

# Concurrency control protocol

# Lost Update

- ▶ T1 and T2 read X, both modify it, then both write it out
  - ▶ The net effect of T1 and T2 should be no change on X
  - ▶ Only T2's change is seen, however, so the final value of X has increased by 5

T1	T2
<b>Read (X)</b> <b><math>X = X - 5</math></b>	<b>Read (X)</b> <b><math>X = X + 5</math></b>
<b>Write (X)</b>	<b>Write (X)</b>
<b>COMMIT</b>	<b>COMMIT</b>

# Uncommitted Update

- ▶ T2 sees the change to X made by T1, but T1 is rolled back
  - ▶ The change made by T1 is undone on rollback
  - ▶ It should be as if that change never happened

T1	T2
Read(X) $X = X - 5$ Write(X)	Read(X) $X = X + 5$ Write(X)
ROLLBACK	COMMIT

# Inconsistent analysis

- ▶ T1 doesn't change the sum of X and Y, but T2 sees a change
  - ▶ T1 consists of two parts - take 5 from X and then add 5 to Y
  - ▶ T2 sees the effect of the first, but not the second

T1	T2
<b>Read(X)</b> <b>X = X - 5</b> <b>Write(X)</b>	<b>Read(X)</b> <b>Read(Y)</b> <b>Sum = X+Y</b>
<b>Read(Y)</b> <b>Y = Y + 5</b> <b>Write(Y)</b>	

# Inconsistent Retrieval Problem

- ▶ Inconsistent retrieval problem occurs when a transaction calculates some aggregate functions over a set of data while transactions are updating the data
  - ▶ Some data may be read after they are changed and some before they are changed yielding inconsistent results

# Retrieval During Update

TRANSACTION T1	TRANSACTION T2
SELECT SUM(PROD_QOH) FROM PRODUCT	UPDATE PRODUCT SET PROD_QOH = PROD_QOH + 10 WHERE PROD_CODE = '1546-QQ2'
	UPDATE PRODUCT SET PROD_QOH = PROD_QOH - 10 WHERE PROD_CODE = '1558-QW1'
	COMMIT;

# Concurrency Control with Locking Methods

- ▶ Lock
  - ▶ Guarantees exclusive use of a data item to a current transaction
    - ▶ T2 does not have access to a data item that is currently being used by T1
    - ▶ Transaction acquires a lock prior to data access; the lock is released when the transaction is complete
  - ▶ Required to prevent another transaction from reading inconsistent data
- ▶ Lock manager
  - ▶ Responsible for assigning and policing the locks used by the transactions

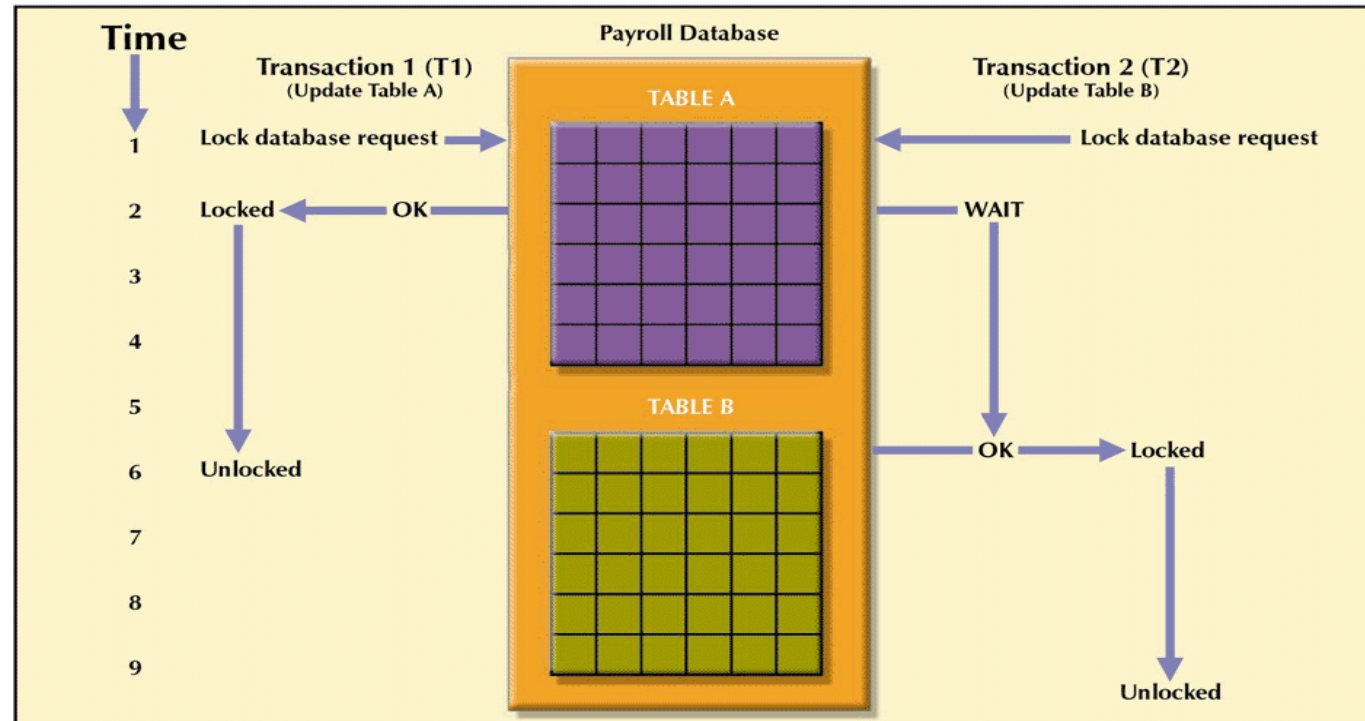
# Lock Granularity

- ▶ Indicates the level of lock use
- ▶ Locking can take place at the following levels:
  - ▶ Database-level lock
    - ▶ Entire database is locked
  - ▶ Table-level lock
    - ▶ Entire table is locked
  - ▶ Page-level lock
    - ▶ Entire disk page is locked
  - ▶ Row-level lock
    - ▶ Allows concurrent transactions to access different rows of the same table, even if the rows are located on the same page
  - ▶ Field-level lock
    - ▶ Allows concurrent transactions to access the same row, as long as they require the use of different fields (attributes) within that row



# A Database-Level Locking Sequence

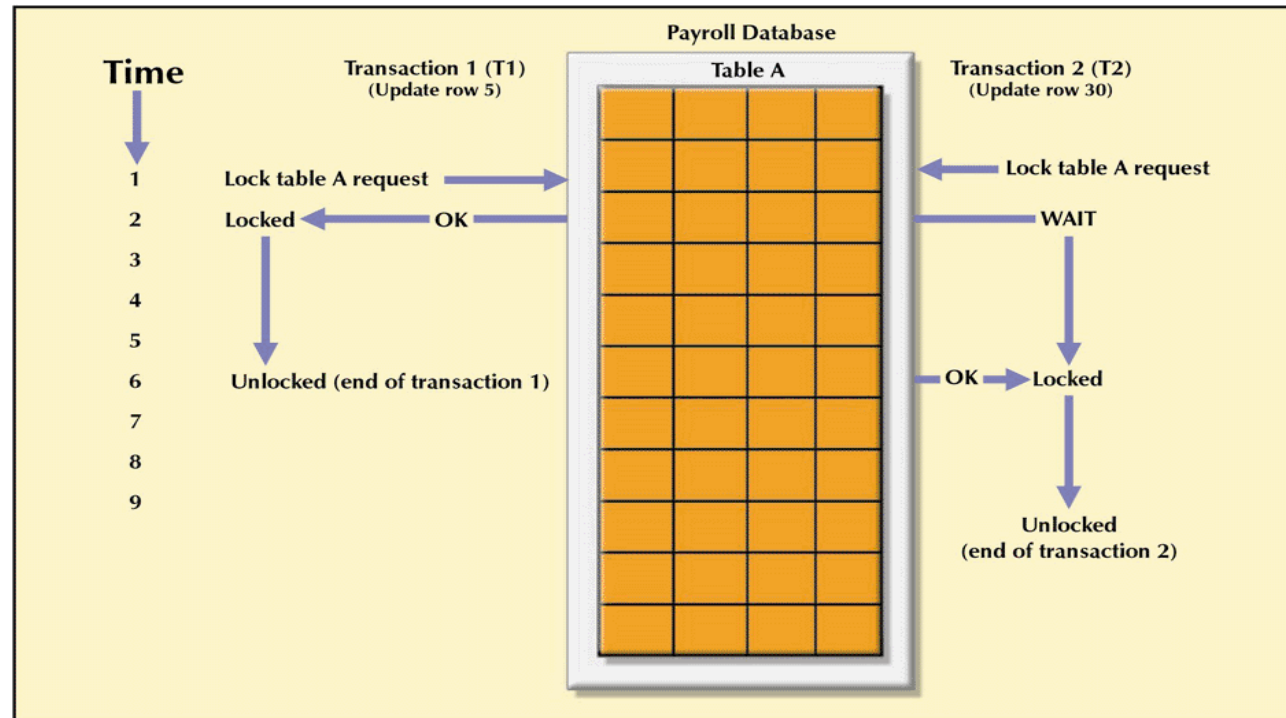
FIGURE 9.3 A DATABASE-LEVEL LOCKING SEQUENCE



- ❑ Good for batch processing but unsuitable for online multi-user DBMSs
- ❑ T1 and T2 can not access the same database concurrently even if they use different tables

# Table-Level Lock

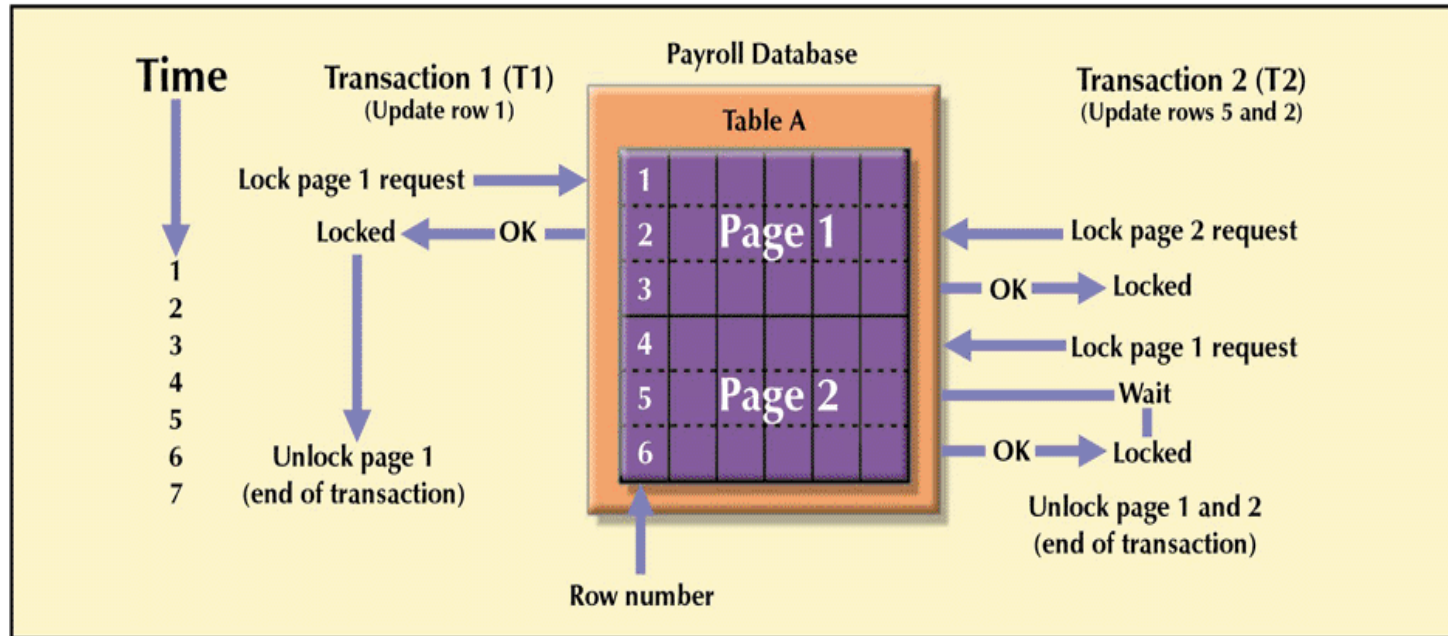
FIGURE 9.4 AN EXAMPLE OF A TABLE-LEVEL LOCK



- ❑ T1 and T2 can access the same database concurrently as long as they use different tables
- ❑ Can cause bottlenecks when many transactions are trying to access the same table (even if the transactions want to access different parts of the table and would not interfere with each other)
- ❑ Not suitable for multi-user DBMSs

# Page-Level Lock

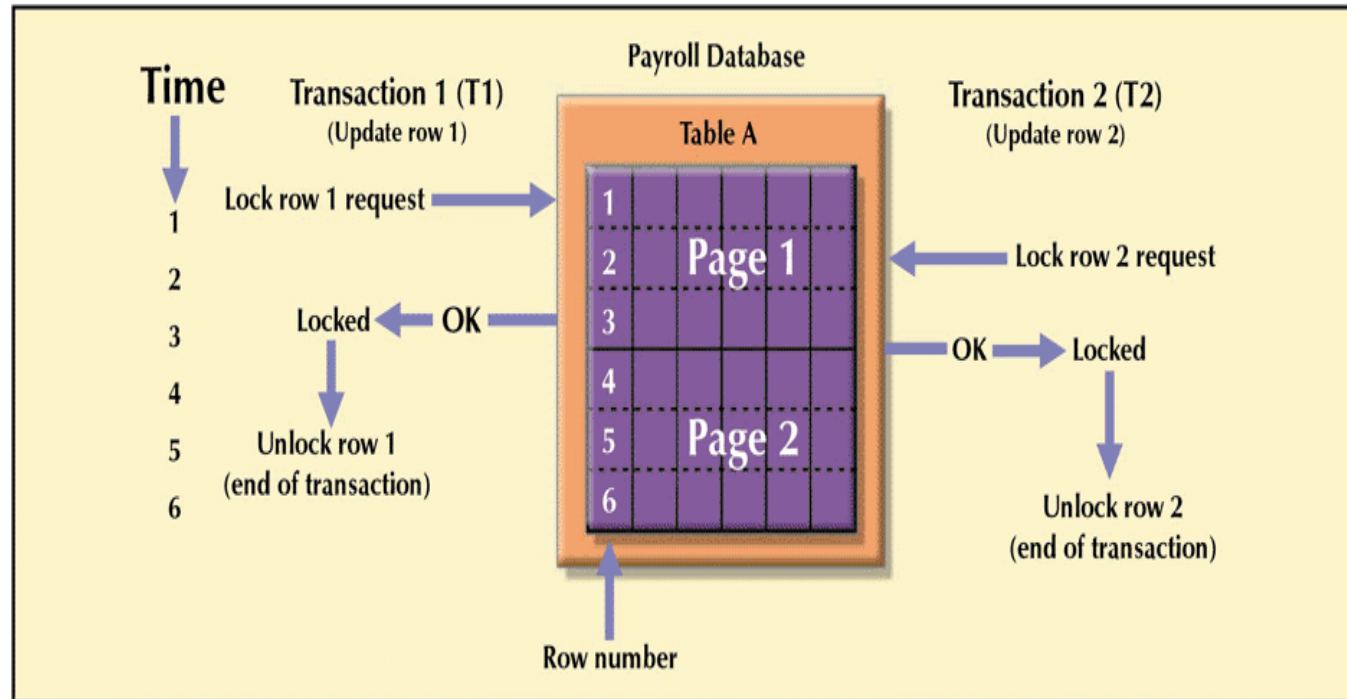
FIGURE 9.5 AN EXAMPLE OF A PAGE-LEVEL LOCK



- ❑ An entire disk page is locked (a table can span several pages and each page can contain several rows of one or more tables)
- ❑ Most frequently used multi-user DBMS locking method

# Row-Level Lock

FIGURE 9.6 AN EXAMPLE OF A ROW-LEVEL LOCK



- ❑ Concurrent transactions can access different rows of the same table even if the rows are located on the same page
- ❑ Improves data availability but with high overhead (each row has a lock that must be read and written to)

# Field-Level Lock

- ❑ Allows concurrent transactions to access the same row as long as they require the use of different fields with that row
- ❑ Most flexible lock but requires an extremely high level of overhead

# Disadvantage

- ▶ All the mentioned locks are dependent on database resources, but not only on transactions.
- ▶ It will increase the overhead and cause deadlocks.
- ▶ Two Phase locking (2PL) is another type of solution where the resources will be locked based on the need of transactions.

# Shared/Exclusive Locks

- ▶ Exclusive lock (X)
  - ▶ Access is specifically reserved for the transaction that locked the object
  - ▶ Must be used when the potential for conflict exists - when a transaction wants to update a data item and no locks are currently held on that data item by another transaction
  - ▶ *Granted if and only if no other locks are held on the data item*
- ▶ Shared lock (S)
  - ▶ Concurrent transactions are granted Read access on the basis of a common lock
  - ▶ Issued when a transaction wants to read data and no exclusive lock is held on that data item
    - ▶ Multiple transactions can each have a shared lock on the same data item if they are all just reading it
- ▶ Mutual Exclusive Rule
  - ▶ Only one transaction at a time can own an exclusive lock in the same object



# Shared/Exclusive Locks

## ► Exclusive lock

Transaction (T)
X(A)
R(A)
W(A)

## ► Shared lock

Transaction (T)
S(A)
R(A)
W(A) <b>X</b>

Not allowed



# Lock compatible table:

REQUESTING $T_j$		
	S	X
S	YES	NO
X	NO	NO

HOLD  $T_i$

# Compatibility table

Request



Grant



	S	X
S	Yes	No
X	No	No

S-S

T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
	U(A)

S-X

T1	T2
S(A)	
R(A)	
	X(A)
	R(A)
	W(A)
	U(A)

X-S

T1	T2
X(A)	
R(A)	
W(A)	
	S(A)
	R(A)
	U(A)

X-X

T1	T2
X(A)	
W(A)	
	X(A)
	R(A)
	W(A)
	U(A)

# Shared/Exclusive Locks

- ▶ Increased overhead
  - ▶ The type of lock held must be known before a lock can be granted
  - ▶ Three lock operations exist:
    - ▶ READ\_LOCK to check the type of lock
    - ▶ WRITE\_LOCK to issue the lock
    - ▶ UNLOCK to release the lock

# Problems: Shared/Exclusive Locks

T1	T2
Lock X(A)	
W(A)	
Unlock X(A)	
	Lock S (A)
	R(A)
	Unlock S(A)
	Lock S (B)
	R(B)
	Unlock S(B)
Lock X (B)	
W(A)	
Unlock X(B)	

- This schedule will executes successfully on s/x locking
- But this schedule is not sufficient to ensure serializability

# Problems in S/X locking

- May not sufficient to produce only serializable schedule

T1	T2
X(A)	
R(A)	
W(A)	
U(A)	
	S(A)
	R(A)
	U(A)
X(A)	
W(A)	
U(A)	

- May not free from irrecoverability

T1	T2
X(A)	
R(A)	
W(A)	
U(A)	
	S(A)
	R(A)
	U(A)
	Commit
S(B)	
R(B)	
Fail	

## Problems in S/X locking Contd...

- ▶ May not free from deadlock

X(A) R(A) W(A)	
	X(B) W(B)
Wants X(B)	
	Wants X(A)

- ▶ May not free from starvation

T1	T2	T3	T4
	S(A)		
X(A) Wait			
		S(A)	
			S(A)
		U(A)	
			U(A)

# Two-Phase Locking to Ensure Serializability

- ▶ Defines how transactions acquire and relinquish locks
- ▶ Guarantees serializability, but it does not prevent deadlocks
  - ▶ *Growing phase*, in which a transaction acquires all the required locks without unlocking any data
  - ▶ *Shrinking phase*, in which a transaction releases all locks and cannot obtain any new lock

# Two-Phase Locking to Ensure Serializability

- ▶ Governed by the following rules:
  - ▶ Two transactions cannot have conflicting locks
  - ▶ No unlock operation can precede a lock operation in the same transaction
  - ▶ No data are affected until all locks are obtained—that is, until the transaction is in its locked point

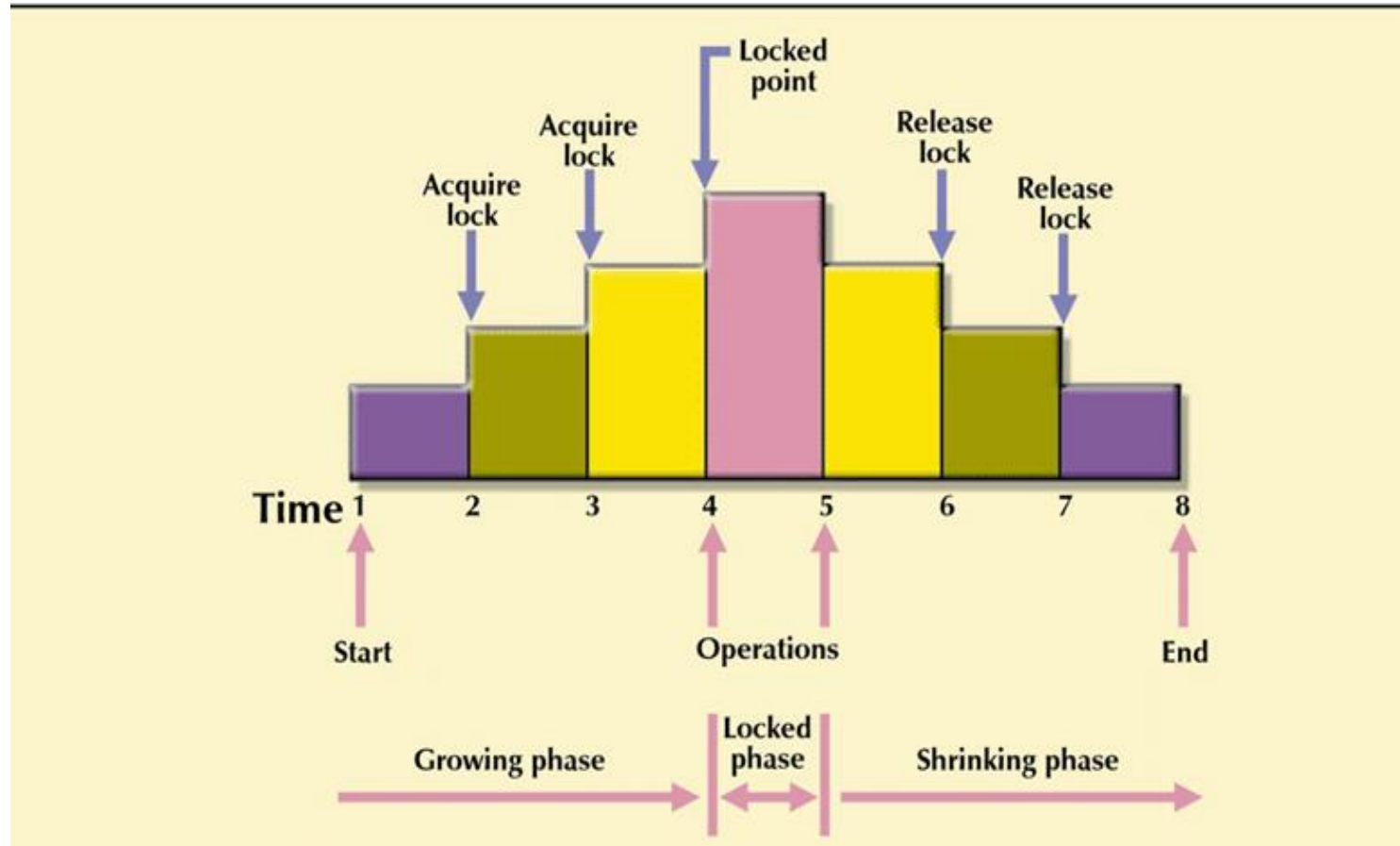


# 2PL example

T1	T2
Lock S(A)	
	Lock S (A)
Lock X(B)	
Unlock (A)	
	Lock X (C)
Unlock X(B)	
	Unlock S(A)
	Unlock X(C)

Advantage: We can always achieve serializability

# Two-Phase Locking Protocol



# Problems in 2PL example

T1	T2
Lock X(A)	
R(A)	
W(A)	
Unlock (A)	
	Lock X (A)
	R(A)
	W(A)
	Commit
	.
	.
	.
	.

T1 can rollback. But T2 already read T1's data and committed. So it is Irrecoverable schedule.

T1	T2
Lock X(A)	
	Lock X(B)
X(B) (wait)	
	X(A) wait
	.
	.
	.
	.

T1 and T2 both have to wait for indefinite time. So it creates deadlock.

The possible solutions are:

- Strict 2PL (all exclusive locks should hold until commit/Abort)
- Rigorous 2PL (all Shared & exclusive locks should hold until commit/Abort)
- Conservative 2PL (1 transaction has to achieve all the locks, otherwise it will release all the locks)

# Concurrency Control with Time Stamping Methods

- ▶ Assigns a global unique time stamp to each transaction
  - ▶ All database operations within the same transaction must have the same time stamp
- ▶ Produces an explicit order in which transactions are submitted to the DBMS
- ▶ Uniqueness
  - ▶ Ensures that no equal time stamp values can exist
- ▶ Monotonicity
  - ▶ Ensures that time stamp values always increase
- ▶ Disadvantage
  - ▶ Each value stored in the database requires two additional time stamp fields - last read, last update

# Time stamp ordering protocol

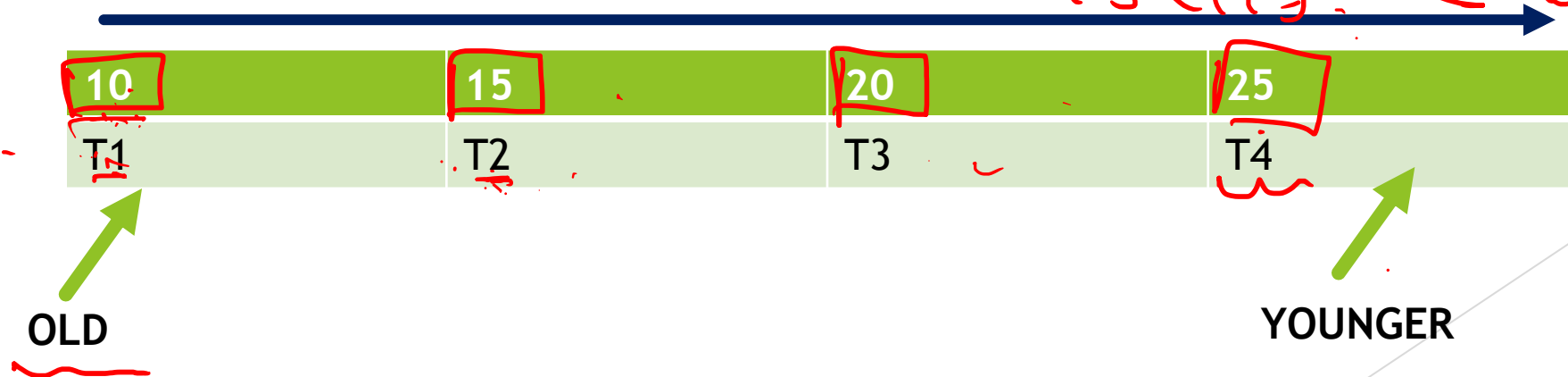
- ▶ Unique value assigned by DBMS (concurrency controller) to every transaction in ascending order.
- ▶ UNIQUE: NO TWO TRANSACTION HAVE THE SAME TS VALUE.
- ▶ ASCENDING ORDER: OLDER  $T^R$ , TS (TIMESTAMP) is less than younger transaction TS value.
- ▶ TS VALUE: STARTING FROM  $T^R$ .

Concurrency Controller

S  
 $T_1, T_2, T_3, T_4$

$\Rightarrow RE$   
 $\Rightarrow W$   
 $TS(T_1) < TS(T_4)$

$TS(Old) < TS(Younger)$   
 $TS(T_1) < TS(T_4)$



# READ TIMESTAMP VALUE (A):

- ▶ HIGHEST  $T^R$  TIMESTAMP VALUE THAT HAS PERFORMED  $(R(A))$  OPERATION SUCESSFULLY

10	15	20	25
T1	T2	T3	T4
R(A)			
		R(A)	
	R(A)		

RTS(A) = 0

✓ 0

# READ TIMESTAMP VALUE (A):

- ▶ HIGHEST  $T^R$  TIMESTAMP VALUE THAT HAS PERFORMED  $R(A)$  OPERATION SUCCESSFULLY

10	15	20	25
T1	T2	T3	T4
<u>R(A)</u>			
		R(A)	
	R(A)		

RTS(A) = 10

# READ TIMESTAMP VALUE (A):

- ▶ HIGHEST  $T^R$  TIMESTAMP VALUE THAT HAS PERFORMED  $R(A)$  OPERATION SUCCESSFULLY

10	15	20	25
T1	T2	T3	T4
R(A)			
		R(A)	
	R(A)		

RTS(A) = 20 ✓



# READ TIMESTAMP VALUE (A):

TSOP

- ▶ HIGHEST  $T^R$  TIMESTAMP VALUE THAT HAS PERFORMED  $R(A)$  OPERATION SUCCESSFULLY

Max (1- - -)

10	15	20	25
T1	T2	T3	T4
R(A)			
		R(A)	
			R(A)

$RTS(A) = 20$

$RTS(A) = 20$

15, 20, 25

NOTE: T2 CAN'T CHANGE THE RTS VALUE OF A BECAUSE TS OF T2 IS LESS THAN T3

0  $\Rightarrow$  4

4  $\rightarrow$  0

# WRITE TIMESTAMP VALUE (A):

- ▶ HIGHEST  $T^R$  TIMESTAMP VALUE THAT HAS PERFORMED  $W(A)$  OPERATION SUCCESSFULLY

10	15	20	25
T1	T2	T3	T4
W(A)			
			W(A)

WTS(A) = 25

NOTE: T2 CAN'T CHANGE THE RTS VALUE OF A BECAUSE TS OF T2 IS LESS THAN T3

# BASIC IDEA OF TS ORDERING PROTOCOLS

old

S

10 T1	30 T2	20 T3
1		
	1	
		1
	1	
1		

✓ Younger

Conflict pair =  $\underline{W(A)}$

Follow  $R(A)$

NOTE: CONCURRENT EXECUTION BETWEEN  $T^R$  SHOULD BE EQUAL TO SERIAL SCHEDULE BASED ON TIMESTAMP ORDERING.

- TIMESTAMP ORDERING PROTOCOL ALLOW SERIAL EXECUTION ONLY IF

=  $\boxed{T1 \rightarrow T3 \rightarrow T2}$   
~~10 20 30~~

$T_2 \Rightarrow T_1$

$\leftarrow \begin{matrix} 20 \\ T_2 \\ W(A) \end{matrix} \quad \begin{matrix} 22 \\ T_1 \\ R(A) \end{matrix}$

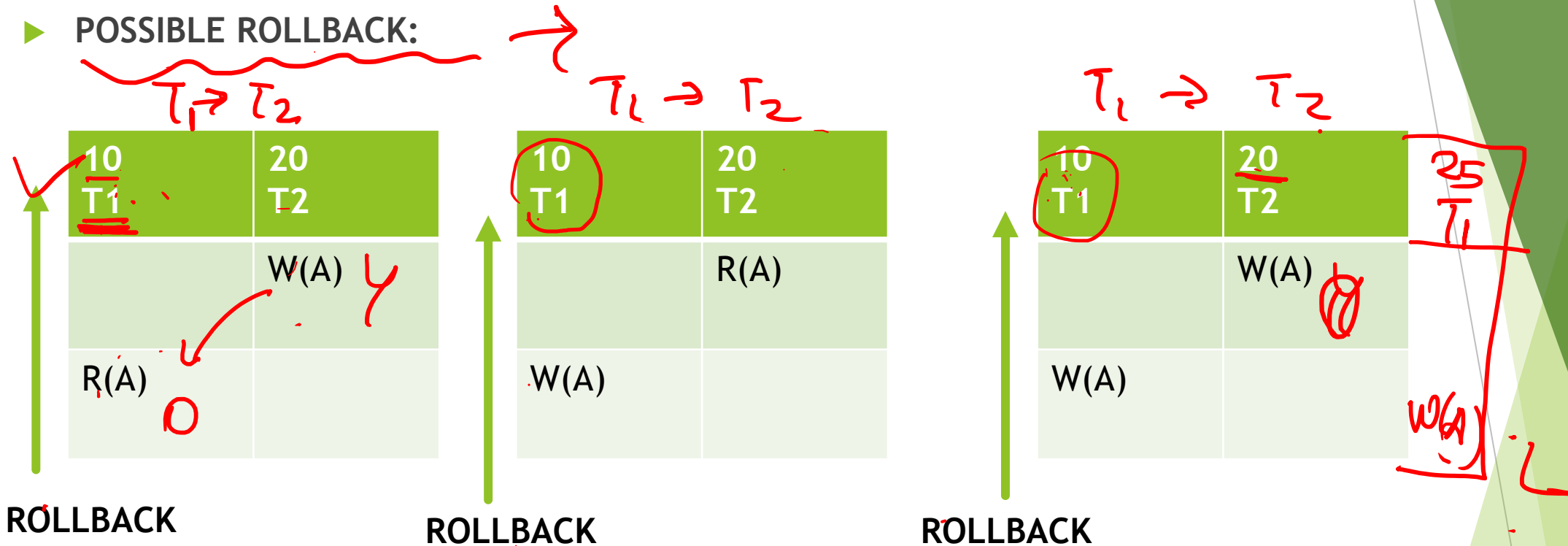
$\Rightarrow \text{No action}$

- If there is a conflict pair from younger TR to old TR then that conflict pair doesn't allowed to schedule
- Ts ordering protocols allowed to execute conflict pair only if conflict pair precedence and Ts order is same.

Action

# BASIC TIMESTAMP ORDERING PROTOCOL

## ► POSSIBLE ROLLBACK:



$Y \Rightarrow O$

$O \rightarrow \text{NO action}$

# Timestamp ordering protocol

## ■ If a transaction $T_i$ issues a read(X) operation –

■ If  $TS(T_i) < \text{W-timestamp}(X)$

■ Operation rejected

Else ■ If  $TS(T_i) \geq \text{W-timestamp}(X)$

■ Operation executed.

■ All data-item timestamps updated.

## ■ If a transaction $T_i$ issues a write(X) operation –

■ If  $TS(T_i) < \text{R-timestamp}(X)$

■ Operation rejected.

■ If  $TS(T_i) < \text{W-timestamp}(X) \rightarrow 0$

■ Operation rejected and  $T_i$  rolled back.

■ Otherwise, operation executed.

$$TS(T_1) = 10$$

$$10 < 20 \checkmark$$

$$20 < 0$$

$$20 < 0$$

# Example:

$$T_1 \rightarrow T_3 \rightarrow T_2$$

<u>10</u> T1	<u>30</u> T2	<u>20</u> T3
<u>R(A)</u>		
	<u>W(A)</u>	
<u>W(B)</u>		
		R(B)
		R(C)
	W(C)	

	<u>A</u>	<u>B</u>	<u>C</u>
<u>RTS</u>	<del>0</del> 10	<del>0</del>	<u>0</u>
<u>WTS</u>	<del>0</del> 30	<del>0</del> 10	<u>0</u>

# Example:

10 T1	30 T2	20 T3
R(A)		
	W(A)	
W(B)		
		R(B)
		R(C)
	W(C)	

✓ None ✓ T1

→ 10  
T1

20  
T2

W(A) ✓

Rollback  
R(A)  
0

	A
RTS	0
WTO	20

	A	B	C
RTS	10	20	20
WTS	30	10	30

↗  
↘

# Deadlocks


- ▶ Condition that occurs when two transactions wait for each other to unlock data
  - ▶ T1 needs data items X and Y; T needs Y and X
  - ▶ Each gets the its first piece of data but then waits to get the second (which the other transaction is already holding) - **deadly embrace**
- ▶ Possible only if one of the transactions wants to obtain an exclusive lock on a data item
  - ▶ No deadlock condition can exist among *shared* locks
- ▶ Control through
  - ▶ Prevention
  - ▶ Detection
  - ▶ Avoidance



# How a Deadlock Condition Is Created

TIME	TRANSACTION	REPLY	LOCK STATUS	
			Data X	Data Y
0			Unlocked	Unlocked
1	T1:LOCK(X)	OK	Unlocked	Unlocked
2	T2: LOCK(Y)	OK	Locked	Unlocked
3	T1:LOCK(Y)	WAIT	Locked	Locked
4	T2:LOCK(X)	WAIT	Locked	Locked
5	T1:LOCK(Y)	WAIT	Locked	Locked
6	T2:LOCK(X)	WAIT	Locked	Locked
7	T1:LOCK(Y)	WAIT	Locked	Locked
8	T2:LOCK(X)	WAIT	Locked	Locked
9	T1:LOCK(Y)	WAIT	Locked	Locked
...	.....	.....	.....	.....
...	.....	.....	.....	.....
...	.....	.....	.....	.....
...	.....	.....	.....	.....

Deadlock



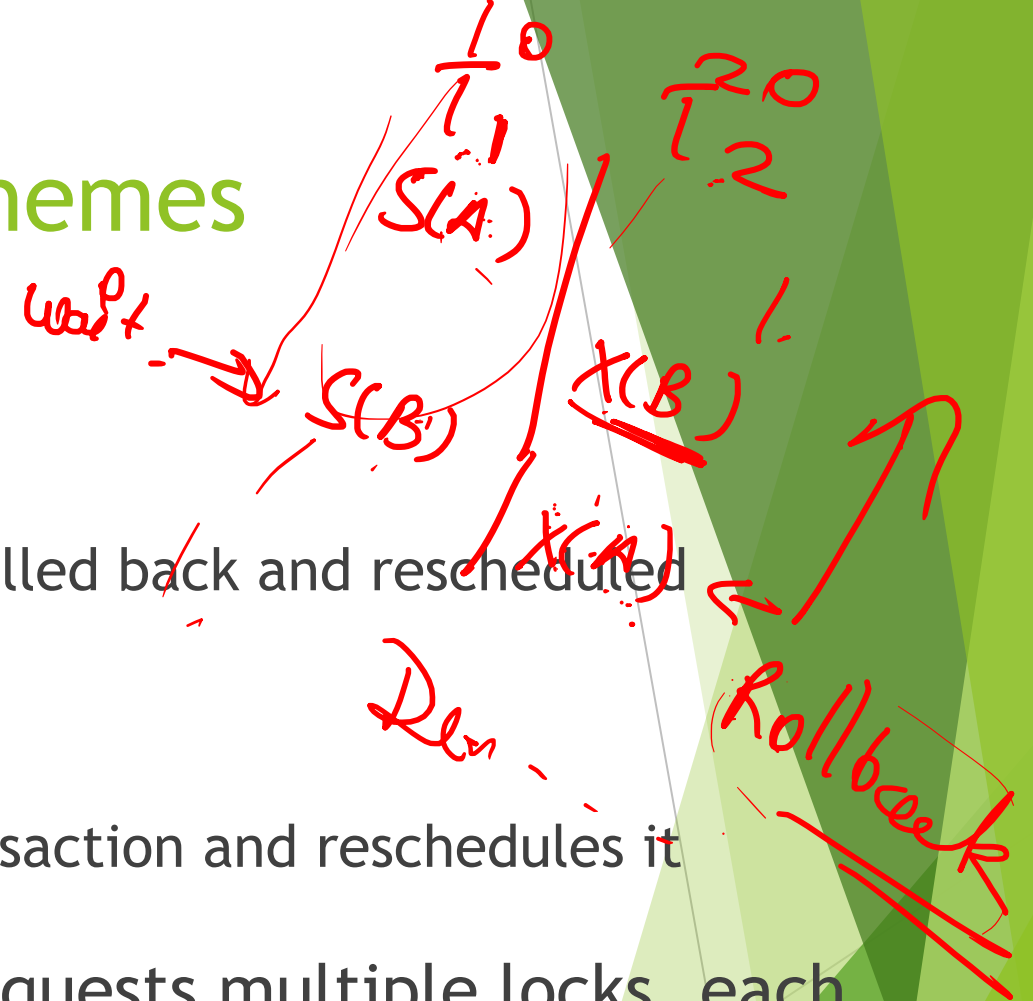
# Wait/Die and Wound/Wait Schemes

## ► Wait/die

- Older transaction waits and the younger is rolled back and rescheduled

## ► Wound/wait

- Older transaction rolls back the younger transaction and reschedules it
- In the situation where a transaction requests multiple locks, each lock request has an associated time-out value. If the lock is not granted before the time-out expires, the transaction is rolled back

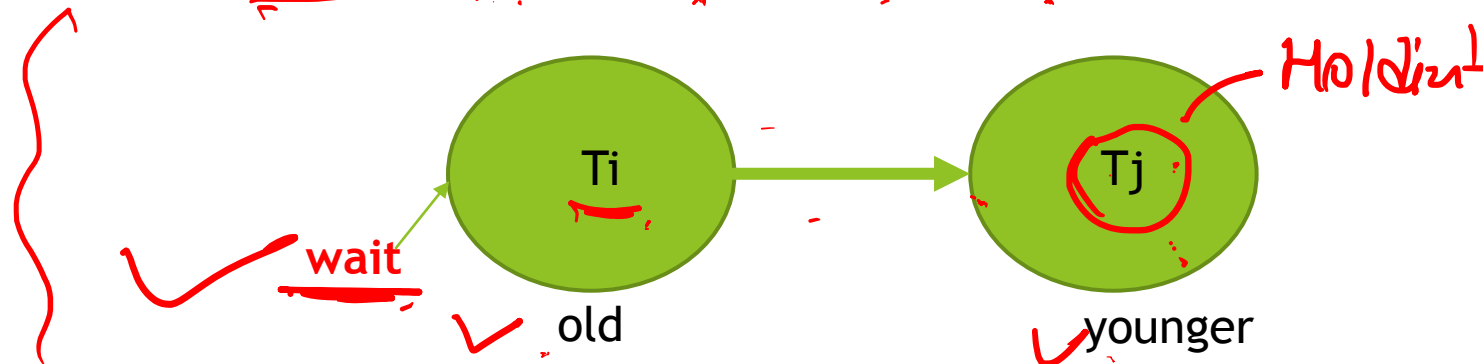


# Wait/Die protocol (non-preemptive)

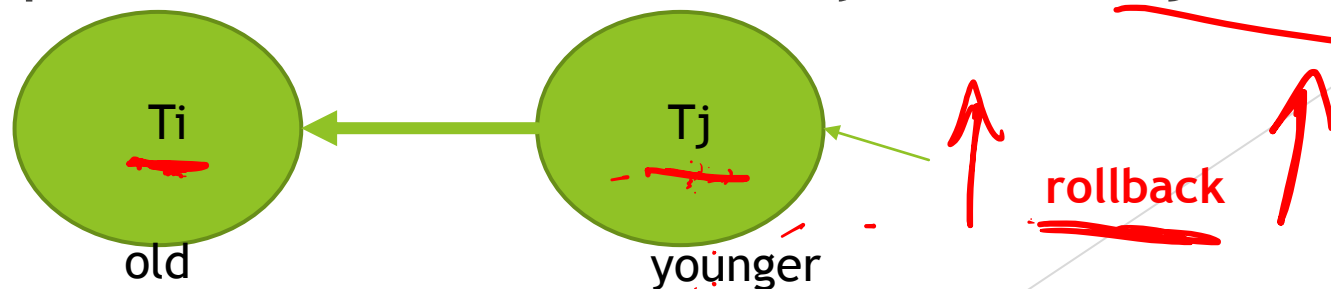
## ► Wait/die

1.  $T_i, T_j$  any transaction in schedule  $S$  such that  $TS(T_i) < TS(T_j)$

Case 1: If  $T_i$  required a resource held by  $T_j$  then  $T_i$  allowed to wait.



Case 2: If  $T_j$  required a resource held by  $T_i$  then  $T_j$  Rollback



And restart with the same time stamp value

Handwritten notes in the top right corner:

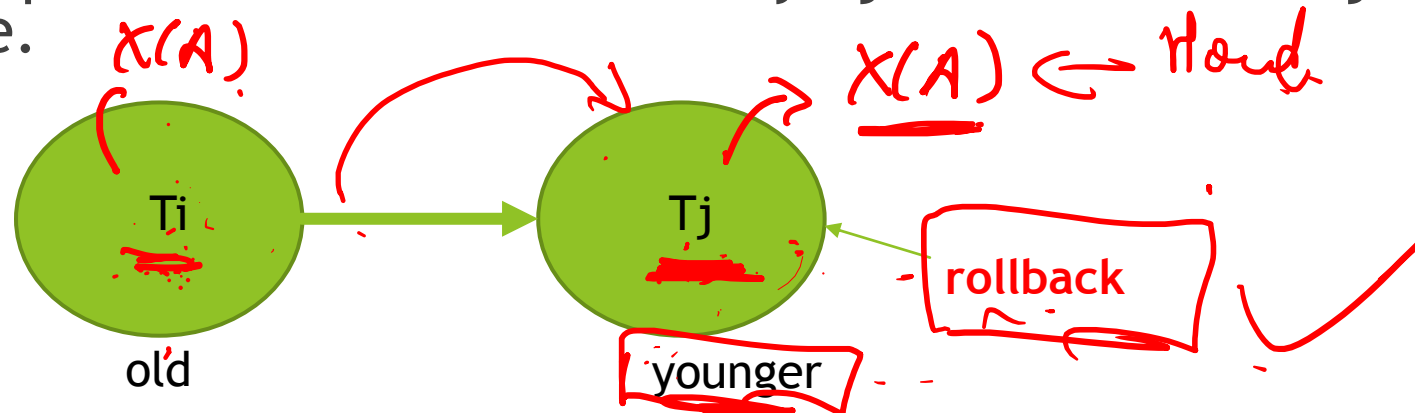
- $0 \frac{10}{1}$
- $\frac{20}{2}$
- $X(A)$
- $X(A)$
- Rollback

# ✓ Wound/Wait protocol (preemptive) ✓

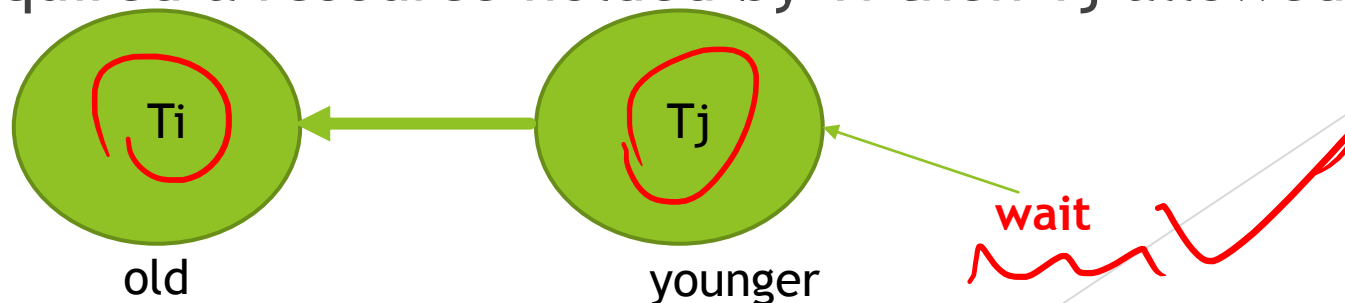
## ► Wait/die

1.  $T_i, T_j$  any transaction in schedule  $S$  such that  $\underline{TS(T_i)} < \underline{TS(T_j)}$

Case 1: If  $T_i$  required a resource holded by  $T_j$  then Rollback  $T_j$  and restart with the same value.



Case 2: If  $T_j$  required a resource holded by  $T_i$  then  $T_j$  allowed to wait.



# Wait/Die and Wound/Wait Concurrency Control Schemes

$$TS(T_2) < TS(T_1)$$

TRANSACTION REQUESTING LOCK	TRANSACTION OWNING LOCK	WAIT/DIE SCHEME	WOUND/WAIT SCHEME
<u>T1 (11548789)</u>	<u>T2 (19562545)</u>	<ul style="list-style-type: none"> <li>T1 waits until T2 is completed and T2 releases its locks.</li> </ul>	<ul style="list-style-type: none"> <li>T1 preempts (rolls back) T2. T2 is rescheduled using the same time stamp.</li> </ul>
<u>T2 (19562545)</u>	T1 (11548789)	<ul style="list-style-type: none"> <li>T2 Dies (rolls back).</li> <li>T2 is rescheduled using the same time stamp.</li> </ul>	<ul style="list-style-type: none"> <li>T2 waits until T1 is completed and T1 releases its locks.</li> </ul>

wait!

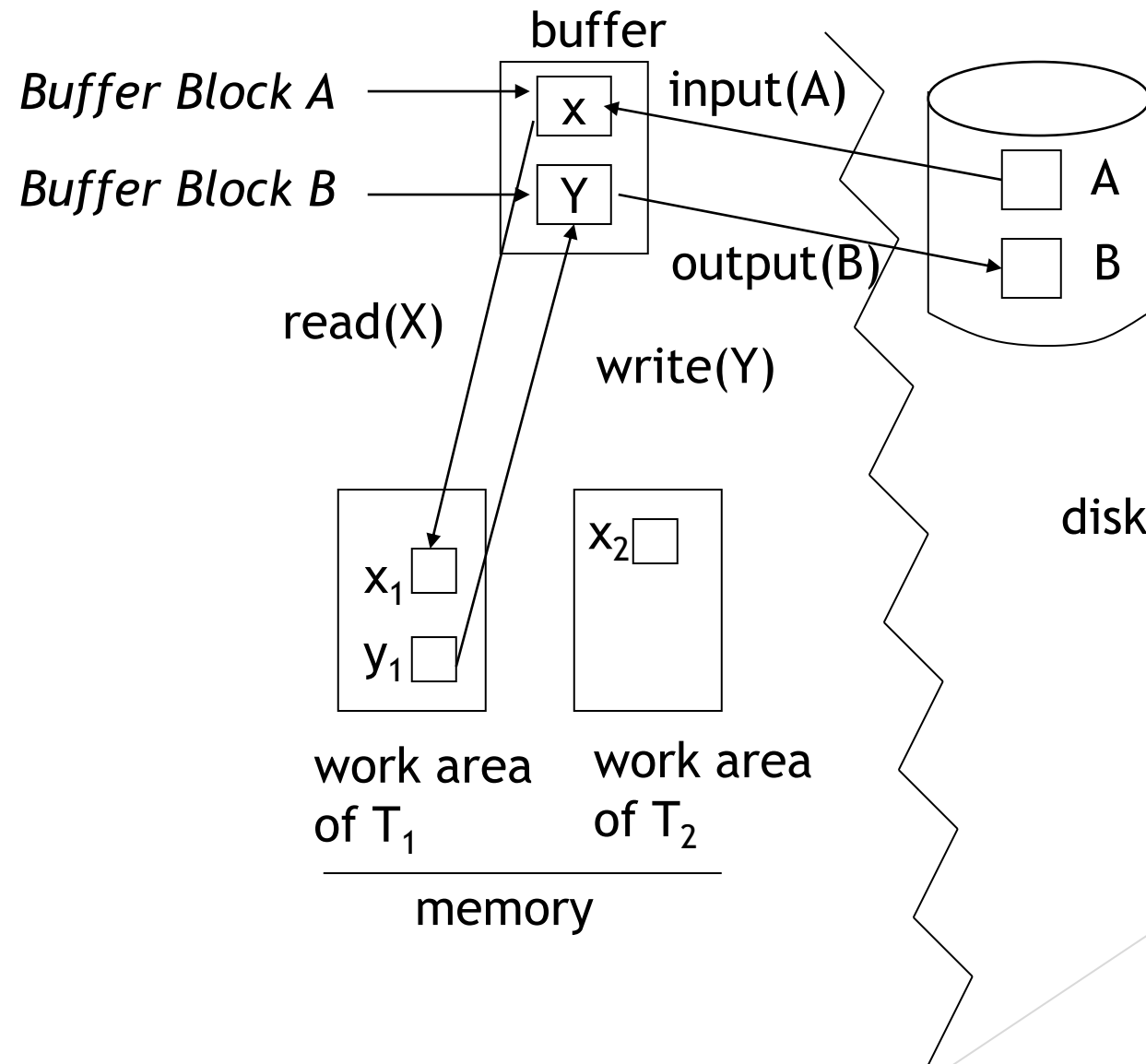
# Recovery

- ▶ Transactions should be durable, but we cannot prevent all sorts of failures:
  - ▶ System crashes
  - ▶ Power failures
  - ▶ Disk crashes
  - ▶ User mistakes
  - ▶ Natural disasters
- ▶ Prevention is better than cure
  - ▶ Reliable OS
  - ▶ Security
  - ▶ UPS and surge protectors
- ▶ Can't protect against everything though

# System Failures

- ▶ A system failure means all running transactions are affected
  - ▶ Software crashes
  - ▶ Power failures
- ▶ The physical media (disks) are not damaged
- ▶ At various times a DBMS takes a checkpoint
  - ▶ All committed transactions are written to disk
  - ▶ A record is made (on disk) of the transactions that are currently running

# Example of Data Access





# Log-Based Recovery

- ▶ A **log** is kept on stable storage.
  - ▶ The log is a sequence of **log records**, and maintains a record of update activities on the database.
- ▶ When transaction  $T_i$  starts, it registers itself by writing a  $\langle T_i \text{ start} \rangle$  log record
- ▶ *Before*  $T_i$  executes **write**( $X$ ), a log record  $\langle T_i, X, V_1, V_2 \rangle$  is written, where  $V_1$  is the value of  $X$  before the write, and  $V_2$  is the value to be written to  $X$ .
  - ▶ Log record notes that  $T_i$  has performed a write on data item  $X_j$ .  $X_j$  had value  $V_1$  before the write, and will have value  $V_2$  after the write.
- ▶ When  $T_i$  finishes its last statement, the log record  $\langle T_i \text{ commit} \rangle$  is written.
- ▶ We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- ▶ Two approaches using logs
  - ▶ Deferred database modification
  - ▶ Immediate database modification

# Example:

**<To start>**  
**<To A, 1000, 950>**  
**<To B, 2000, 2050>**  
**<To commit>**  
**<T1 start>**  
**<T1 C, 700, 600>**  
**<Ti commit>**

# Deferred Database Modification

- The deferred modification technique occurs if the transaction does not modify the database until it has committed.
- In this method, all the logs are created and stored in the stable storage, and the database is updated when a transaction commits
- Below we show the log as it appears at three instances of time.

$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$
$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$
	$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$
	$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$
	$\langle T_1, C, 600 \rangle$	$\langle T_1, C, 600 \rangle$
		$\langle T_1 \text{ commit} \rangle$
(a)	(b)	(c)

- If log on stable storage at time of crash is as in case:

(a) No redo actions need to be taken

(b) redo( $T_0$ ) must be performed since  $\langle T_0 \text{ commit} \rangle$  is present

(c) redo( $T_0$ ) must be performed followed by redo( $T_1$ ) since  
 $\langle T_0 \text{ commit} \rangle$  and  $\langle T_1 \text{ commit} \rangle$  are present

# Immediate DB Modification Recovery Example

- ▶ The Immediate modification technique occurs if database modification occurs while the transaction is still active.
- ▶ In this technique, the database is modified immediately after every operation. It follows an actual database modification.

Below we show the log as it appears at three instances of time.

$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
$\langle T_0, A, 1000, 950 \rangle$	$\langle T_0, A, 1000, 950 \rangle$	$\langle T_0, A, 1000, 950 \rangle$
$\langle T_0, B, 2000, 2050 \rangle$	$\langle T_0, B, 2000, 2050 \rangle$	$\langle T_0, B, 2000, 2050 \rangle$
	$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$
	$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$
	$\langle T_1, C, 700, 600 \rangle$	$\langle T_1, C, 700, 600 \rangle$
		$\langle T_1 \text{ commit} \rangle$
(a)	(b)	(c)

Recovery actions in each case above are:

- (a) undo ( $T_0$ ): B is restored to 2000 and A to 1000.
- (b) undo ( $T_1$ ) and redo ( $T_0$ ): C is restored to 700, and then A and B are set to 950 and 2050 respectively.
- (c) redo ( $T_0$ ) and redo ( $T_1$ ): A and B are set to 950 and 2050 respectively. Then C is set to 600

# Checkpoints

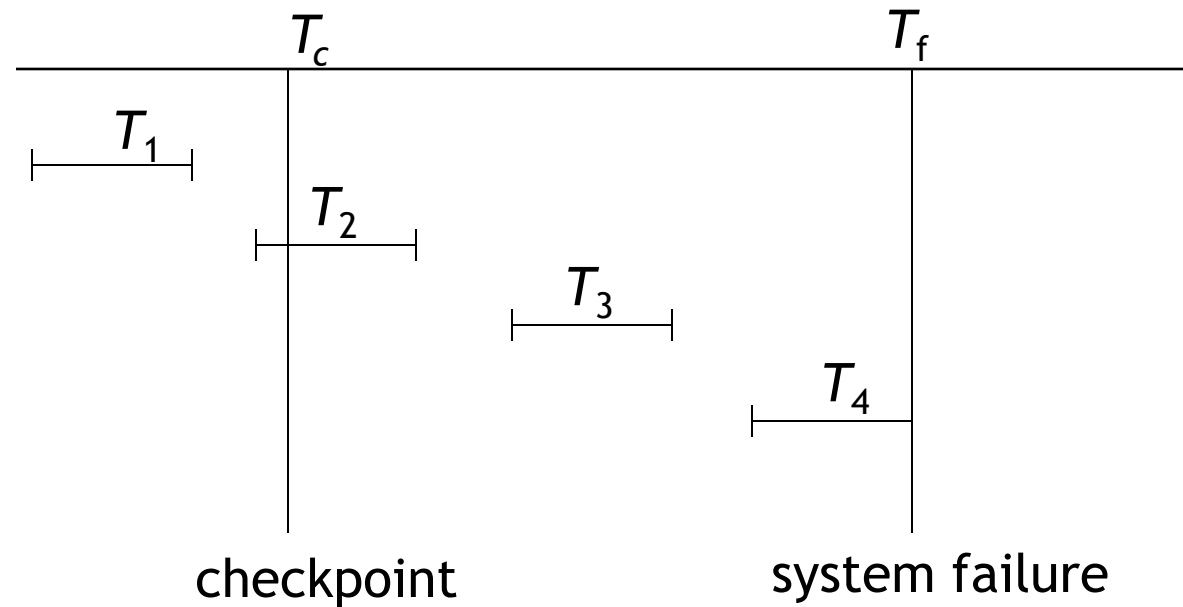
- ▶ Problems in recovery procedure as discussed earlier :
- ▶ searching the entire log is time-consuming
- ▶ we might unnecessarily redo transactions which have already
- ▶ output their updates to the database.
- ▶ Streamline recovery procedure by periodically performing checkpointing
- ▶ Output all log records currently residing in main memory onto stable storage.
- ▶ Output all modified buffer blocks to the disk.
- ▶ Write a log record < checkpoint> onto stable storage.

# Checkpoints (Cont.)

► During recovery we need to consider only the most recent transaction  $T_i$  that started before the checkpoint, and transactions that started after  $T_i$ .

1. Scan backwards from end of log to find the most recent <checkpoint> record
2. Continue scanning backwards till a record < $T_i$  start> is found.
3. Need only consider the part of log following above start record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
4. For all transactions (starting from  $T_i$  or later) with no < $T_i$  commit>, execute  $\text{undo}(T_i)$ . (Done only in case of immediate modification.)
5. Scanning forward in the log, for all transactions starting from  $T_i$  or later with a < $T_i$  commit>, execute  $\text{redo}(T_i)$ .

# Example of Checkpoints



- ▶  $T_1$  can be ignored (updates already output to disk due to checkpoint)
- ▶  $T_2$  and  $T_3$  redone.
- ▶  $T_4$  undone

# Transaction Recovery Method

UNDO and REDO: lists of transactions

UNDO = all transactions running at the last checkpoint

REDO = empty

For each entry in the log, starting at the last checkpoint

- If a BEGIN TRANSACTION entry is found for T

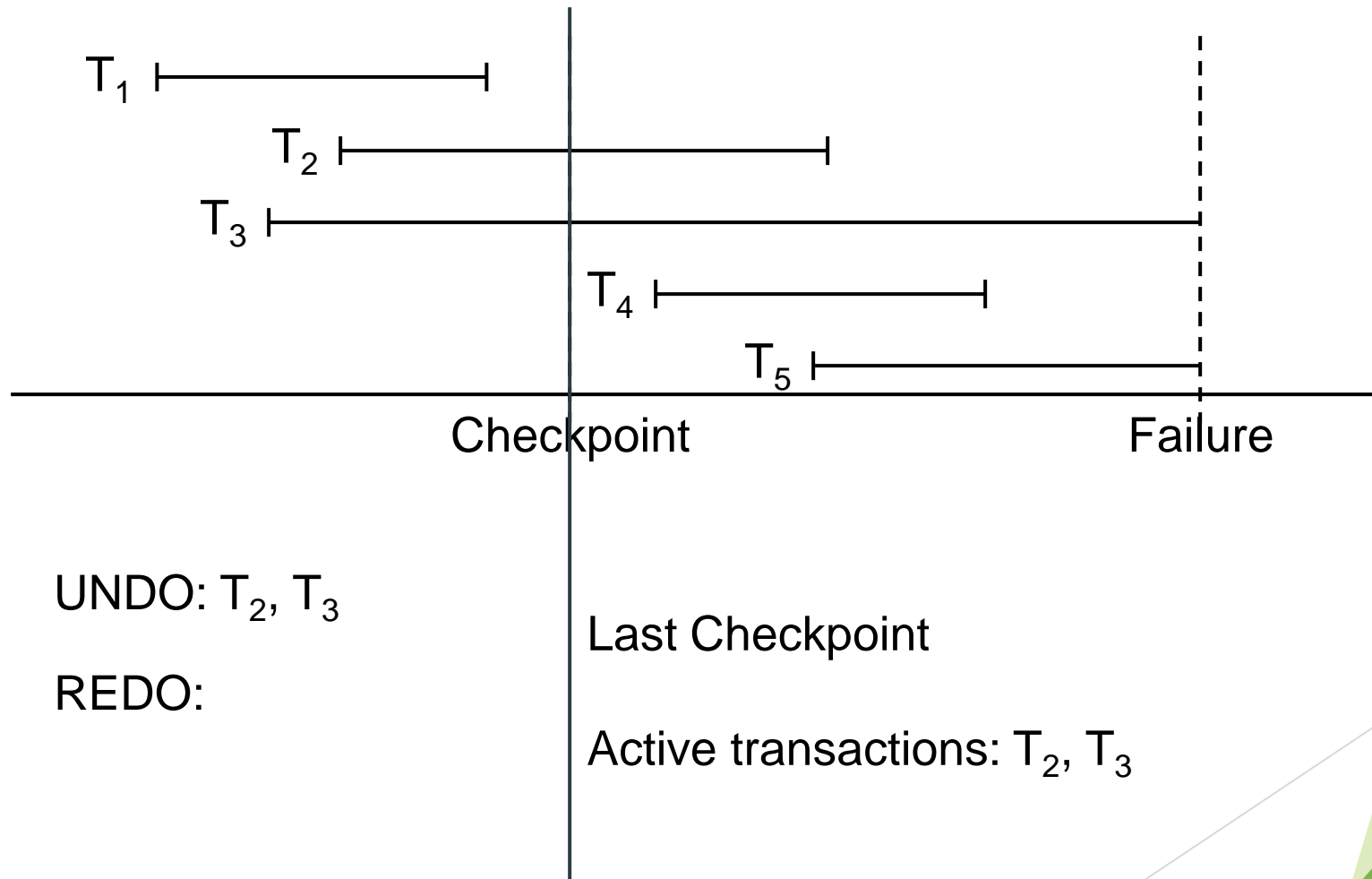
  - Add T to UNDO

- If a COMMIT entry is found for T

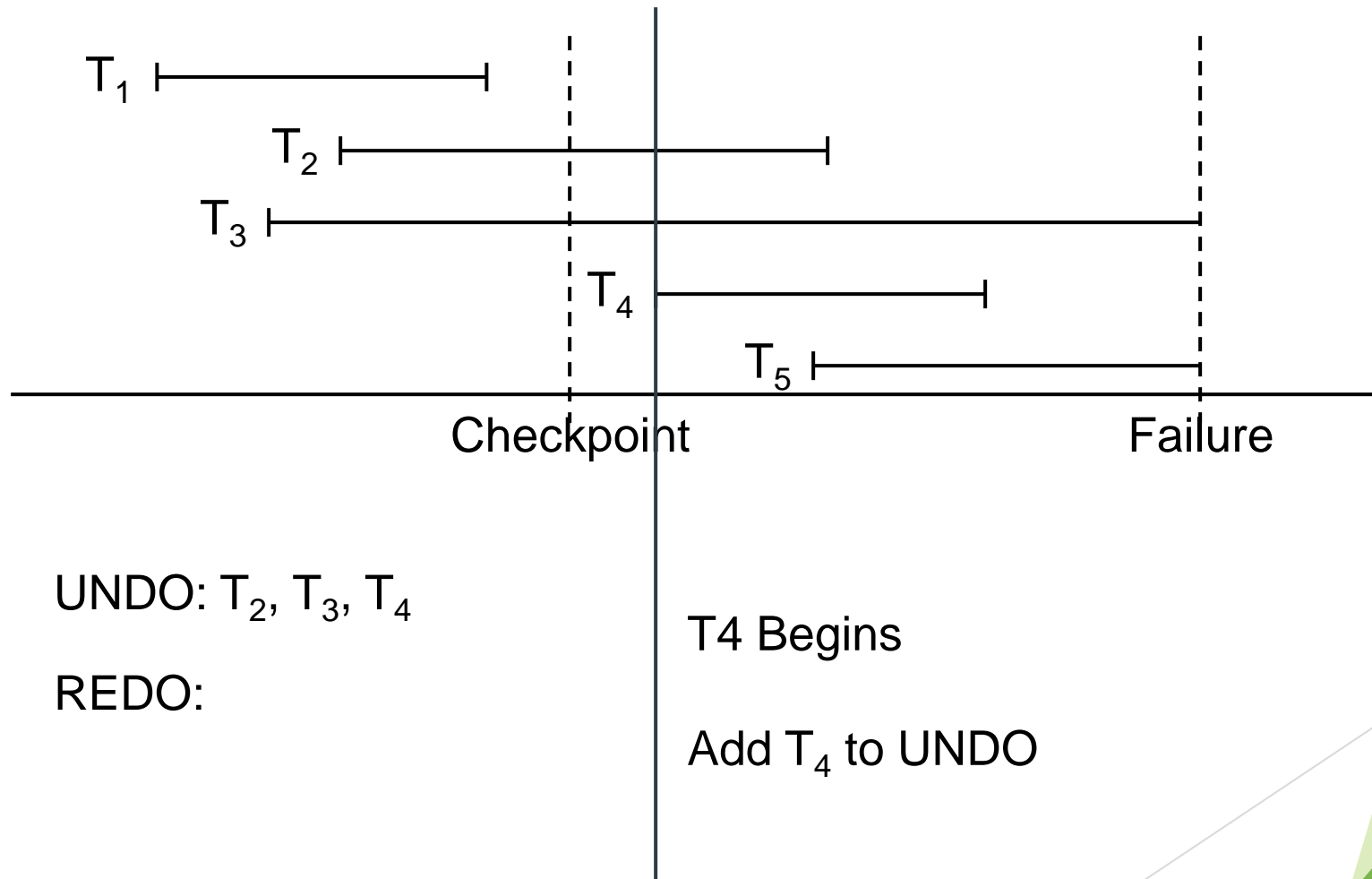
  - Move T from UNDO to REDO



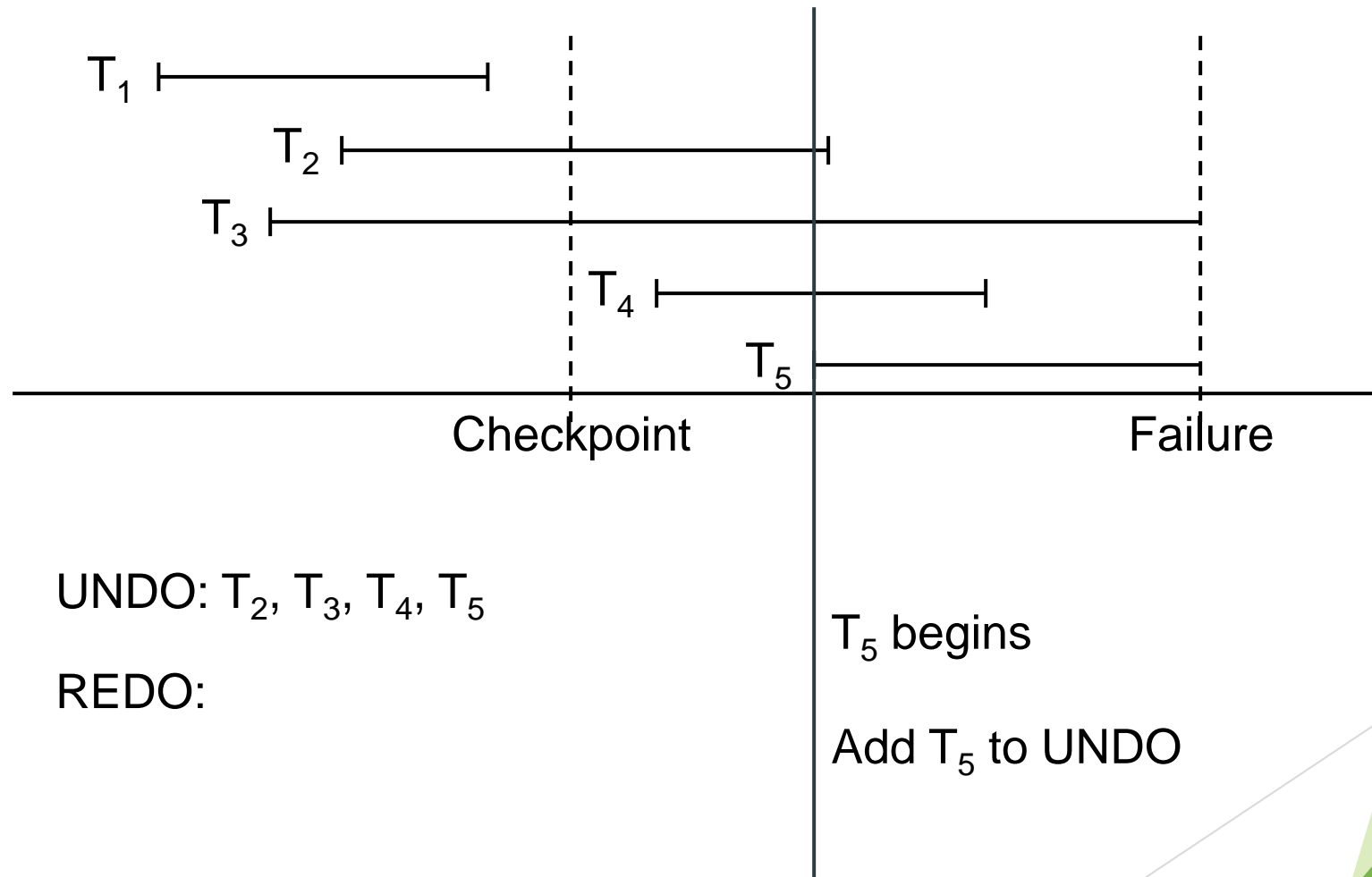
# Transaction Recovery



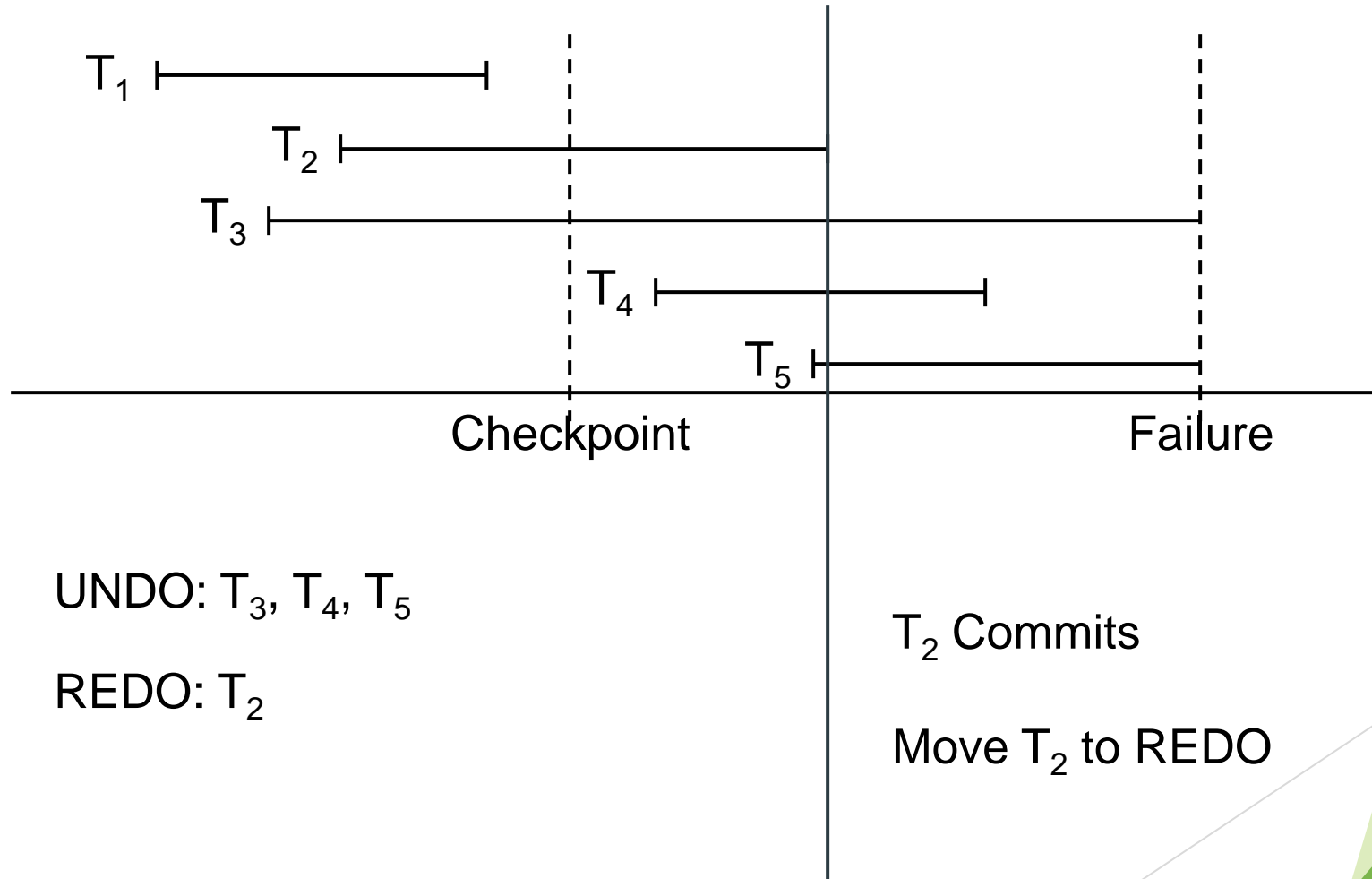
# Transaction Recovery



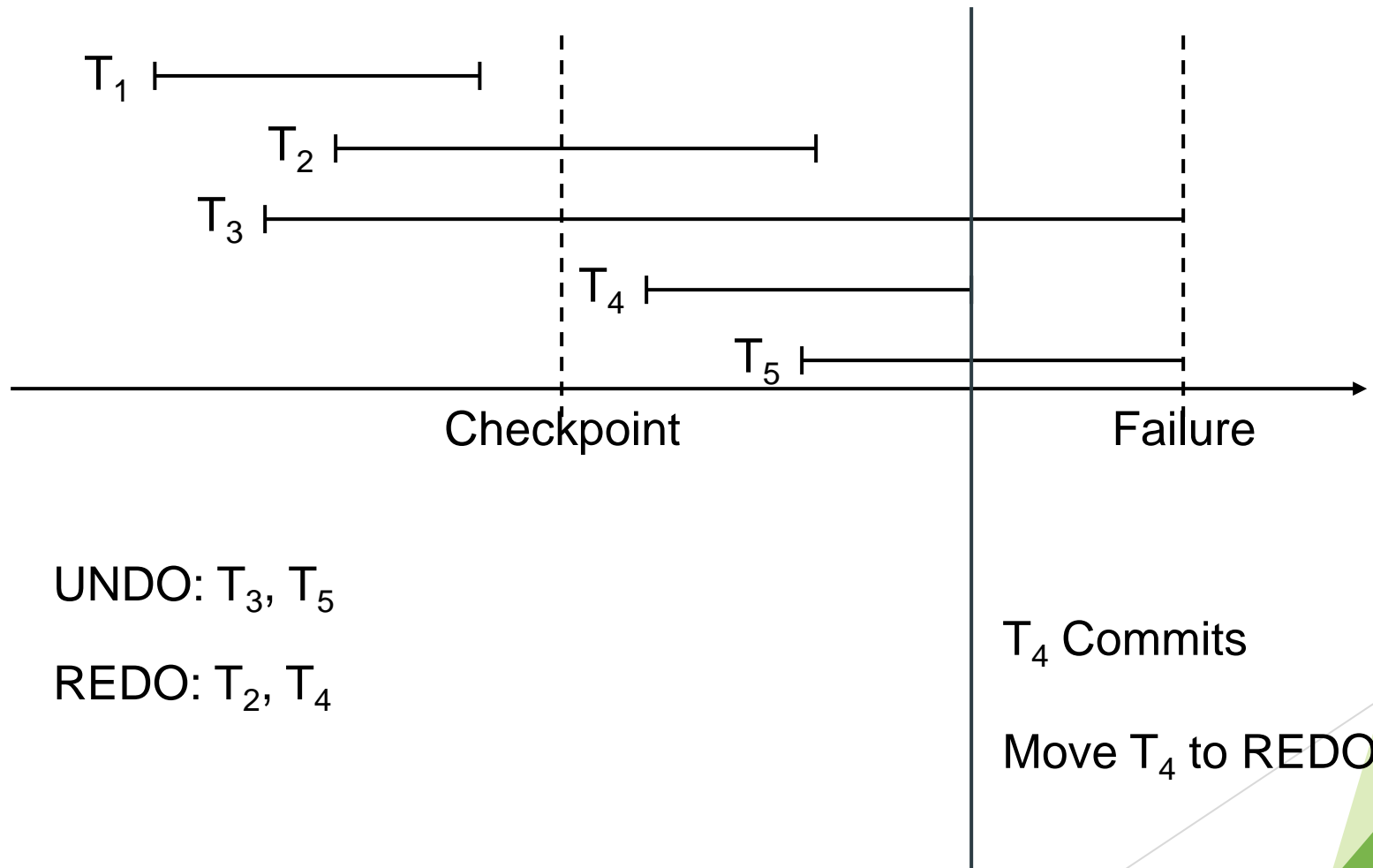
# Transaction Recovery



# Transaction Recovery



# Transaction Recovery



# Forwards and Backwards

## ► Backwards recovery

- We need to undo some transactions
- Working backwards through the log we undo any operation by a transaction on the UNDO list
- This returns the database to a consistent state

## ► Forwards recovery

- Some transactions need to be redone
- Working forwards through the log we redo any operation by a transaction on the REDO list
- This brings the database up to date

# Media Failures

- ▶ System failures are not too severe
  - ▶ Only information since the last checkpoint is affected
  - ▶ This can be recovered from the transaction log
- ▶ Media failures (disk crashes etc) are more serious
  - ▶ The data stored to disk is damaged
  - ▶ The transaction log itself may be damaged

# Backups

- ▶ Backups are needed to recover from media failure
  - ▶ The transaction log and entire contents of the database is written to secondary storage (often tape)
  - ▶ Time consuming, and often requires down time
- ▶ Backups frequency
  - ▶ Frequent enough that little information is lost
  - ▶ Not so frequent as to cause problems
  - ▶ Every day (night) is common
- ▶ Backup storage



# Recovery from Media Failure

- ▶ Restore the database from the last backup
- ▶ Use the transaction log to redo any changes made since the last backup
- ▶ If the transaction log is damaged you can't do step 2
  - ▶ Store the log on a separate physical device to the database
  - ▶ The risk of losing both is then reduced

# References

- ▶ Advanced Database Systems, Shivrath Babu, Department of Computer Science, Duke University Durham, NC 27708
- ▶ Database System Concepts, Silberschatz, Korth and Sudarshan, 6<sup>th</sup> Ed.