



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David



CONCURRENCY CONTROL

17-Nov-2023

Week 7: Database Systems (DBS)
Dr Suresh Shanmugasundaram

So far...



1. Conceptual Data Model using E-R Diagrams
2. Converting ERD to Relational Model
3. Normalisation of the Relational Model
4. Transaction Management and its properties
5. Concurrency Control and its problems

Learning Objectives



1. Serialisability & Recoverability
2. Locking
3. Deadlocks

Recap - Quiz



<https://forms.office.com/e/URW1akRAan>

Serialisability

- To find non-serial schedules
- To produce a database state that could be produced by a serial execution.

Transaction T1	Transaction T2
R(A)	
W(A)	
R(B)	
W(B)	
commit	
	R(A)
	W(B)
	commit

Transaction T1	Transaction T2
R1(A)	
W1(A)	
	R2(A)
	W2(A)
R1(B)	
W1(B)	
	R2(B)
	W2(B)

Serialisability – Example Schedules

Schedule3

$A = £1000$, $B = £2000$: $A + B = £3000$

T_1	T_2
$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$	$\text{read}(A)$ $\text{temp} := A * 0.1$ $A := A - \text{temp}$ $\text{write}(A)$
$\text{read}(B)$ $B := B + 50$ $\text{write}(B)$ commit	$\text{read}(B)$ $B := B + \text{temp}$ $\text{write}(B)$ commit

Schedule1

$A = £1000$, $B = £2000$, $A + B = £3000$

T_1	T_2
$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$ $\text{read}(B)$ $B := B + 50$ $\text{write}(B)$ commit	$\text{read}(A)$ $\text{temp} := A * 0.1$ $A := A - \text{temp}$ $\text{write}(A)$ $\text{read}(B)$ $B := B + \text{temp}$ $\text{write}(B)$ commit

Serialisability – Example Schedules

Schedule3

A=£1000, B=£2000: A+B=£3000

T_1	T_2
read(A) $A := A - 50$ write(A)	read(A) $temp := A * 0.1$ $A := A - temp$ write(A)
read(B) $B := B + 50$ write(B) commit	read(B) $B := B + temp$ write(B) commit

A=£855, B=£2145: A+B=£3000

Schedule1

A=£1000, B=£2000, A+B=£3000

T_1	T_2
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B) commit	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B) commit

Serialisability – Example Schedules

Schedule3

A=£1000, B=£2000: A+B=£3000

T_1	T_2
read(A) $A := A - 50$ write(A)	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A)
read(B) $B := B + 50$ write(B) commit	
	read(B) $B := B + temp$ write(B) commit

A=£855, B=£2145: A+B=£3000

Schedule1

A=£1000, B=£2000, A+B=£3000

T_1	T_2
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B) commit	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B) commit

A=£855, B=£2145: A+B=£3000

Serialisability – Example Schedules

Schedule3

A=£1000, B=£2000: A+B=£3000

T_1	T_2
read(A) $A := A - 50$ write(A)	read(A) $temp := A * 0.1$ $A := A - temp$ write(A)
read(B) $B := B + 50$ write(B) commit	read(B) $B := B + temp$ write(B) commit

A=£855, B=£2145: A+B=£3000

Schedule1

A=£1000, B=£2000, A+B=£3000

T_1	T_2
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B) commit	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B) commit

A=£855, B=£2145: A+B=£3000

equivalent

Serializable Vs Serial Schedule

T_1	T_2	T_1	T_2
$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$	$\text{read}(A)$ $\text{temp} := A * 0.1$ $A := A - \text{temp}$ $\text{write}(A)$	$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$ $\text{read}(B)$ $B := B + 50$ $\text{write}(B)$ commit	$\text{read}(A)$ $\text{temp} := A * 0.1$ $A := A - \text{temp}$ $\text{write}(A)$ $\text{read}(B)$ $B := B + \text{temp}$ $\text{write}(B)$ commit

Concurrency is allowed.

Thus, multiple transactions can execute concurrently.

Serializable schedules improve both resource utilization and CPU throughput.

Serializable Schedules are always better than serial schedules.

No concurrency is allowed.

Thus, all the transactions necessarily execute serially one after the other.

Serial schedules lead to less resource utilization and CPU throughput.

Serial Schedules are less efficient as compared to serializable schedules.

Types of Serializability



1. Conflict Serializability
2. View Serializability

Conflict Serialisability

- it can be transformed into a serial schedule by swapping non-conflicting operations.

		Transaction T1	Transaction T2
No conflict	[	Read(A)	
		Write(A)	
	]		Read(A)
			Write(A)
	[	Read(B)	
		Write(B)	
	]		Read(B)
			Write(B)

No conflict

No conflict

Conflict Serialisability

- it can be transformed into a serial schedule by swapping non-conflicting operations.

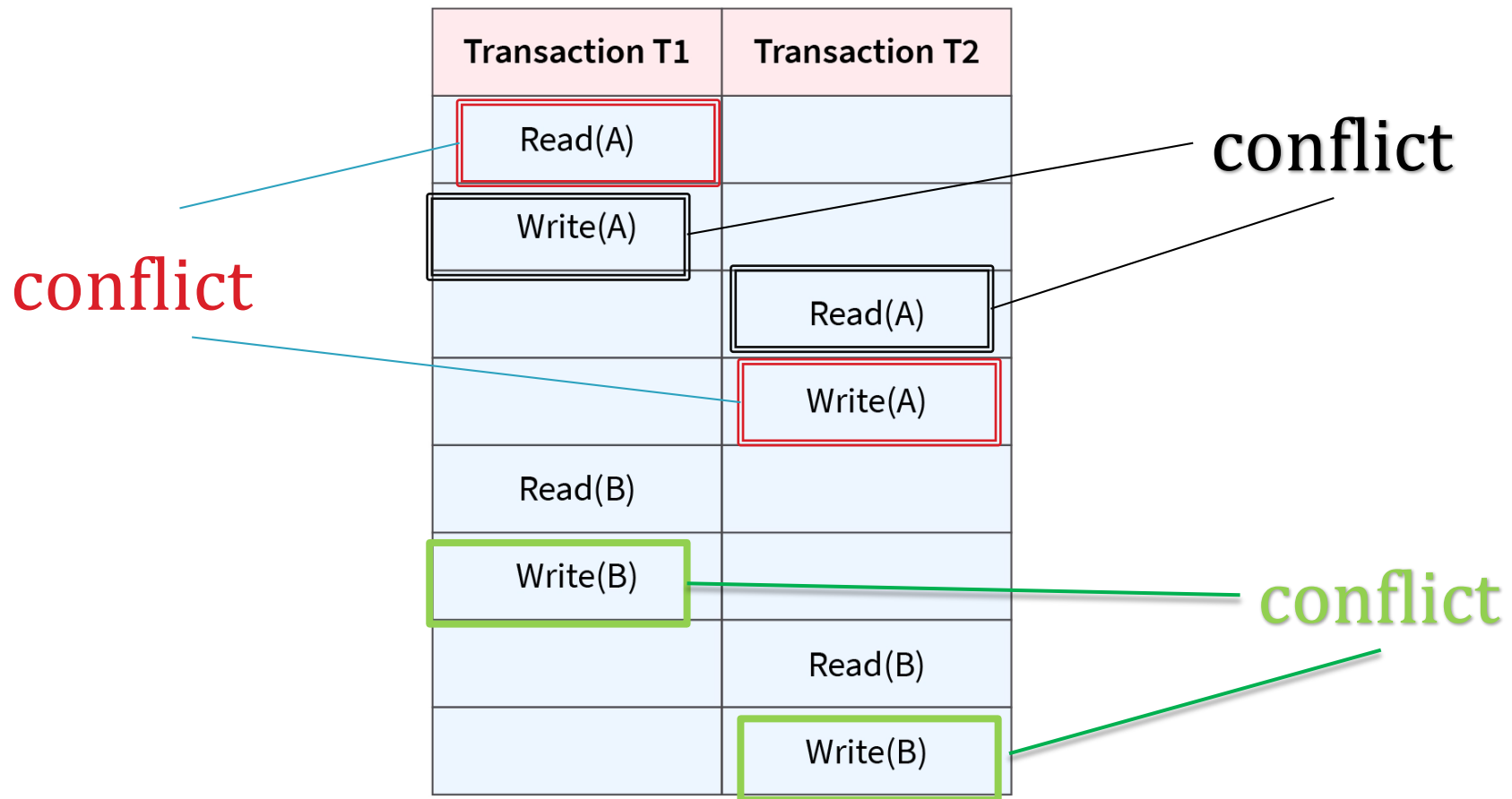
		Transaction T1	Transaction T2
No conflict		Read(A)	
		Write(A)	
			Read(A)
			Write(A)
		Read(B)	
		Write(B)	
			Read(B)
			Write(B)

No conflict

No conflict

Conflict Serialisability

- it can be transformed into a serial schedule by swapping non-conflicting operations.



Conflict Serialisability

- it can be transformed into a serial schedule by swapping non-conflicting operations.

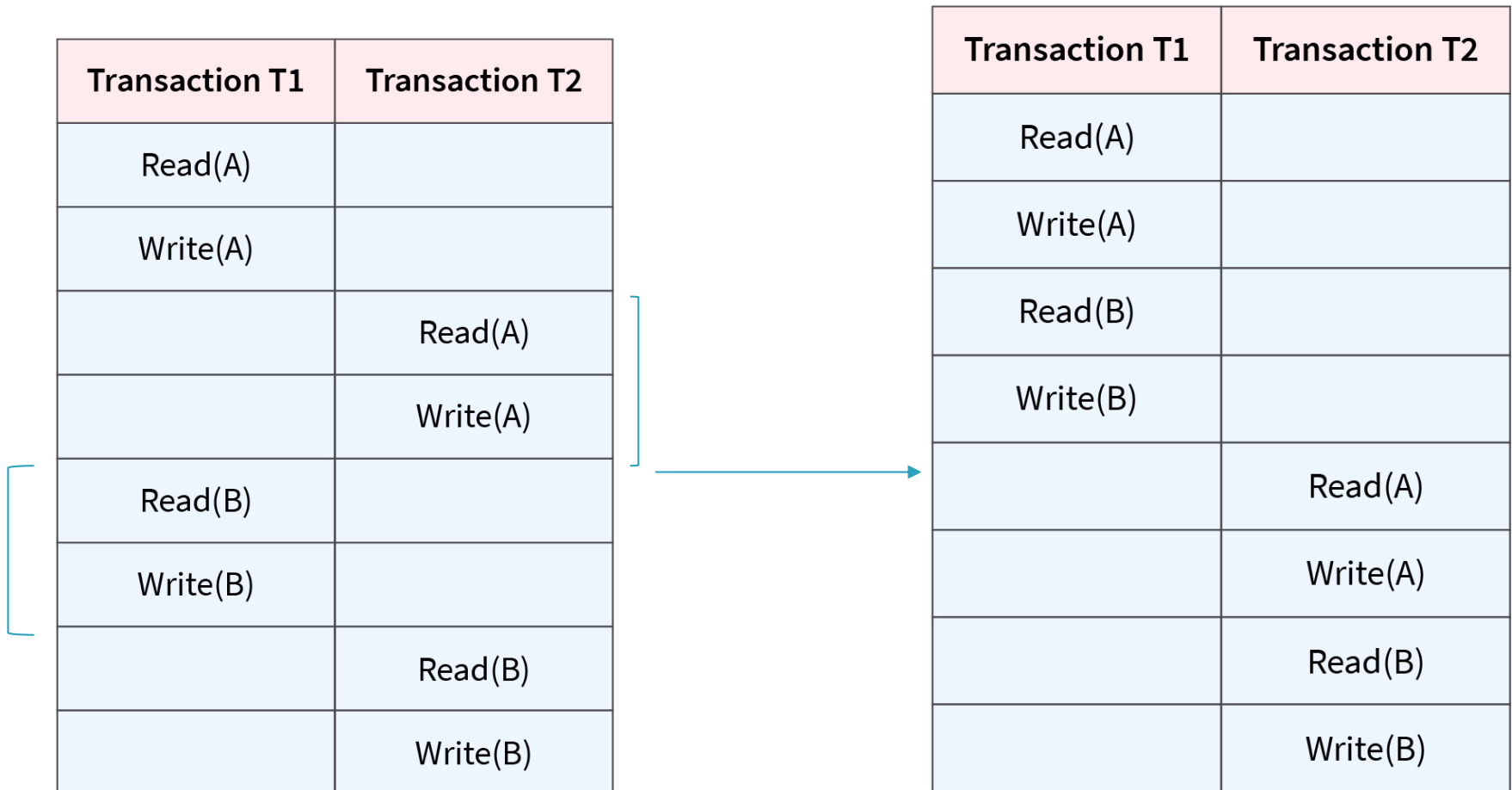
		Transaction T1	Transaction T2
No conflict	[	Read(A)	
		Write(A)	
	]		Read(A)
			Write(A)
	[	Read(B)	
		Write(B)	
	]		Read(B)
			Write(B)

No conflict

No conflict

Conflict Serialisability

- it can be transformed into a serial schedule by swapping non-conflicting operations.



View Serialisability

- If a schedule is conflict serializable, then it will be view serializable

Non-Serial		Serial	
S1		S2	
Transaction T1	Transaction T2	Transaction T1	Transaction T2
R(X)		R(X)	
W(X)		W(X)	
	R(X)	R(Y)	
	W(X)	W(Y)	
R(Y)			R(X)
W(Y)			W(X)
	R(Y)		R(Y)
	W(Y)		W(Y)

View Serialisability

- If a schedule is conflict serializable, then it will be view serializable

Non-Serial		Serial	
S1		S2	
Transaction T1	Transaction T2	Transaction T1	Transaction T2
R(X)		R(X)	
W(X)		W(X)	
	R(X)	R(Y)	
	W(X)	W(Y)	
R(Y)			R(X)
W(Y)			W(X)
	R(Y)		R(Y)
	W(Y)		W(Y)

Accounts:

AccountID	Balance
-----	-----
A101	1000
A102	1500

Transactions:

TransactionID	AccountID	Amount
-----	-----	-----
T1	A101	-200
T2	A102	+200

View Serialisability

- If a schedule is conflict serializable, then it will be view serializable

Accounts:

AccountID	Balance
A101	1000
A102	1500

Transactions:

TransactionID	AccountID	Amount
T1	A101	-200
T2	A102	+200

Schedule 1

T1: Read(A101), Subtract(A101, 200)

T2: Read(A102), Add(A102, 200)

Schedule 2

T2: Read(A102), Add(A102, 200)

T1: Read(A101), Subtract(A101, 200)

Schedule	Initial State	Final State
1	A101 = 1000 A102 = 1500	A101 = ? A102 = ?
2	A101 = 1000 A102 = 1500	A101 = ? A102 = ?

View Serialisability

- If a schedule is conflict serializable, then it will be view serializable

Accounts:

AccountID	Balance
A101	1000
A102	1500

Transactions:

TransactionID	AccountID	Amount
T1	A101	-200
T2	A102	+200

Schedule 1

T1: Read(A101), Subtract(A101, 200)

T2: Read(A102), Add(A102, 200)

Schedule 2

T2: Read(A102), Add(A102, 200)

T1: Read(A101), Subtract(A101, 200)

Schedule	Initial State	Final State
1	A101 = 1000 A102 = 1500	A101 = 800 A102 = 1700
2	A101 = 1000 A102 = 1500	A101 = 800 A102 = 1700

Non-Serializability

- ❑ a non-serial schedule that is not serializable
- ❑ Non-serializable schedules may/may not be consistent or recoverable.

- ❑ Non-serializable Schedule types:

Recoverable Schedule

Non-recoverable Schedule

Recoverable Schedule

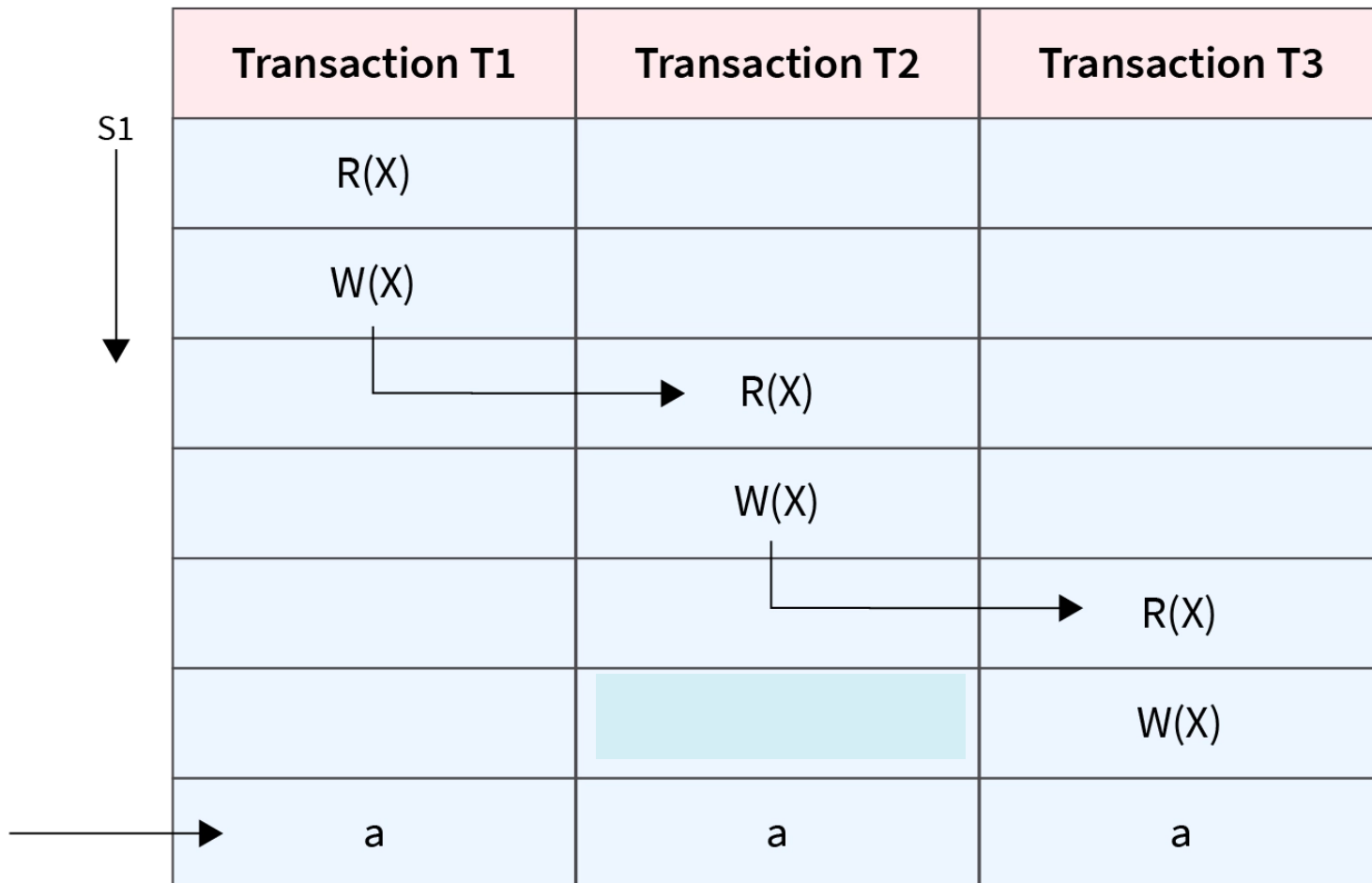
Transaction T1	Transaction T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
Commit	
	Commit

Recoverable Schedule - Types

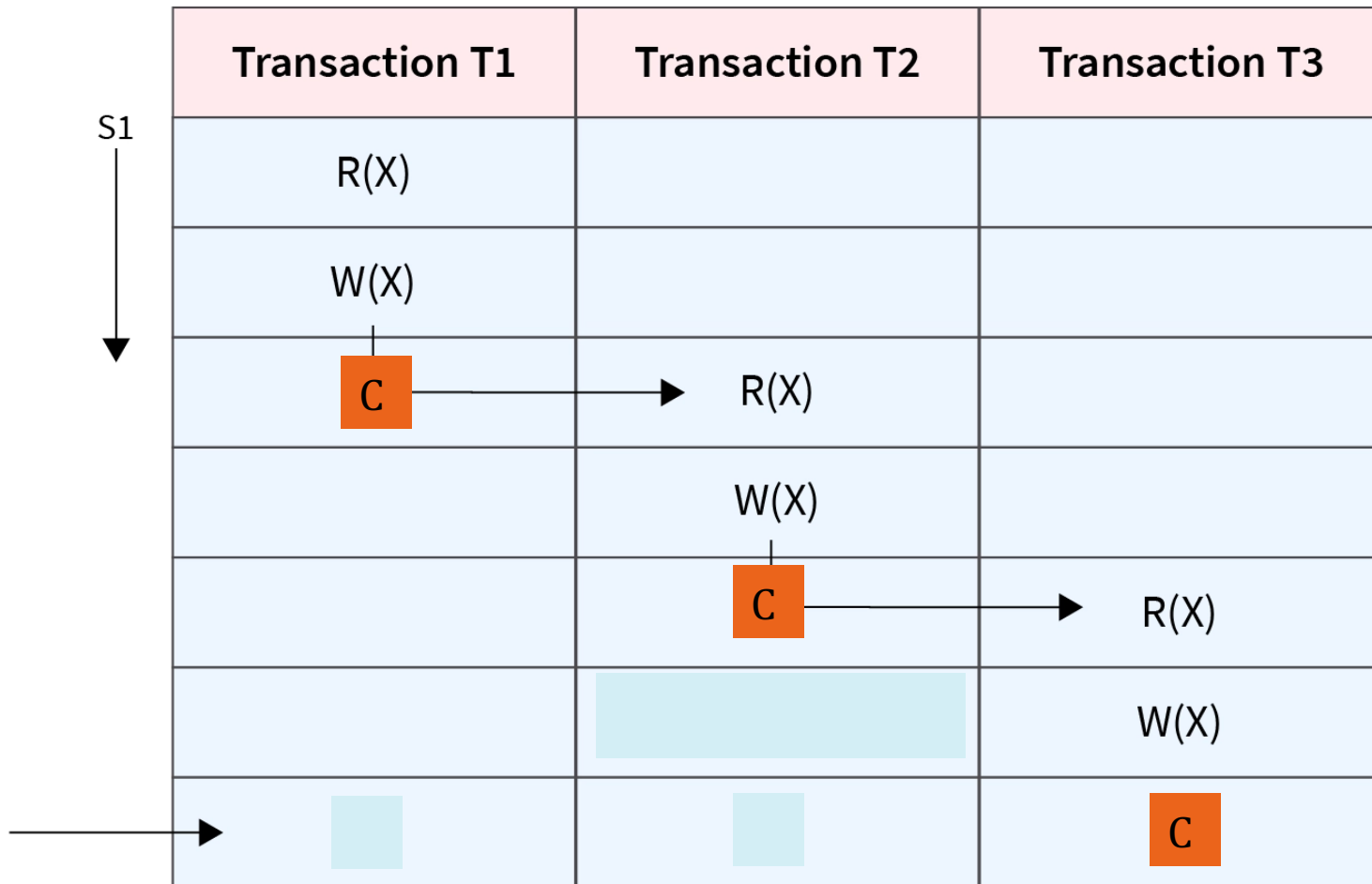
Transaction T1	Transaction T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
Commit	
	Commit

1. Cascading Schedule
2. Cascadeless Schedule
3. Strict Schedule

Cascading Schedule/ Rollback/Abort



Cascadeless Schedule



Strict Schedule

If in a schedule, *until the last transaction that has written it is committed or aborted*, a transaction is neither allowed to read nor write data item, then such a schedule is called as Strict Schedule.

Transaction (Ta)	Transaction (Tb)
R(X)	
	R(X)
W(X)	
Commit	
	W(X)
	R(X)
	commit

Non-Recoverable Schedule

If a transaction reads the value of an operation from an uncommitted transaction and commits before the transaction from where it has read the value, then such a schedule is called Non-Recoverable schedule.

Transaction T1	Transaction T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	commit

Concurrency Control Techniques

- ❑ Serializability can be achieved in several ways.
- ❑ Two main concurrency control techniques that allow transaction to execute safely in parallel subject to certain constraints.
 - a. Locking
 - b. Timestamp methods.

Locking Methods

- ❑ **Locking:** A procedure used to control concurrent access to data. When one transaction is accessing the database, a lock may deny access to other transactions to prevent incorrect results.
- ❑ Several methods of Locking, but they share a concept: that a transaction must claim a **shared** (*read*) or **exclusive** (*write*) lock on a data item before the corresponding database read or write operation.



Key elements of Locking

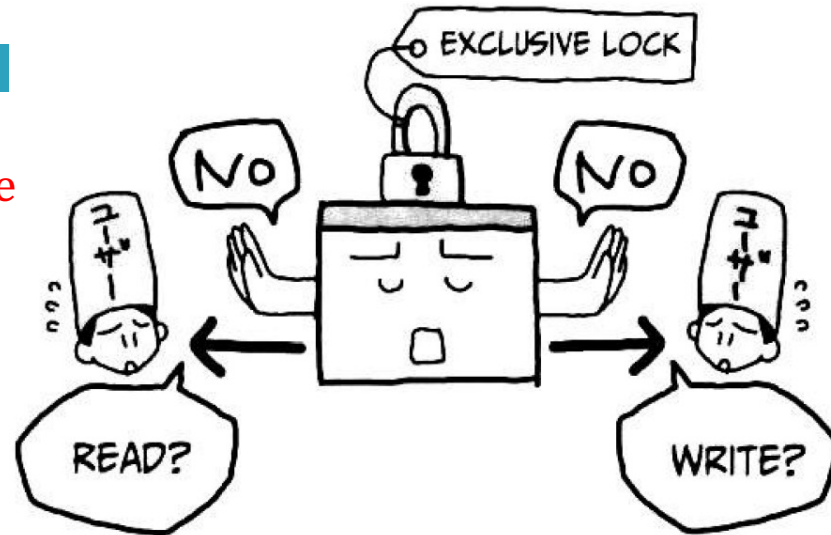
- ❑ **Lock:** Prevents another transaction from modifying or reading the data, in case of an exclusive lock.
- ❑ Locks can be placed on an entire database, or a table, or even a field.
- ❑ The size of the item determines the **granularity** of the lock.
- ❑ **Multiple Mode Locks:**
 - ❑ ***a. Shared/Read lock:*** If a transaction has a shared lock on a data item, it can read the item but not update it.
 - ❑ ***b. Exclusive/Write lock:*** If a transaction has an exclusive lock on a data item, it can both read and update the item.

Lock Compatibility

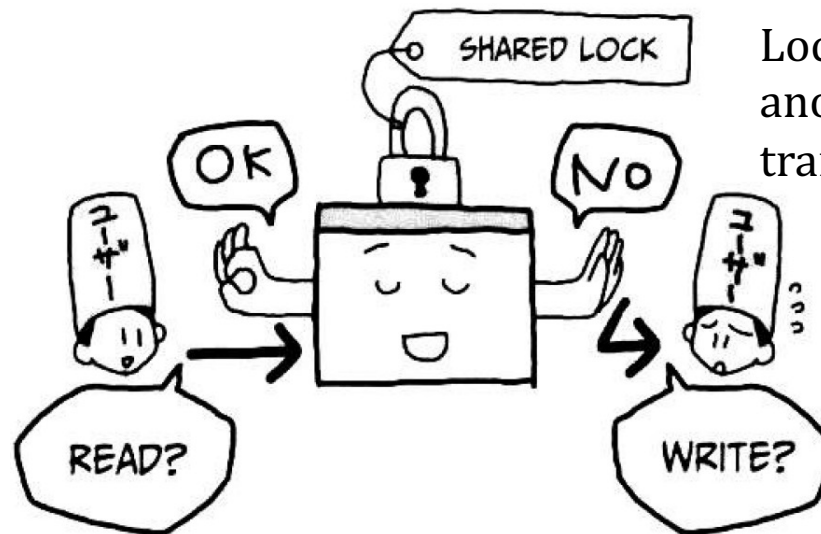
A transaction obtains a lock only if the lock is **compatible** with those locks **already** on the data item.

X → Exclusive

S → Shared



Locked by
another
transaction



Locked by
another
transaction

	X	S	---
X	False	False	True
S	False	True	True
---	True	True	True

Locking Example

- **Simple locking** **won't** successfully allow us to serialise all schedules.
For example:

A:1000, B:2000

T1	T2
Read(A) $A = A - 500$ Write(A) Read(B) $B = B + 500$ Write(B)	Read(A) Read(B) $A = A * 1.1$ $B = B * 1.1$ Write(A) Write(B)

A:550, B:2750 (expected)

Serial schedule

A:1000, B:2000

T1	T2
Lock-X(A) → Read(A) $A = A - 500$ Write(A) Unlock (A) → Lock-X(B) → Read(B) $B = B + 500$ Write(B) Unlock (B) →	 ← Lock-X(A) Lock-X(B) Read(A) Read(B) $A = A * 1.1$ $B = B * 1.1$ Write(A) Write(B) ← Unlock (A) Unlock (B)

A:550, **B:2700**

Locking Example

A:1000, B:2000

T1	T2
Read(A)	

Lock-X(A) →

A:1000, B:2000

T1	T2
Read(A)	

To guarantee serializability, we must follow an additional protocol concerning the positioning of the lock and unlock operations in every transaction. The best-known protocol is **two-phase locking (2PL)**.

	Write(A) Write(B)
--	----------------------

A:550, B:2750 (expected)

Unlock(B) →

B = B + 500 Write(B)	
-------------------------	--

A:550, B:2700

Serial schedule

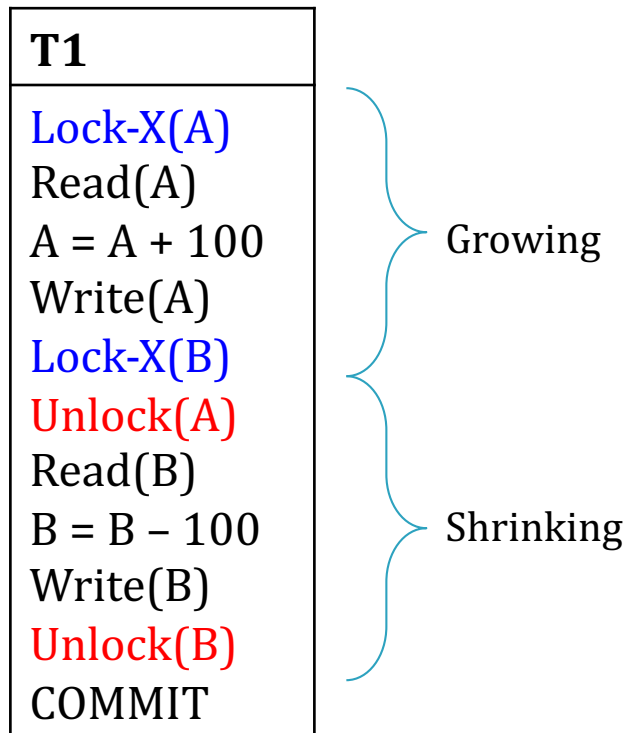
2PL – Two Phase Locking

- A transaction follows the **two-phase locking protocol** if all locking operations precede the first unlock operation in the transaction.
- Transaction divided into:
 - a. **Growing phase:** acquires all the locks needed but cannot release any locks.
 - b. **Shrinking phase:** it releases its locks but cannot acquire any new locks.
- Any schedule of two-phase locking transactions is ***conflict serializable***.

2PL Example

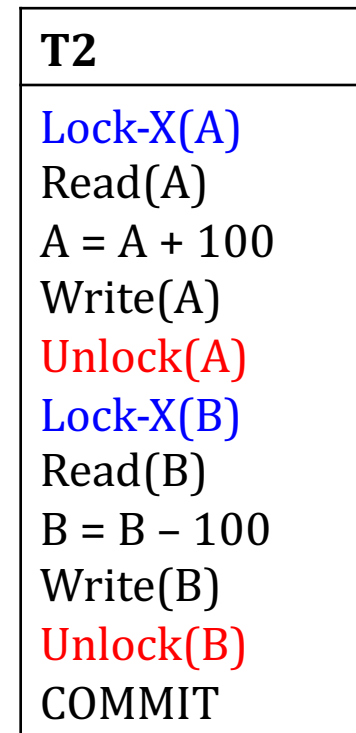
□ T1 follows 2PL protocol

- All locks in T1 are acquired **before** any are released.
- This happens even if the resource is no longer used.



□ T2 does not

- Releases a lock on A, which is no longer needed, before acquiring on B.



2PL - Questions

- When to lock?
 - ▣ Tuple **retrieve** operation: Lock-S
 - ▣ Tuple **update** operation (i.e. insert, delete, update): Lock-X
- When to unlock?
 - ▣ **Strict** 2PL Locking: unlock only at the end of transaction.
- Lock promotion in growing phase:
 - ▣ If a transaction T
 - holds a Lock-S on the tuple A, and
 - wishes to update tuple A
 - ▣ T must **promote/upgrade** Lock-S to Lock-X (according to lock compatibility).

Strict 2PL – solving Lost Update

- Strict 2PL prevents Lost Update but leads to deadlock!!!

Lock	X	S	---
X	False	False	True
S	False	True	True
---	True	True	True

T1	T2
Read(X) X = X + 10	
Write(X)	Read(X) X = X + 10
	Write(X) COMMIT
COMMIT	

Lock-S(X)

→ upgrade
Lock-X(X)

T1	T2
Read(X) X = X + 10	
Write(X) WAIT WAIT ...	Read(X) X = X + 10
	Write(X) WAIT WAIT ...
	...
	...

← Lock-S(X)

← upgrade
Lock-X(X)

Strict 2PL – solving Uncommitted Update

Lock	X	S	---
X	False	False	True
S	False	True	True
---	True	True	True

T1	T2
Read(X) X = X + 10 Write(X)	Read(X) X = X + 10 Write(X)
ROLLBACK	COMMIT

Lock-S(X) →
 upgrade →
 Lock-X(X) →
 Unlock (X)

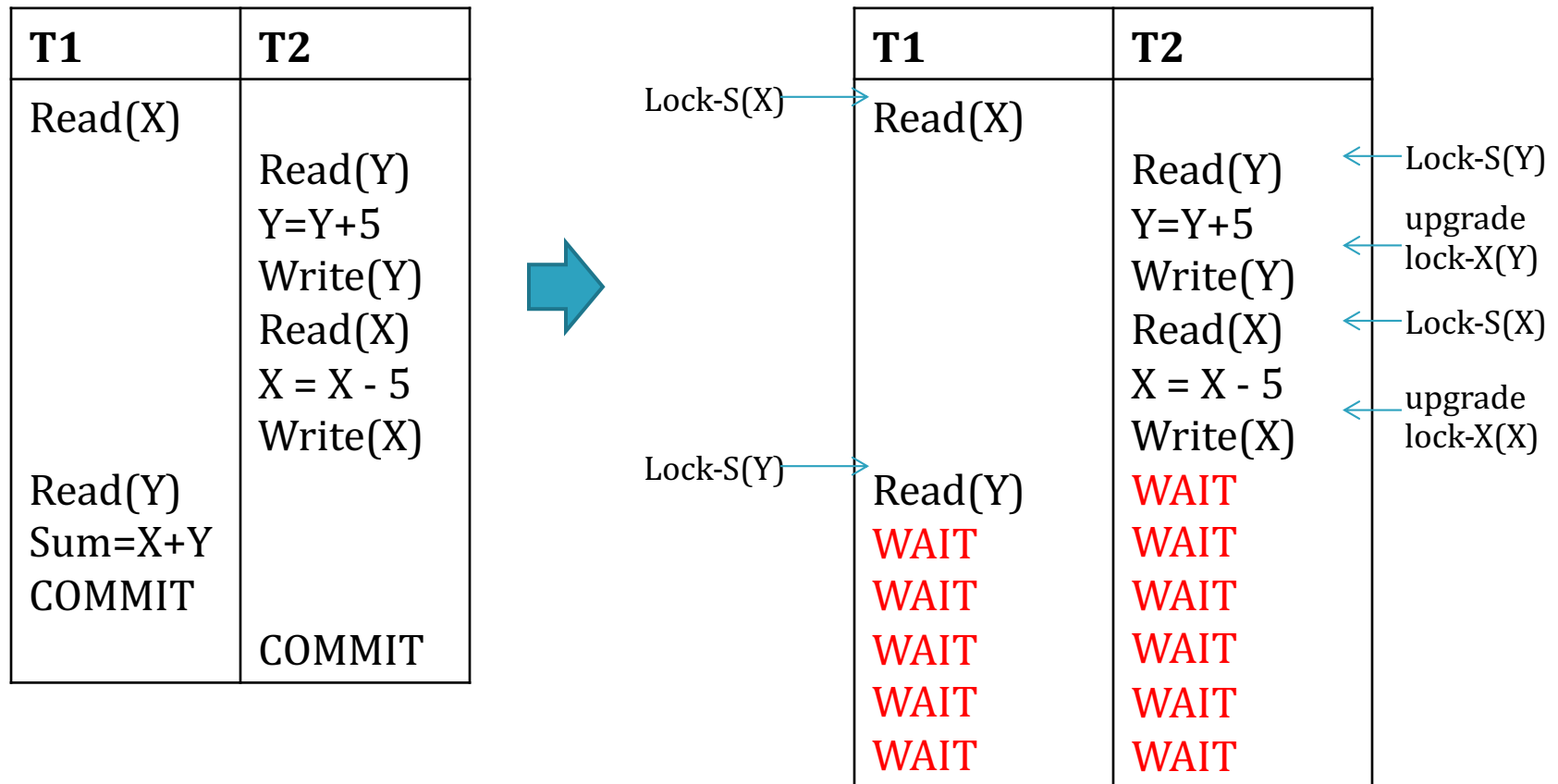
T1	T2
Read(X) X = X + 10 Write(X) ROLLBACK	WAIT Read(X) X = X + 10 Write(X) COMMIT

Lock-S(X) ←
 upgrade ←
 Lock-W(X) ←
 Unlock(X) ←

Strict 2PL – solving Inconsistent Analysis

- Strict 2PL prevent Inconsistent Analysis but leads to deadlock.

Lock	X	S	---
X	False	False	True
S	False	True	True
---	True	True	True



Deadlock

- A problem with two-phase locking, which applies to all locking-based schemes, is that it can cause **deadlock**, since transactions can wait for locks on data items.
- **Definition:** An impasse that may result when two (or more) transactions are each waiting for locks to be released that are held by the other.
- Once deadlock occurs, the applications involved cannot resolve the problem. Instead, the DBMS must recognize that deadlock exists and break the deadlock in some way.

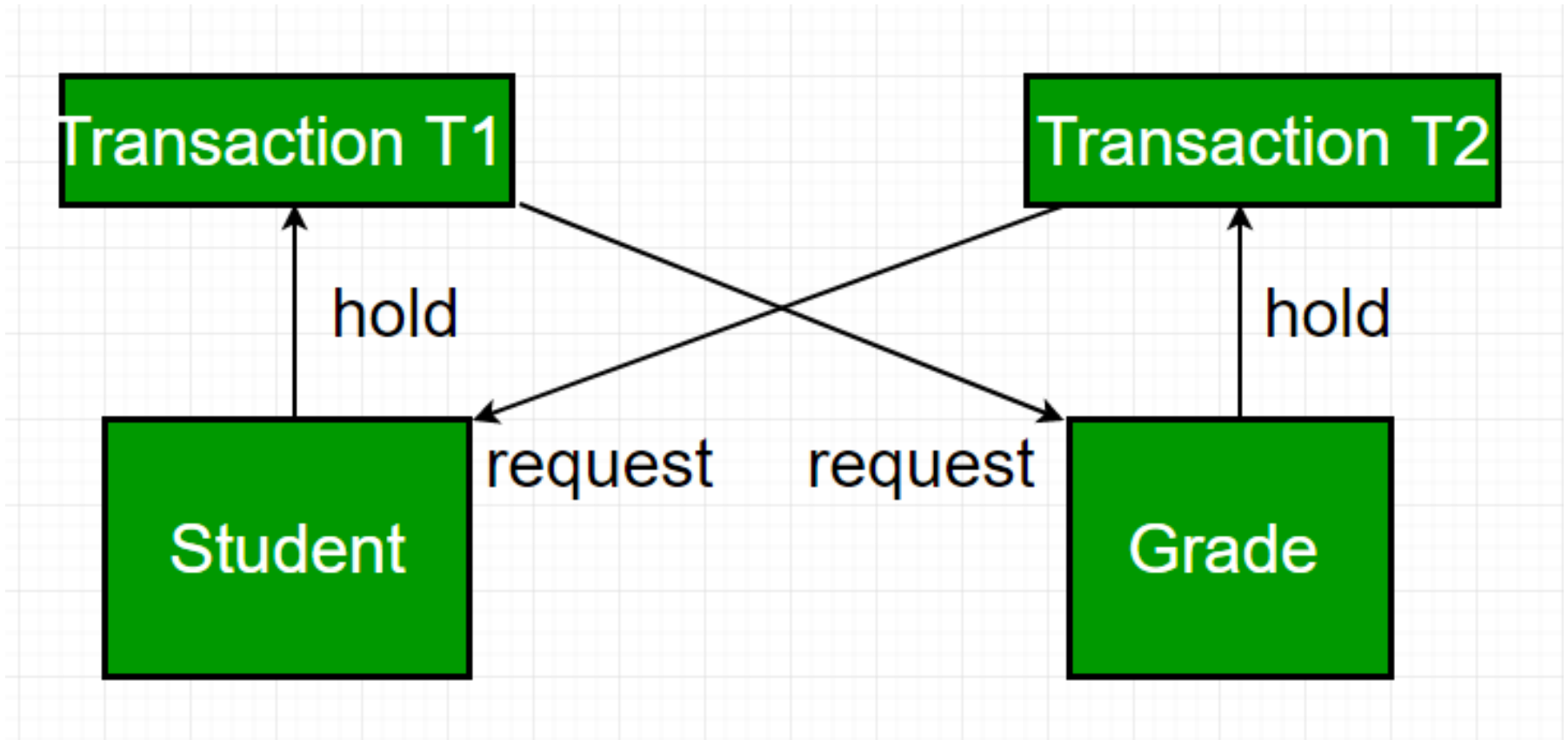
10



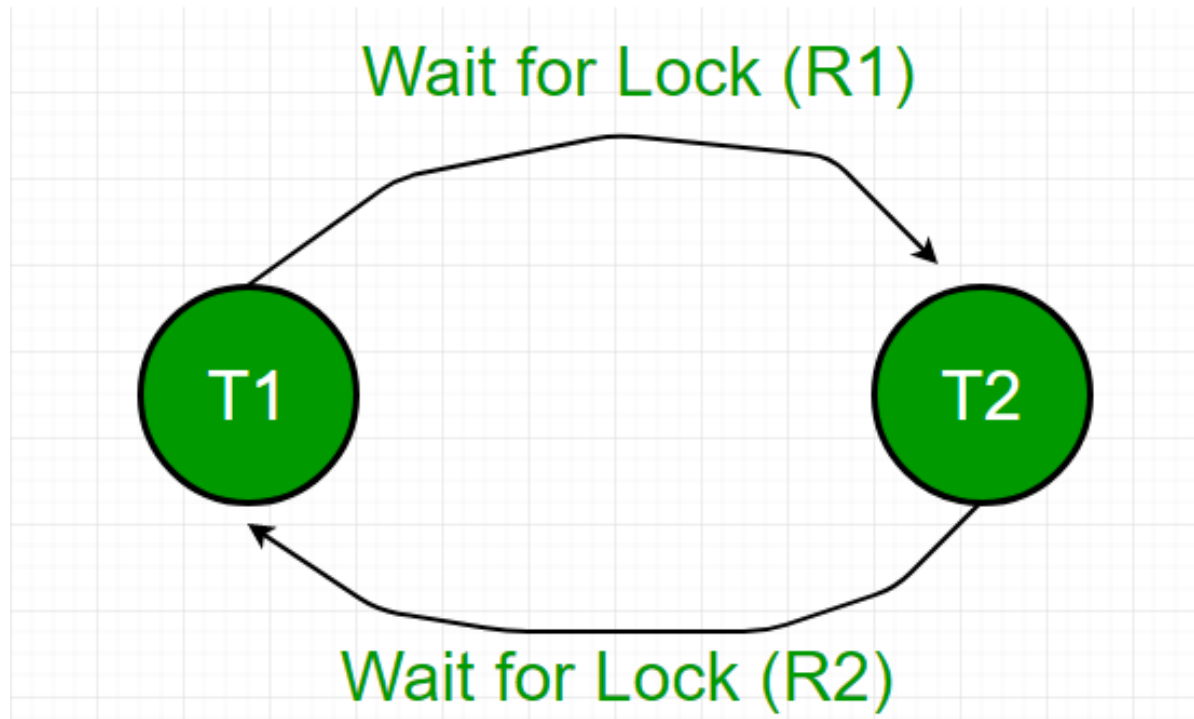
T1	T2
Read(X)	
Write(X)	
	Read(Y)
	Write(Y)
Read(Y)	
WAIT	Read(X)
WAIT	
WAIT	WAIT
WAIT	WAIT
WAIT	WAIT
WAIT	WAIT
...	WAIT
	...

- Lock-S(Y)
- Lock-X(Y)
- Lock-X(X)

Deadlocks

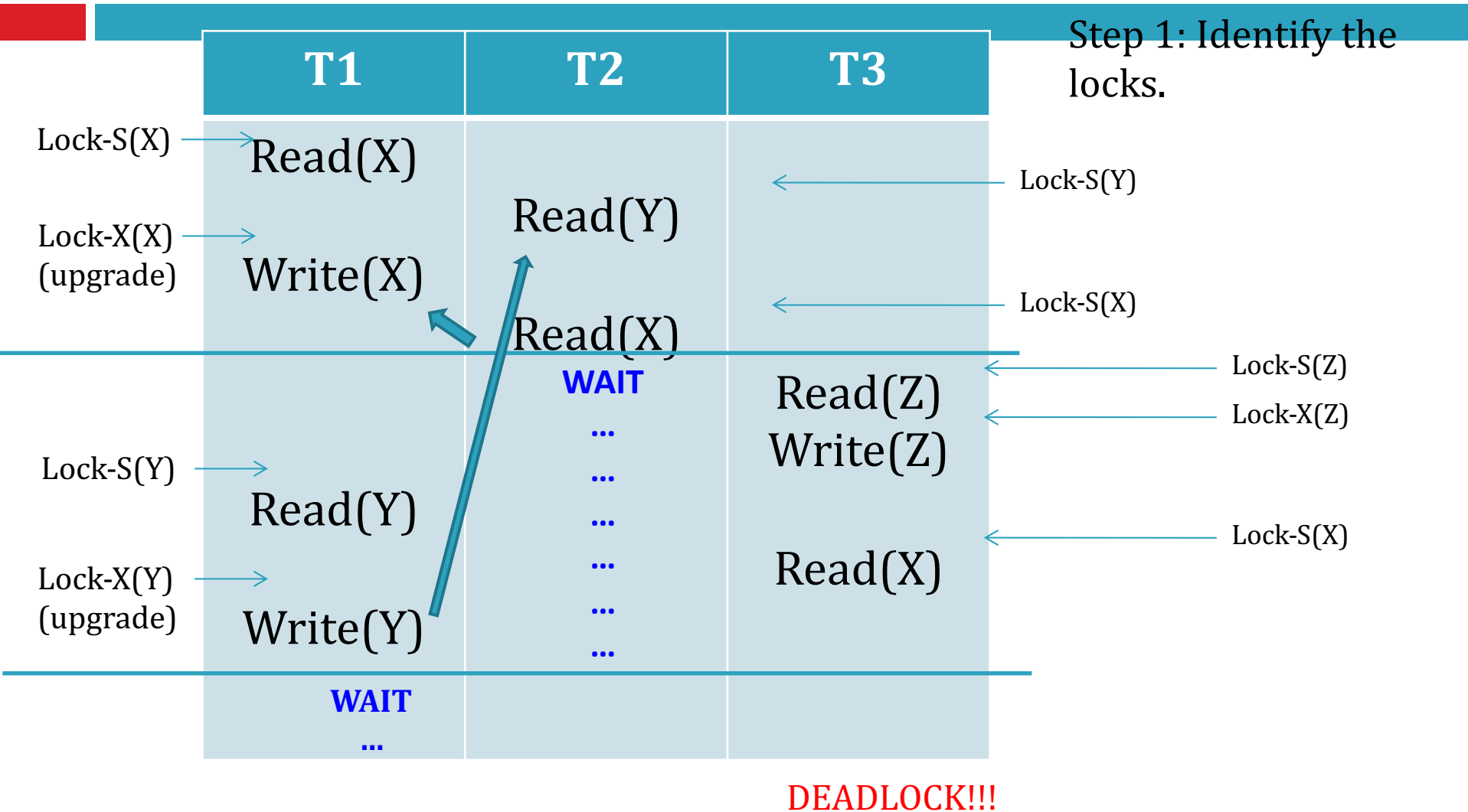


Finding a deadlock

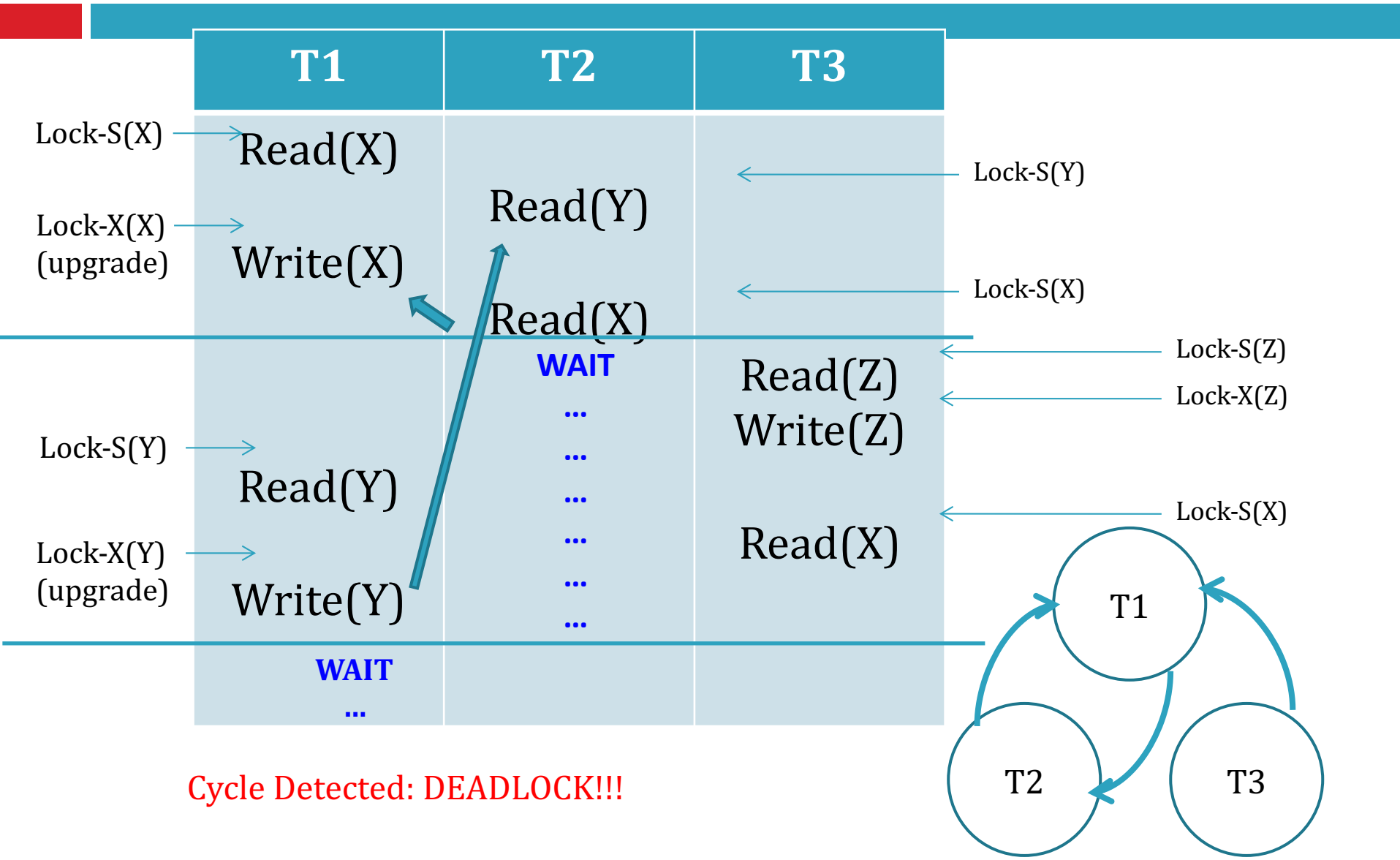


DEADLOCK!!!

Finding a deadlock



Wait-For Graph



Handling Deadlocks

- There are three general techniques for handling deadlock:
 - a. **Timeouts:** the transaction that has requested a lock waits for at most a specified period of time.
 - b. **deadlock prevention:** the DBMS looks ahead to determine if a transaction would cause deadlock, and never allows dead- lock to occur. Algorithms used:
 - a. Wait-Die
 - b. Wound-Wait
 - c. Conservative 2PL
 - c. **deadlock detection and recovery:** DBMS allows deadlock to occur but recognizes occurrences of deadlock and breaks them.

Handling Deadlocks

□ Algorithms used:

- a. **Wait-Die**
- b. Wound-Wait
- c. Conservative 2PL

T1 is older and T2 has held some resource	T1 is allowed to wait until the resource is available
T1 is older and it has held some resource required by T2	T2 is killed and restarted after a random delay with the same timestamp

Handling Deadlocks

□ Algorithms used:

- a. Wait-Die
- b. **Wound-Wait**
- c. Conservative 2PL

T1 is older and T2(younger) has held some resource	T1 forces T2 to kill and release the resource. T2 restarts after a minute with the same timestamp
T1 is older and it has held some resource required by T2	T2 is asked to wait till T1 releases it.

Deadlock Detection – Wait-For Graph

- Each transaction is a vertex
- Edge from **T2 to T1** if
 - ▣ T1 Lock-S(X) then T2 tries to Lock-X(X) it
 - ▣ T1 Lock-X(X) then T2 tries to Lock-S(X) it
 - ▣ T1 Lock-X(X) then T2 tries to Lock-X(X) it

Lock	X	S	---
X	False	False	True
S	False	True	True
---	True	True	True

Recovery from Deadlock Detection

- a. **Choice of deadlock victim:** long running transaction?, how many data items is still to update etc.
- b. **How far to roll a transaction back:** rollback part of it or rollback all of it?
- c. **Avoiding starvation:** Ensure if the same transaction is a victim, how to ensure there is a fairness in choosing the victim.

Timestamping Methods

- ❑ **Timestamp:** A unique identifier created by the DBMS that indicates the relative starting time of a transaction.
- ❑ **Timestamping:** A concurrency control protocol that orders transactions in such a way that older transactions, transactions with *smaller* timestamps, get priority in the event of conflict.
- ❑ **Timestamp ordering protocol:** Each transaction is assigned a timestamp. Older transactions have smaller timestamps, younger transactions have larger timestamps.

Timestamping Ordering Protocol (TOP)

- For each data item Q , TOP maintains:

W-timestamp(Q) = the largest timestamp of *all* transactions that wrote to Q .

When it is last being written?

- **R-timestamp**(Q) = the largest timestamp of *all* transactions that read Q .

When it is last being read?

Read Rule of TOP

□ A transaction T requests **read**(Q)

1. $TS(T) < W\text{-timestamp}(Q)$,

Rejected, and T is rolled back.

(If T proceeds, it would read Q that was already updated by a younger transaction).

2. $TS(T) \geq W\text{-timestamp}(Q)$

Granted and $R\text{-timestamp}(Q) = \text{MAX}(R\text{-timestamp}(Q), TS(T))$

Write Rule of TOP

- A transaction T requests **write**(Q).
 1. $TS(T) < \text{R-timestamp}(Q)$
Rejected (a younger transaction already read Q).
 2. $TS(T) < \text{W-timestamp}(Q)$,
Rejected (a younger transaction already modified Q).
 3. Otherwise
Granted and $\text{W-timestamp}(Q) = TS(T)$.

Properties of TOP

- ❑ Guarantees conflict serializability.
- ❑ No deadlock
 - ▣ No transaction ever waits
- ❑ Problems
 - ▣ Long transactions might keep getting restarted by new transactions - starvation
 - ▣ Bias to younger transactions => Rolls back old transactions, which may have done a lot of work

Summary



- Concurrency Control
- Problems associated with concurrency control
- Locking
- Deadlocks
- Timestamps

Coming up...

Recovery
&
Security

Lab Exercise



- ❑ Download Instructions from Moodle