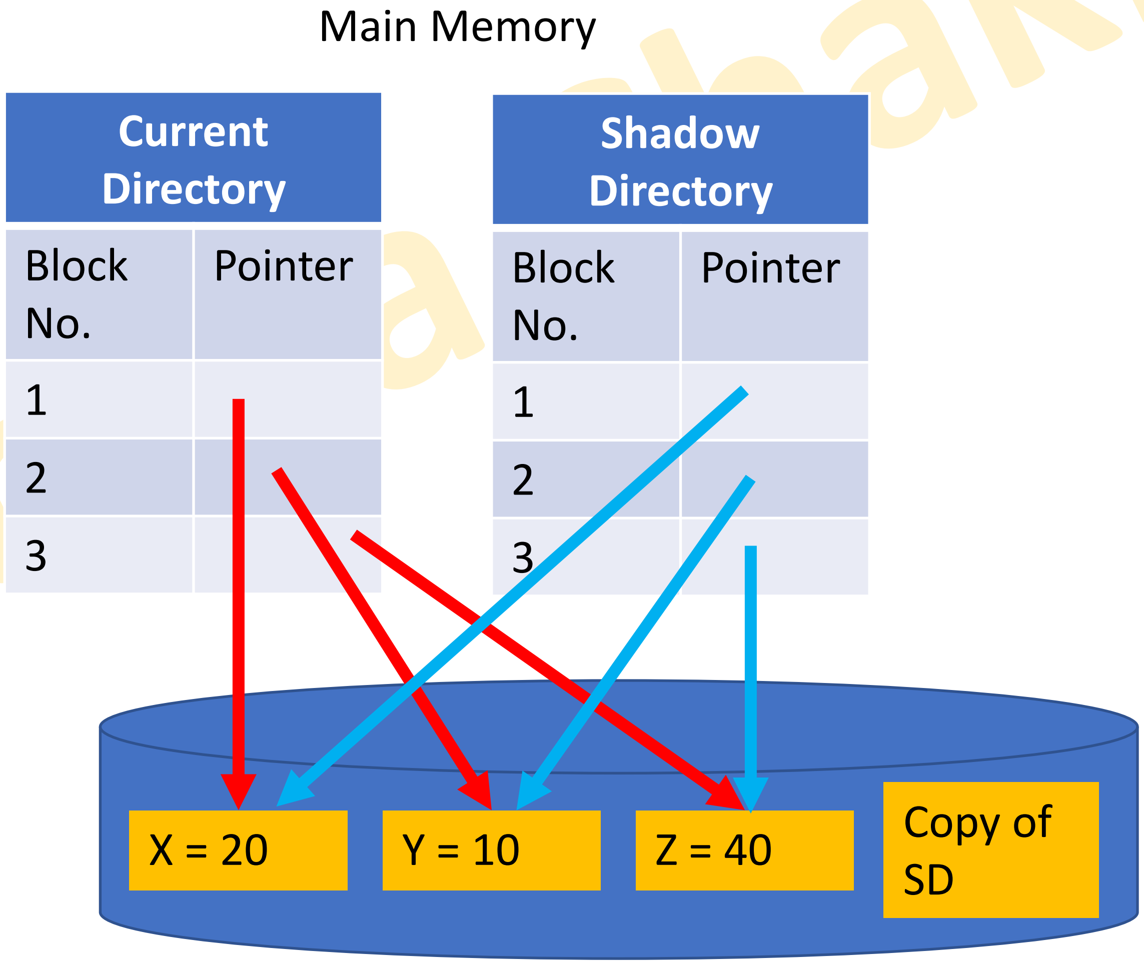
- To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory.
- The state of the database before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory.
- The database thus is returned to its state prior to the transaction that was executing when the crash occurred, and any modified pages are discarded.
- Committing a transaction corresponds to discarding the previous shadow directory.
- Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery.

Shadow paging: Example



Main Memory

Current Directory	
Block No.	Pointer
1	
2	
3	

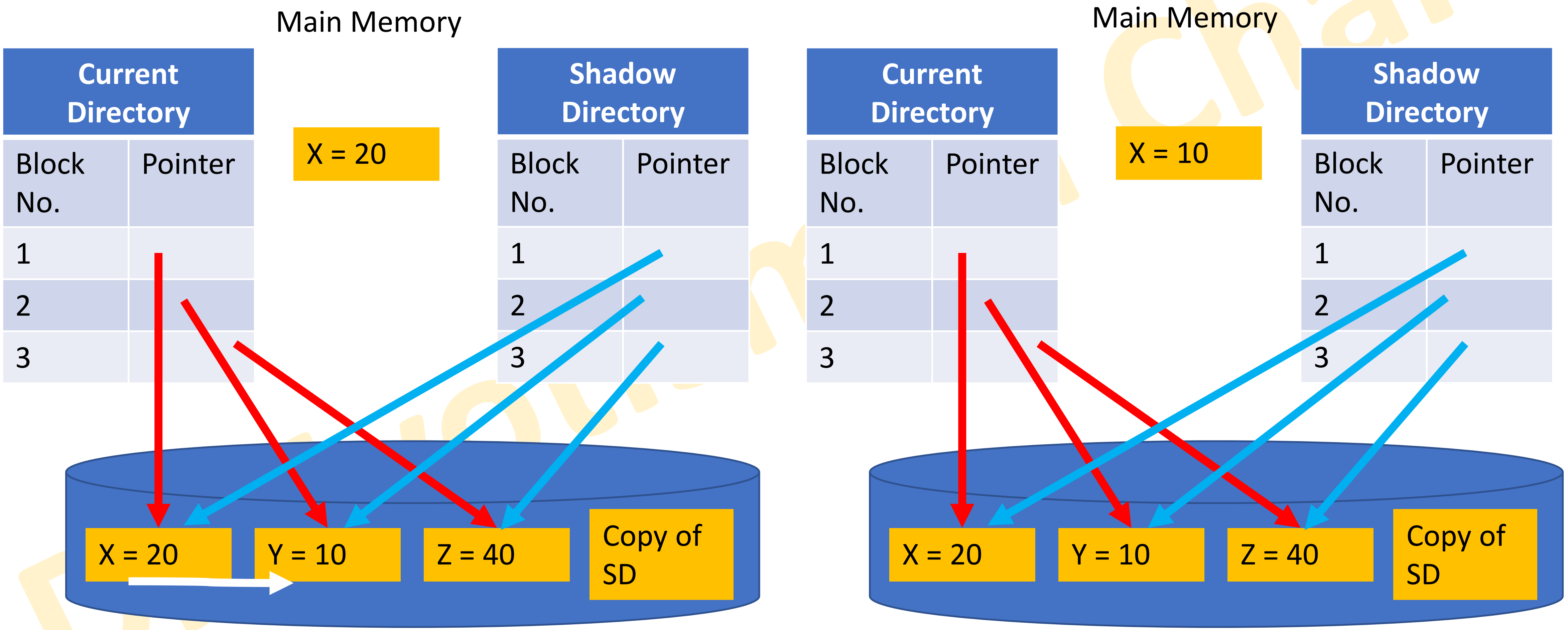


Main Memory

Current Directory	
Block No.	Pointer
1	
2	
3	

Shadow Directory	
Block No.	Pointer
1	
2	
3	

Shadow paging: Example



Main Memory

Current Directory	
Block No.	Pointer
1	
2	
3	

X = 20

Shadow Directory	
Block No.	Pointer
1	
2	
3	

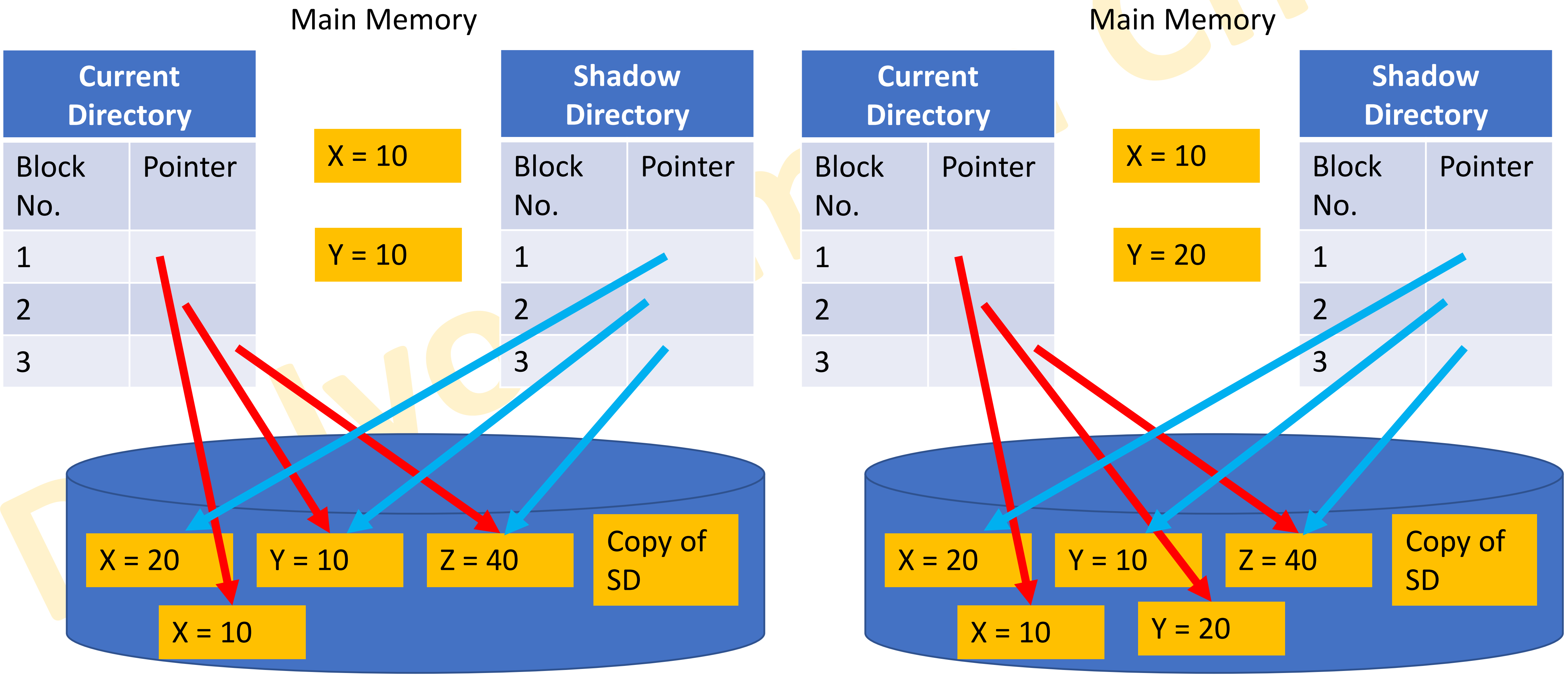
Main Memory

Current Directory	
Block No.	Pointer
1	
2	
3	

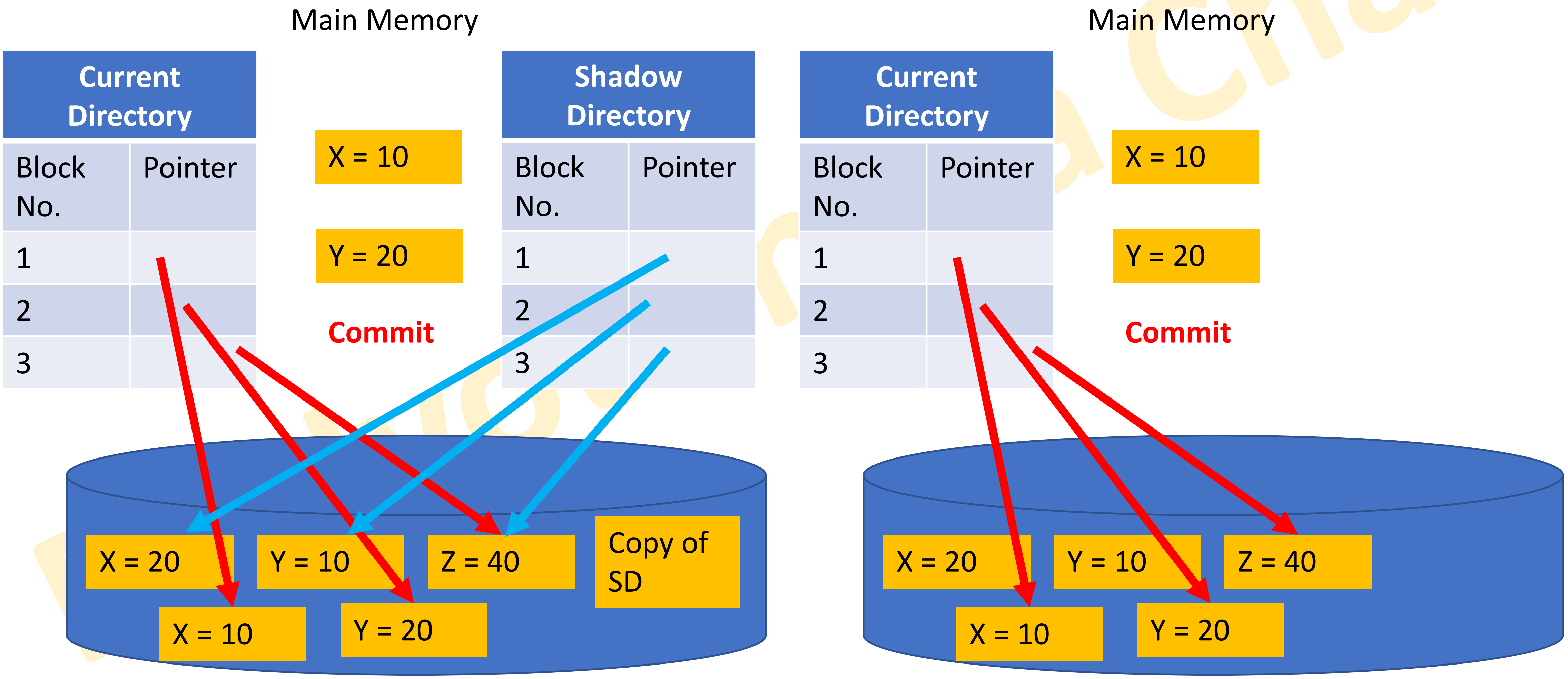
X = 10

Shadow Directory	
Block No.	Pointer
1	
2	
3	

Shadow paging: Example



Shadow paging: Example



Main Memory

Current Directory	
Block No.	Pointer
1	
2	
3	

X = 10

Y = 20

Commit

Shadow Directory	
Block No.	Pointer
1	
2	
3	

Main Memory

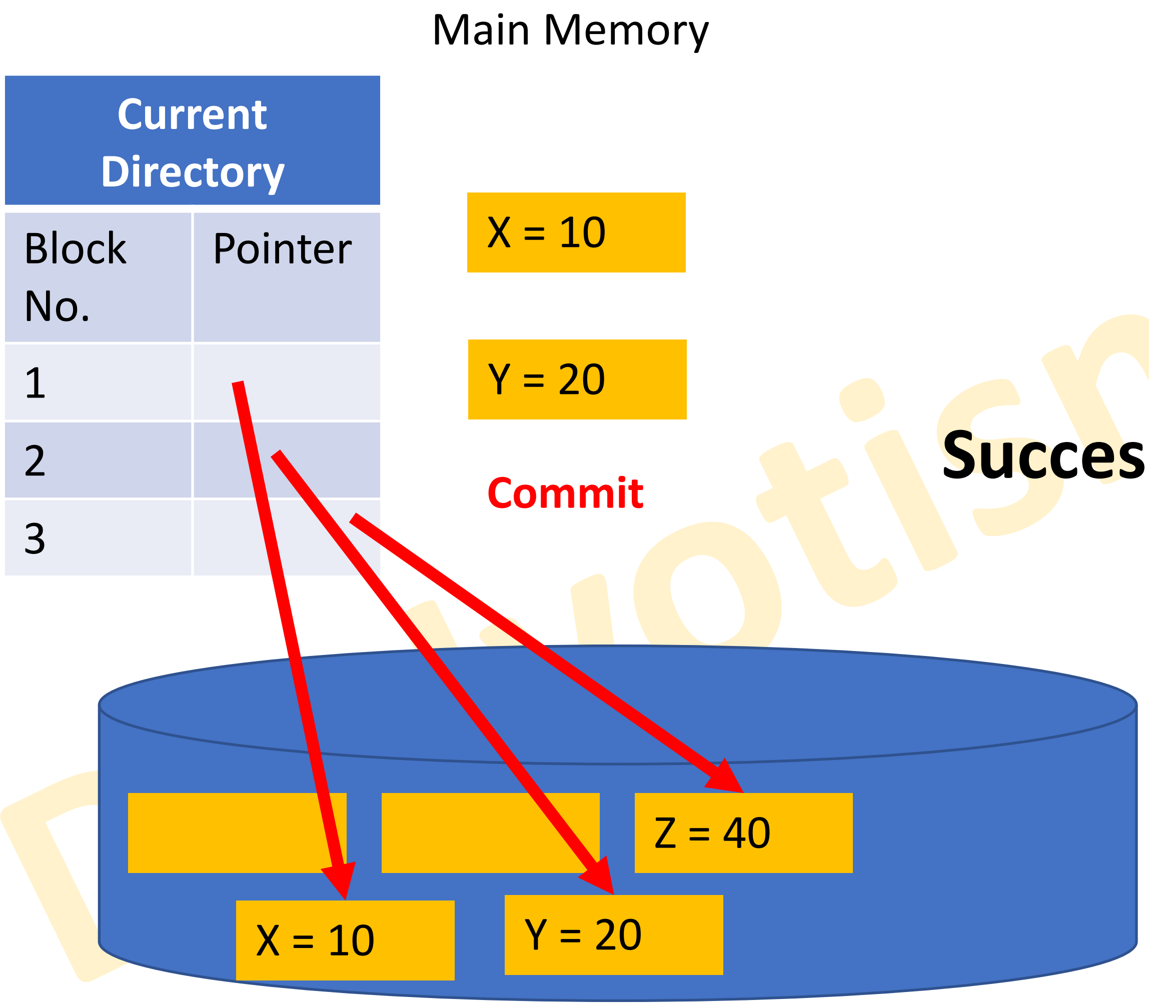
Current Directory	
Block No.	Pointer
1	
2	
3	

X = 10

Y = 20

Commit

Shadow paging: Example



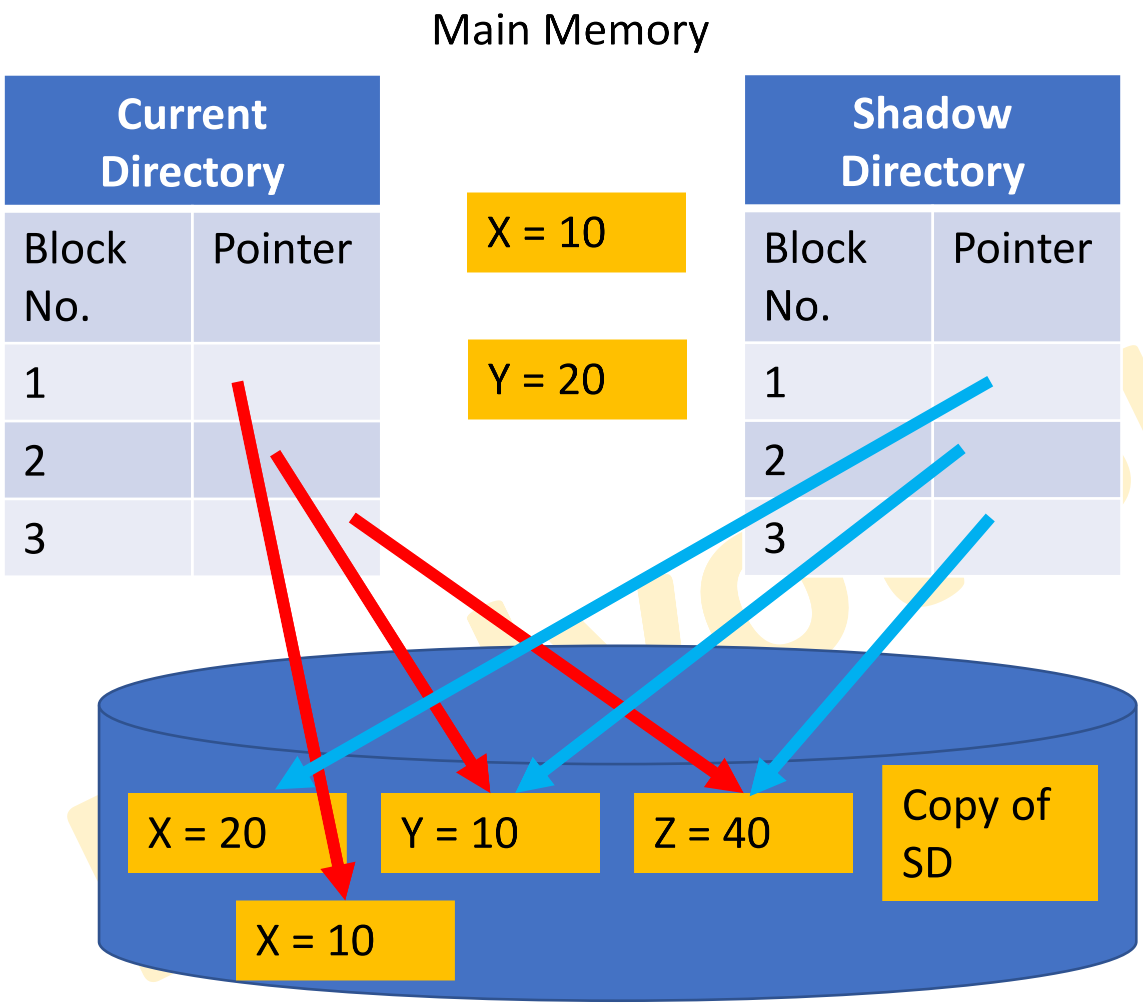
Main Memory

Current Directory	
Block No.	Pointer
1	
2	
3	

Commit

Successful Transaction

Shadow paging: Example



Main Memory

Current Directory	
Block No.	Pointer
1	
2	
3	

X = 10

Y = 20

Shadow Directory	
Block No.	Pointer
1	
2	
3	

X = 20

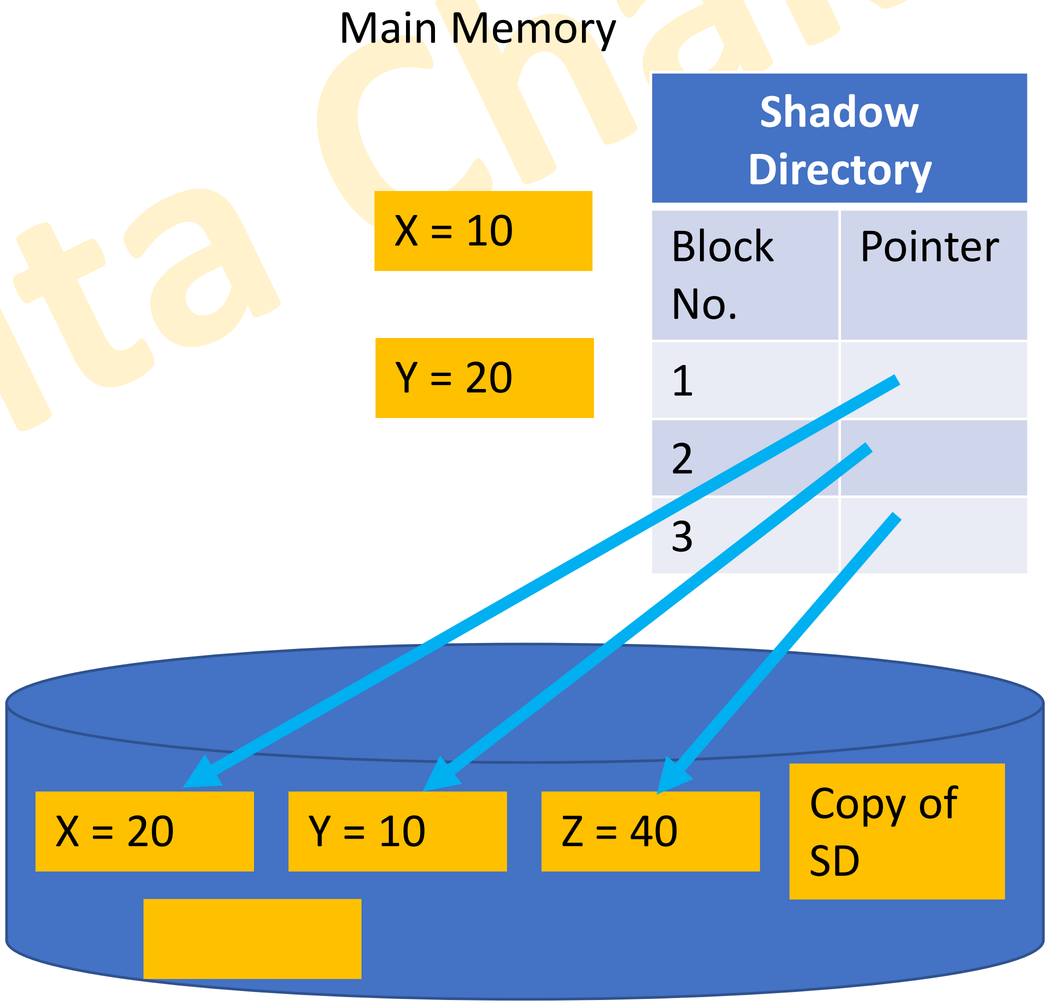
Y = 10

Z = 40

Copy of SD

X = 10

Transaction Fails



Main Memory

X = 10

Y = 20

Shadow Directory	
Block No.	Pointer
1	
2	
3	

X = 20

Y = 10

Z = 40

Copy of SD

No Undo No Redo