Transaction Management

Content

Transaction Concepts

A ***transaction*** is a *unit* of program execution that accesses and possibly updates various data items.

A transaction must see a consistent database.

During transaction execution the database may be inconsistent.

When the transaction is committed, the database must be consistent.

Transaction Concepts (cont…)

If the transaction aborted, the DB must be restored to its prior state. Means such transaction must be undone or rolled back

Two main issues to deal with:

- Failures of various kinds, such as hardware failures and system crashes
- Concurrent execution of multiple transactions

ACID Properties

To preserve integrity of data, the database system must ensure:

Atomicity.

Either all operations of the transaction are properly reflected in the database or none are.

Consistency

Execution of a transaction in isolation preserves the consistency of the database.

ACID Properties (cont...)

Isolation.

Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

- That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.

ACID Properties (cont…)

Durability.

After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Trans. Mgt. with SQL

COMMIT statement – ends the SQL trans.; effects permanently recorded within DB

ROLLBACK statement – DB is rolled back to its previous consistent state and all the changes are aborted

Reach end of the program successfully – similar to COMMIT

Program abnormally terminated – similar to ROLLBACK

Transaction Log

Keep track of all transactions that update the DB

If failure occurs, information that was stored here will be used for recovery

It is triggered by ROLL BACK statement, program abnormal termination, or system failure

It stores before-and-after data of the DB and the tables, rows and attribute values that participated in the transaction

Transaction Log (cont…)

The transaction log is subject to dangers such as disk full conditions and disk crashes

It has to be managed like other DBs

Transaction log will increase the processing overhead – but it is worthwhile

Types of log record:

- [start_transaction,T]: Records that transaction T has started execution.
- [write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.
- [read_item,T,X]: Records that transaction T has read the value of database item X.
- [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
- [abort,T]: Records that transaction T has been aborted.

Write ahead log rule (WAL)

- The entry in the log must be made before COMMIT processing can complete

Example of Fund Transfer

Transaction to transfer $50 from account *A* to account *B*:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)
7. *Commit*

Consistency requirement – the sum of *A* and *B* is unchanged by the execution of the transaction.

Atomicity requirement — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

Fund Transfer (cont…)

Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.

Isolation requirement — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database
(the sum *A* + *B* will be less than it should be).
Can be ensured trivially by running transactions ***serially***, that is one after the other. However, executing multiple transactions concurrently has significant benefits (this is not covered in WXES2103)

Transaction state

Active, the initial state; the transaction stays in this state while it is executing

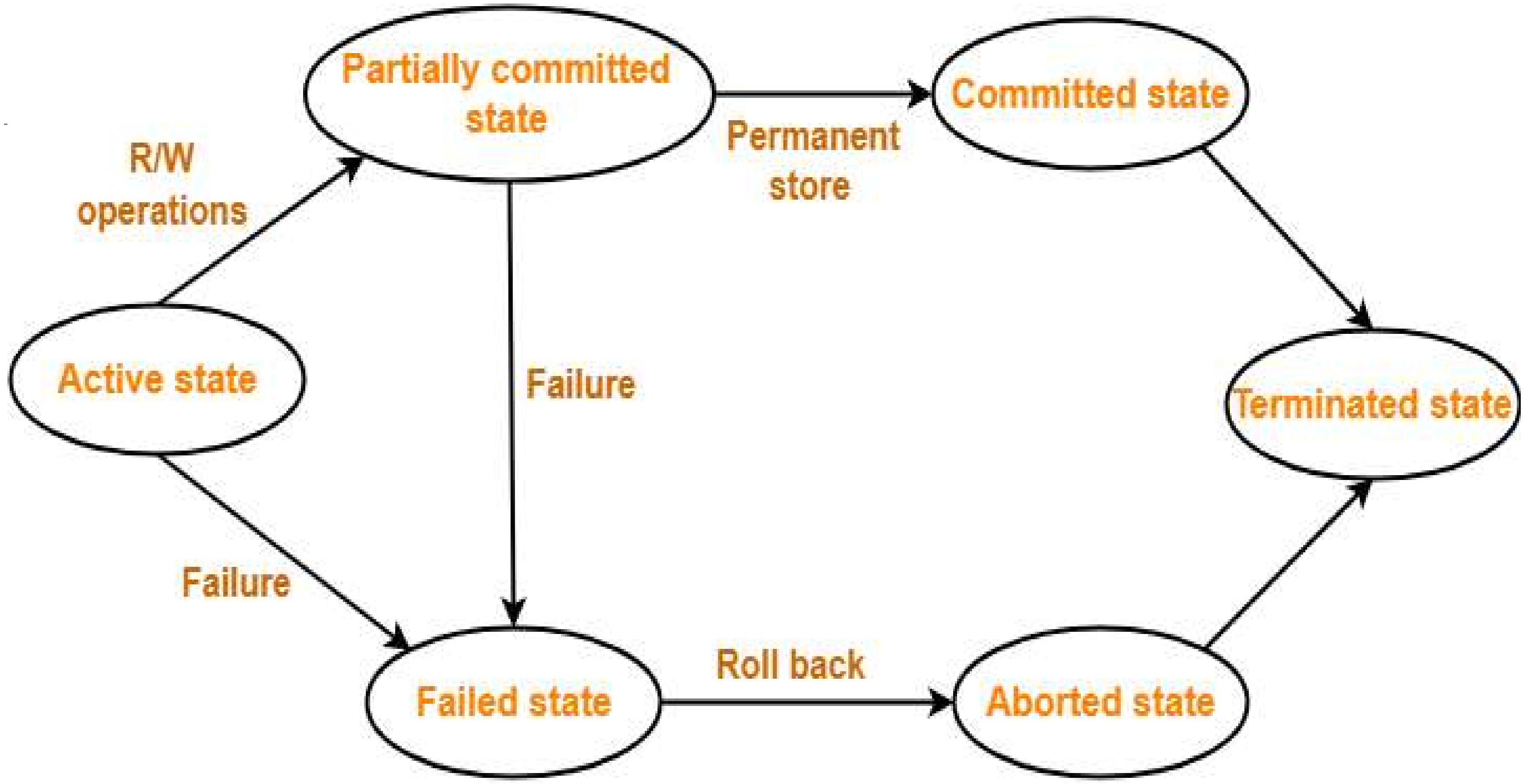
Partially committed, after the final statement has been executed.

Failed, after the discovery that normal execution can no longer proceed.

Aborted, after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:

- restart the transaction – only if no internal logical error
- kill the transaction

Committed, after *successful completion*.



Transaction States in DBMS

Transaction Recovery

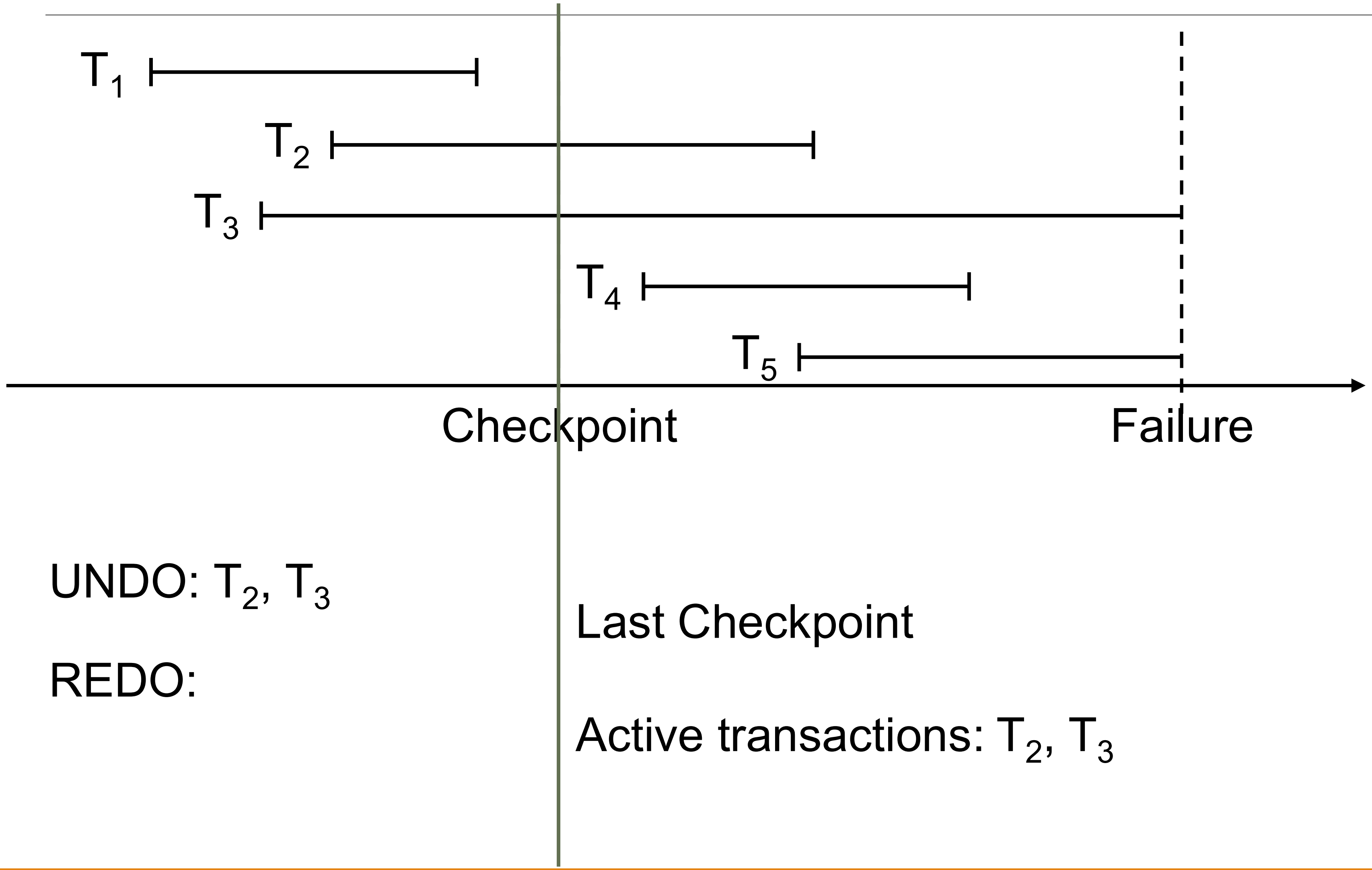
UNDO and REDO: lists of transactions

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



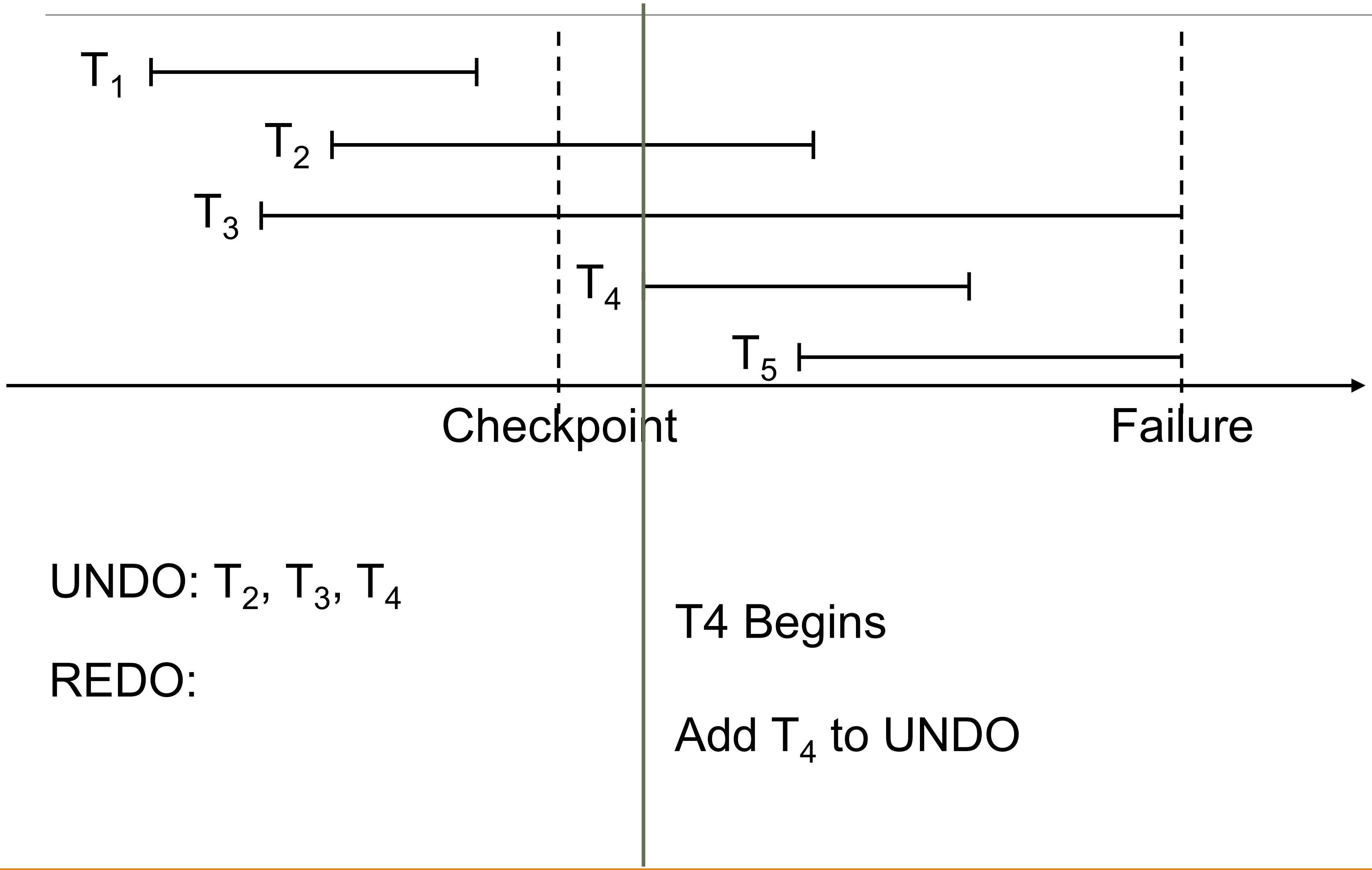
UNDO: T_2, T_3

REDO:

Last Checkpoint

Active transactions: T_2, T_3

Transaction Recovery



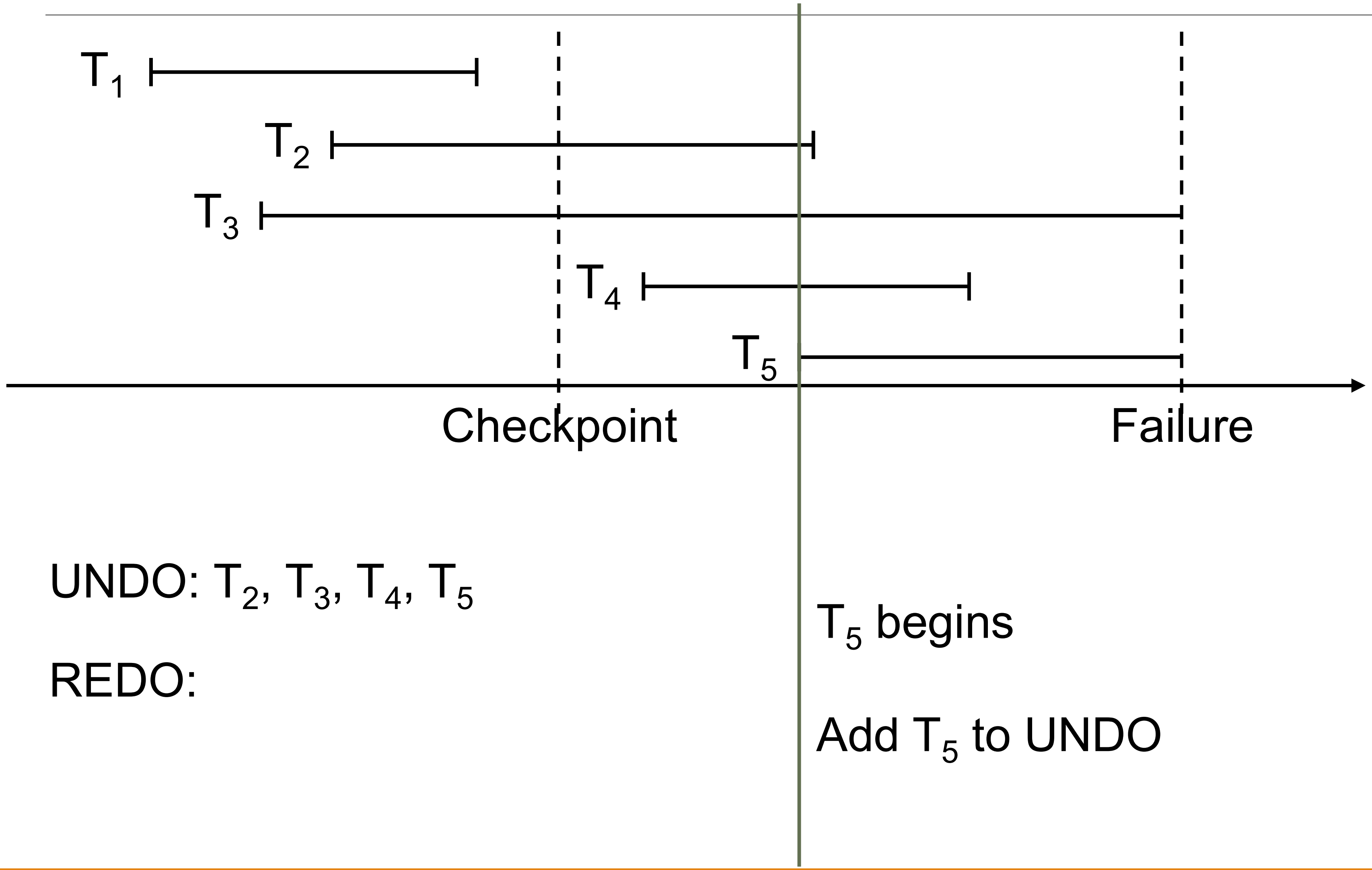
UNDO: T_2, T_3, T_4

REDO:

T4 Begins

Add T_4 to UNDO

Transaction Recovery



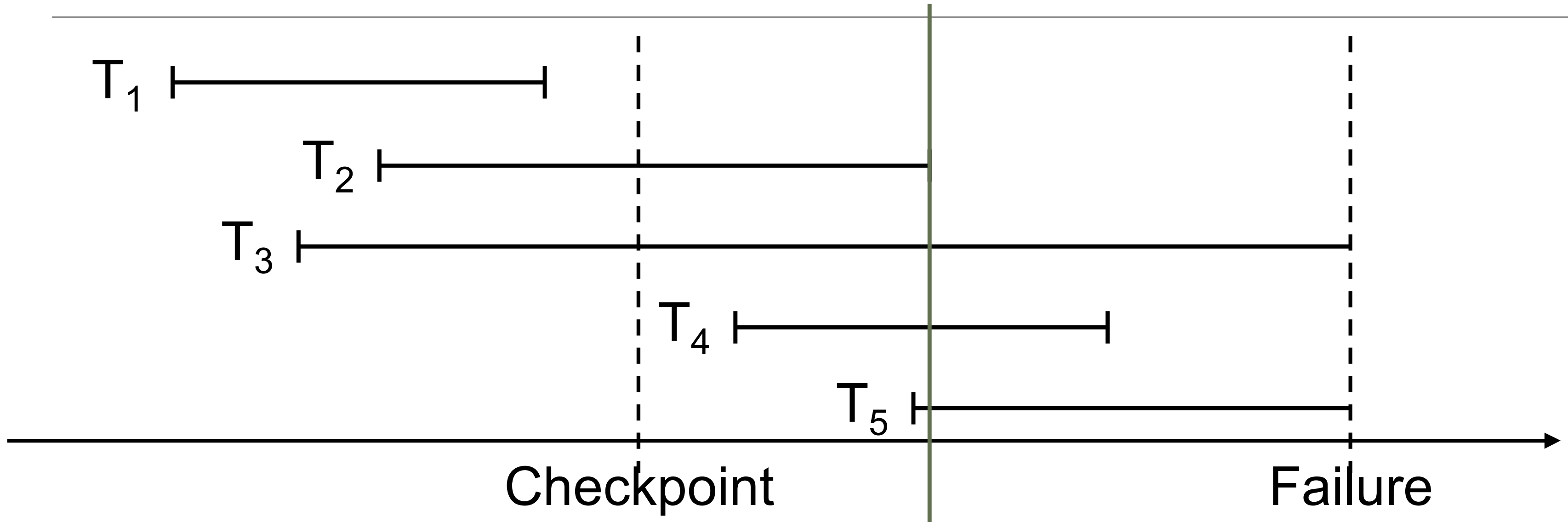
UNDO: T_2, T_3, T_4, T_5

REDO:

T_5 begins

Add T_5 to UNDO

Transaction Recovery



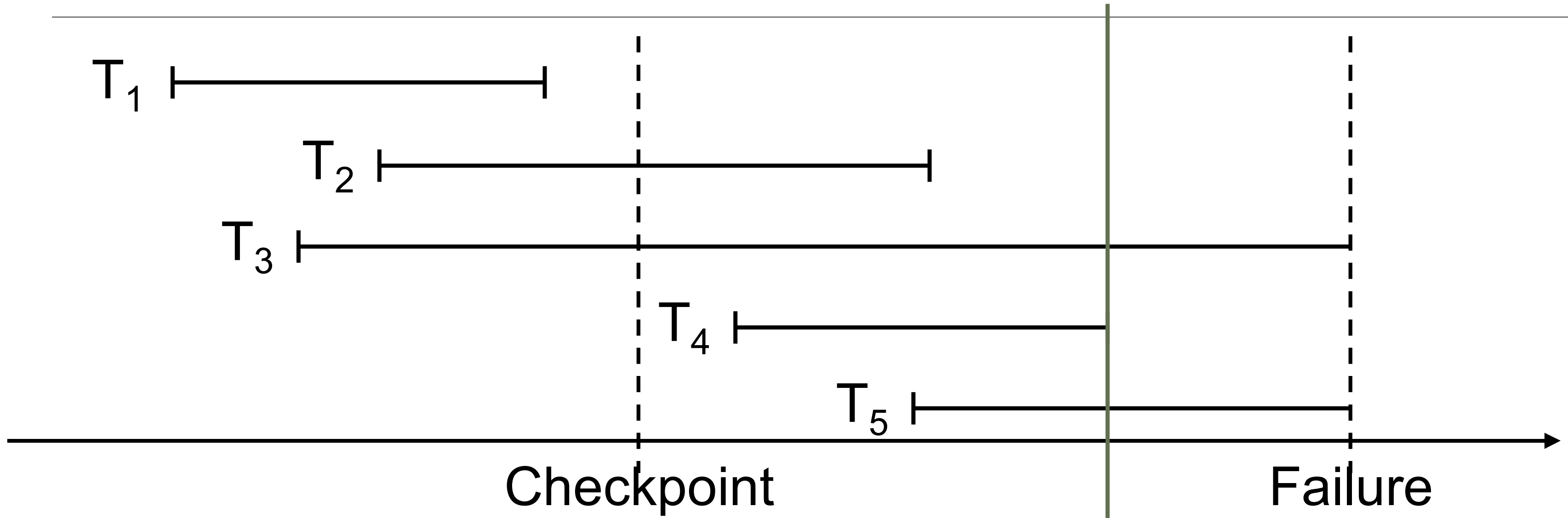
UNDO: T_3, T_4, T_5

REDO: T_2

T_2 Commits

Move T_2 to REDO

Transaction Recovery



UNDO: T_3, T_5

REDO: T_2, T_4

T_4 Commits

Move T_4 to REDO

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

DBMS Transaction Subsystem

Trans. Mgr. coordinates transactions on

behalf of application program. It communicates with scheduler.

Scheduler implements a strategy for concurrency control.

If any failure occurs, ***recovery manager*** handles it.

Buffer manager in charge of transferring data between disk storage and main memory.

DBMS Transaction Subsystem

File manager manipulates the underlying storage files and manages the allocation of storage space on disk.

File manager does not directly manage the physical input and output of data, rather it passes the requests on to the ***access manager***.

Appropriate access method is used to either read or write data into the ***system manager***.

Single-User System:
- At most one user at a time can use the system.

Multiuser System:
- Many users can access the system concurrently.

Concurrency
- **Interleaved processing**:
  - Concurrent execution of processes is interleaved in a single CPU
- **Parallel processing**:
  - Processes are concurrently executed in multiple CPUs.

Concurrency

Large databases are used by many people

- Many transactions to be run on the database
- It is desirable to let them run at the same time as each other
- Need to preserve isolation

If we don’t allow for concurrency then transactions are run sequentially

- Have a queue of transactions
- Long transactions (eg backups) will make others wait for long periods

In order to run transactions concurrently we interleave their operations

Each transaction gets a share of the computing time

This leads to several sorts of problems

- Lost updates
- Uncommitted updates
- Incorrect analysis

All arise because isolation is broken

Purpose of Concurrency

- To enforce Isolation (through mutual exclusion) among conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- To resolve read-write and write-write conflicts.

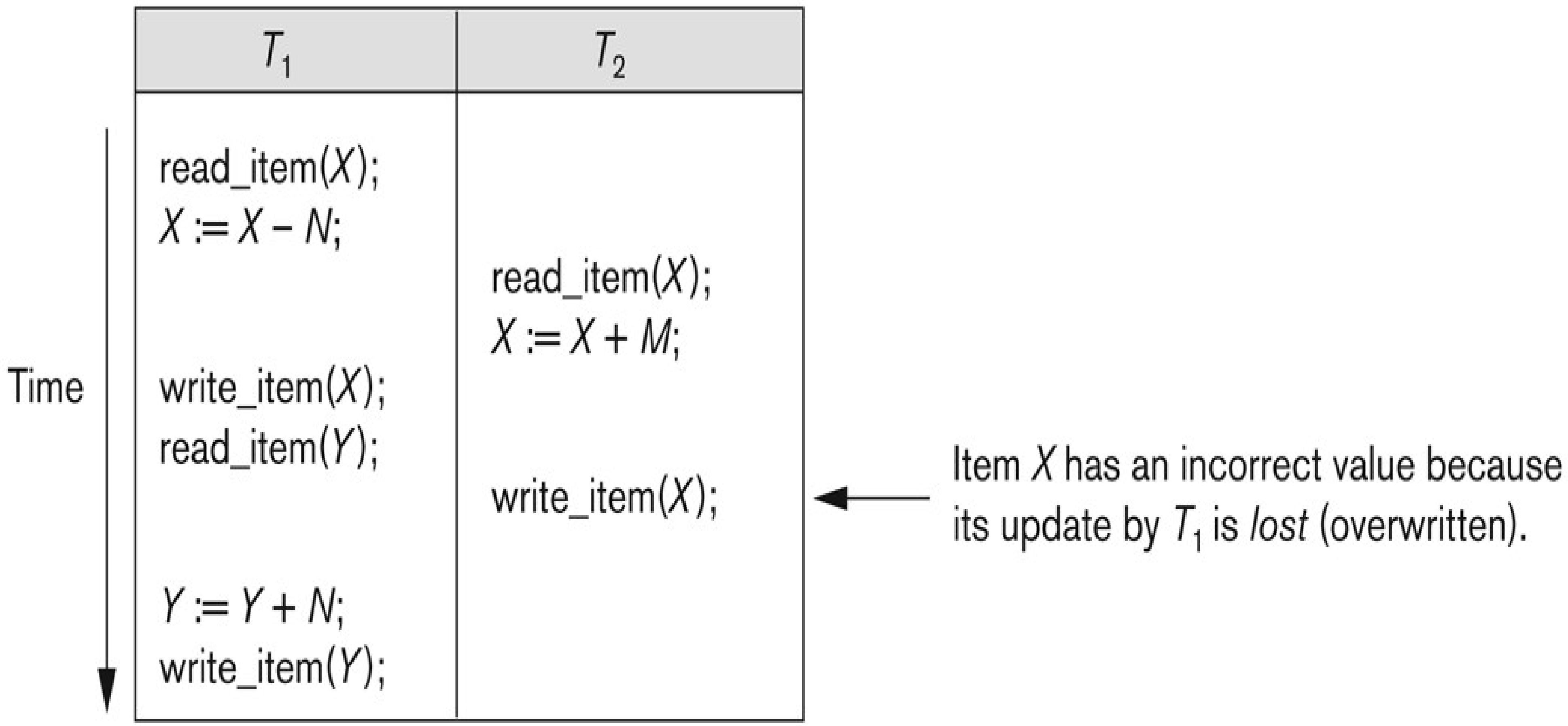
Concurrency Problem

Lost Update

Temporary Update or Dirty Read

Incorrect Summary

Lost Update



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

write_item(X);
read_item(Y);	

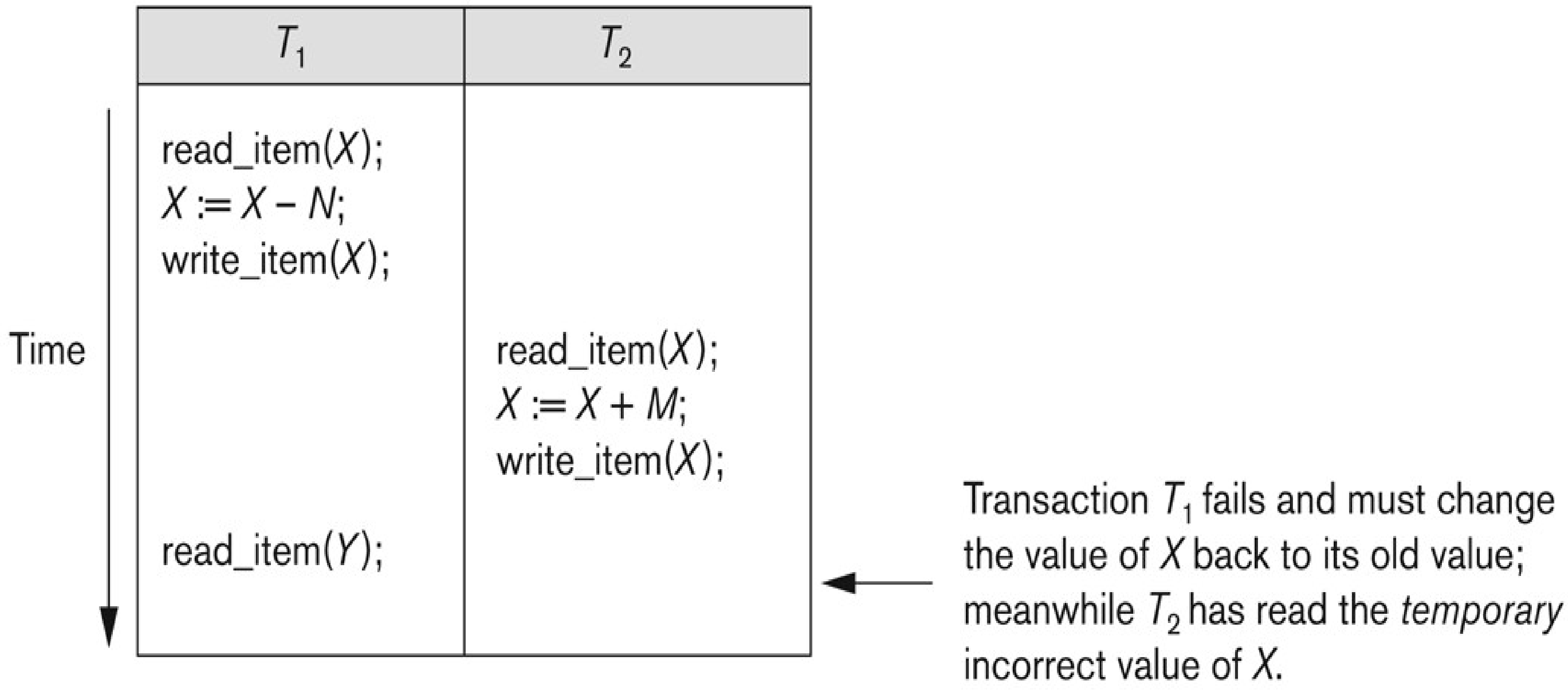
	write_item(X);
$Y := Y + N$; write_item(Y);	

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

The Lost Update Problem

- This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

Dirty Read



Time	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.

The Temporary Update (or Dirty Read) Problem

- This occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 17.1.4).
- The updated item is accessed by another transaction before it is changed back to its original value.

Incorrect Summary

T1	T2
`Read(X)`	
`X = X - 5`	
`Write(X)`	
	`Read(X)`
	`Read(Y)`
	`Sum = X+Y`
`Read(Y)`	
`Y = Y + 5`	
`Write(Y)`	

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

Why **recovery** is needed:

(What causes a Transaction to fail)

1. A computer failure (system crash):

   A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer’s internal memory may be lost.

2. A transaction or system error:

   Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

Why **recovery** is needed (Contd.):

(What causes a Transaction to fail)

3. Local errors or exception conditions detected by the transaction:

   Certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.

   A programmed abort in the transaction causes it to fail.

4. Concurrency control enforcement:

   The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock

Schedule

Transaction schedule or history:

- When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as a transaction schedule (or history).

- It is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1.
- Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

Types of Schedule based on Recoverability

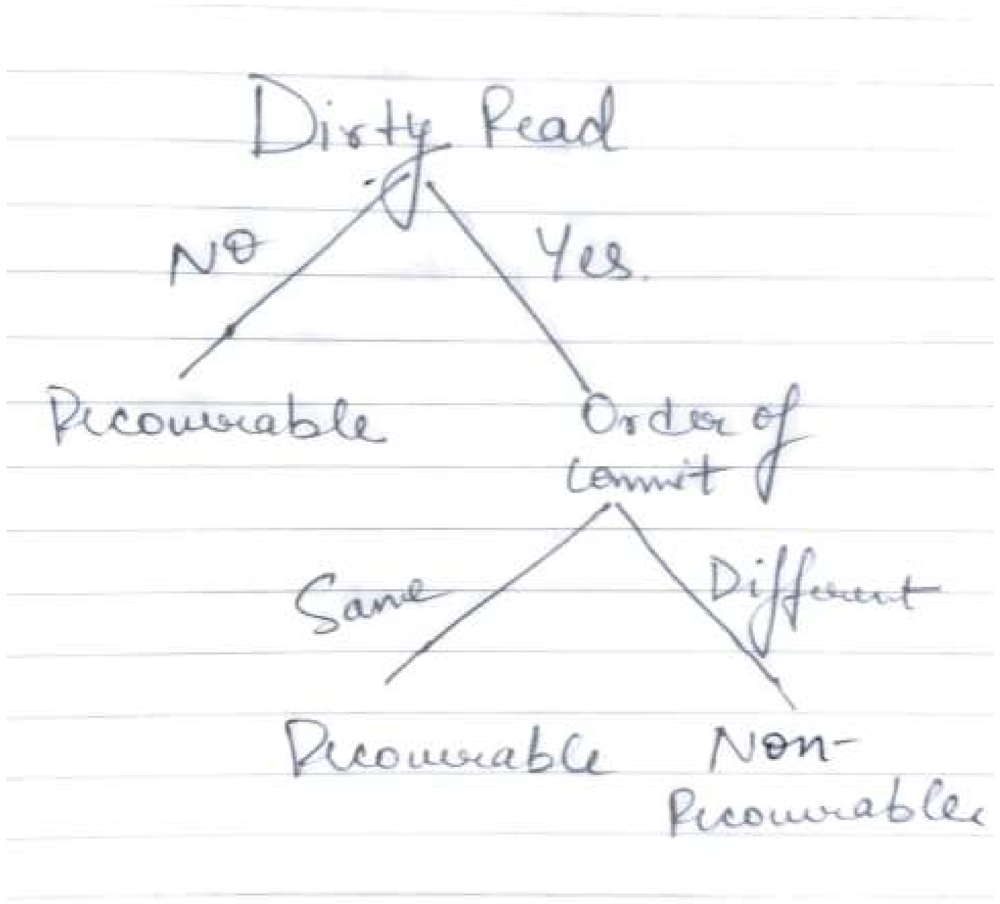
Recoverable schedule:

- A schedule S is recoverable if no transaction T in S commits until all transactions T’ that have written an item that T reads have committed.

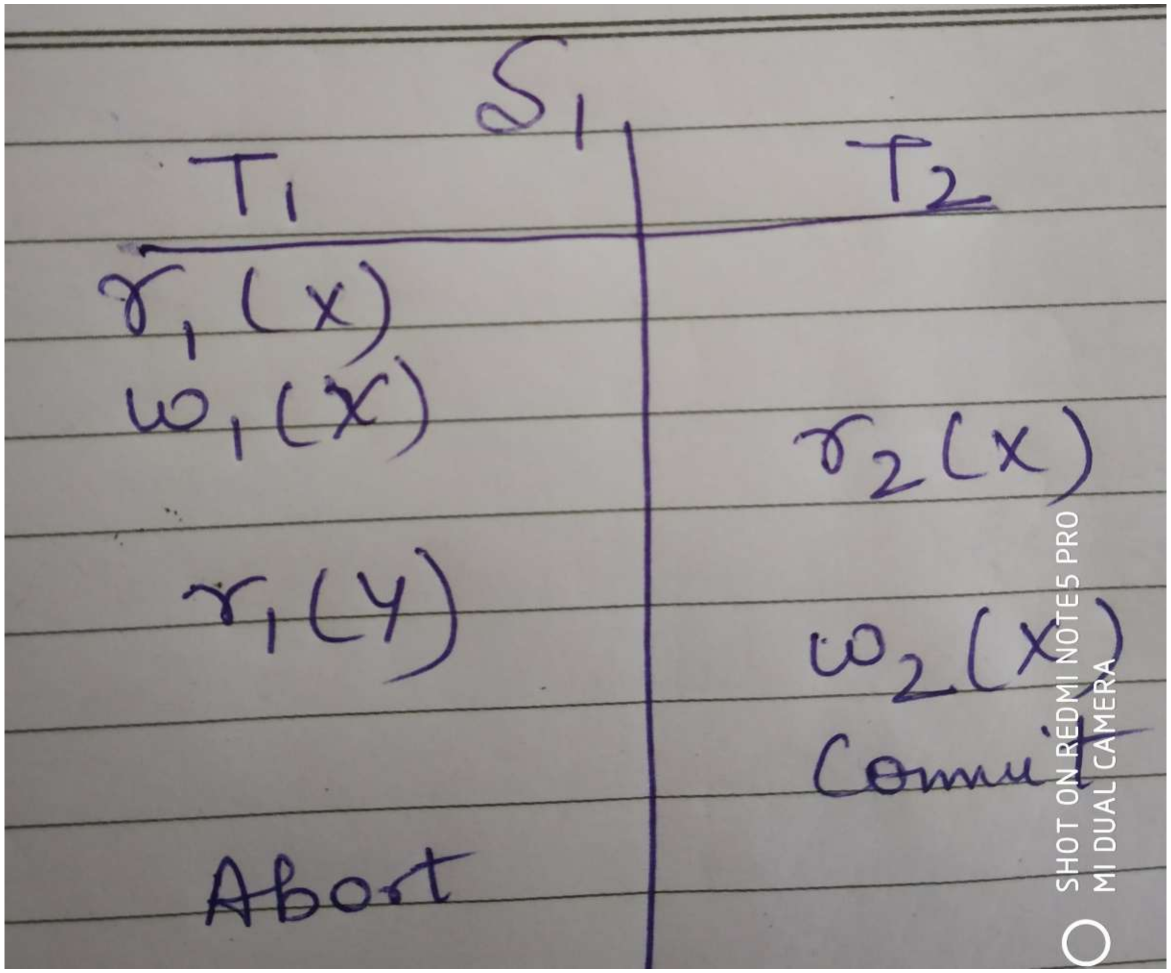
Cascadeless schedule:

- One where every transaction reads only the items that are written by committed transactions.

Method to find Recoverable Schedule



S1- $r_1(X)\ w_1(X)\ r_2(X)\ r_1(Y)\ w_2(X)\ c_2\ a_1$

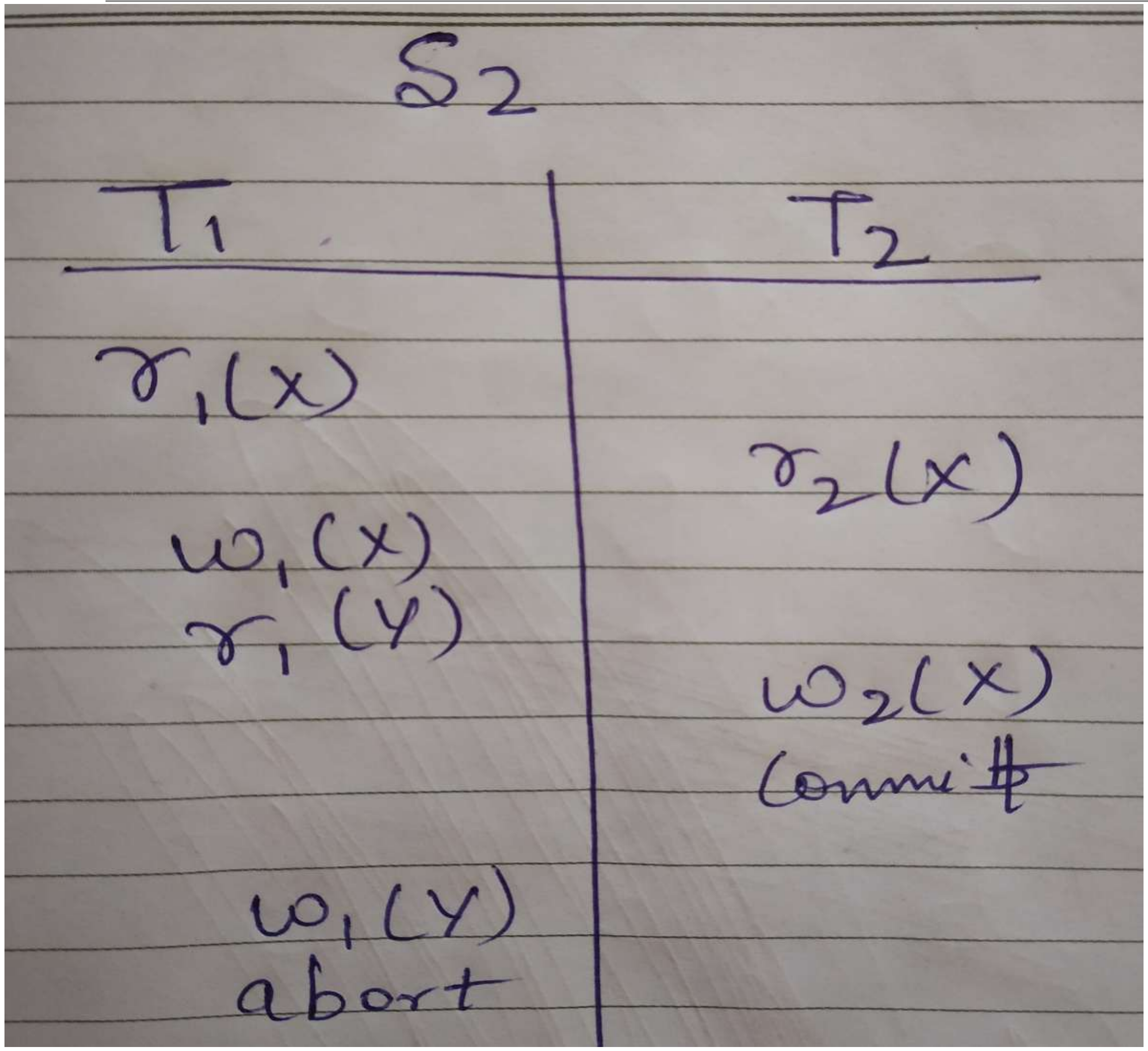


T_1	T_2
$r_1(x)$	
$w_1(x)$	
	$r_2(x)$
$r_1(Y)$	
	$w_2(x)$
	Commit
Abort	

SHOT ON REDMI NOTE5 PRO
MI DUAL CAMERA

Non-Recoverable (T_2 must be rolled back when T_1 aborts)

S2 - $r_1(X)\ r_2(X)\ w_1(X)\ r_1(Y)\ w_2(X)\ c_2\ w_1(Y)\ a_1$

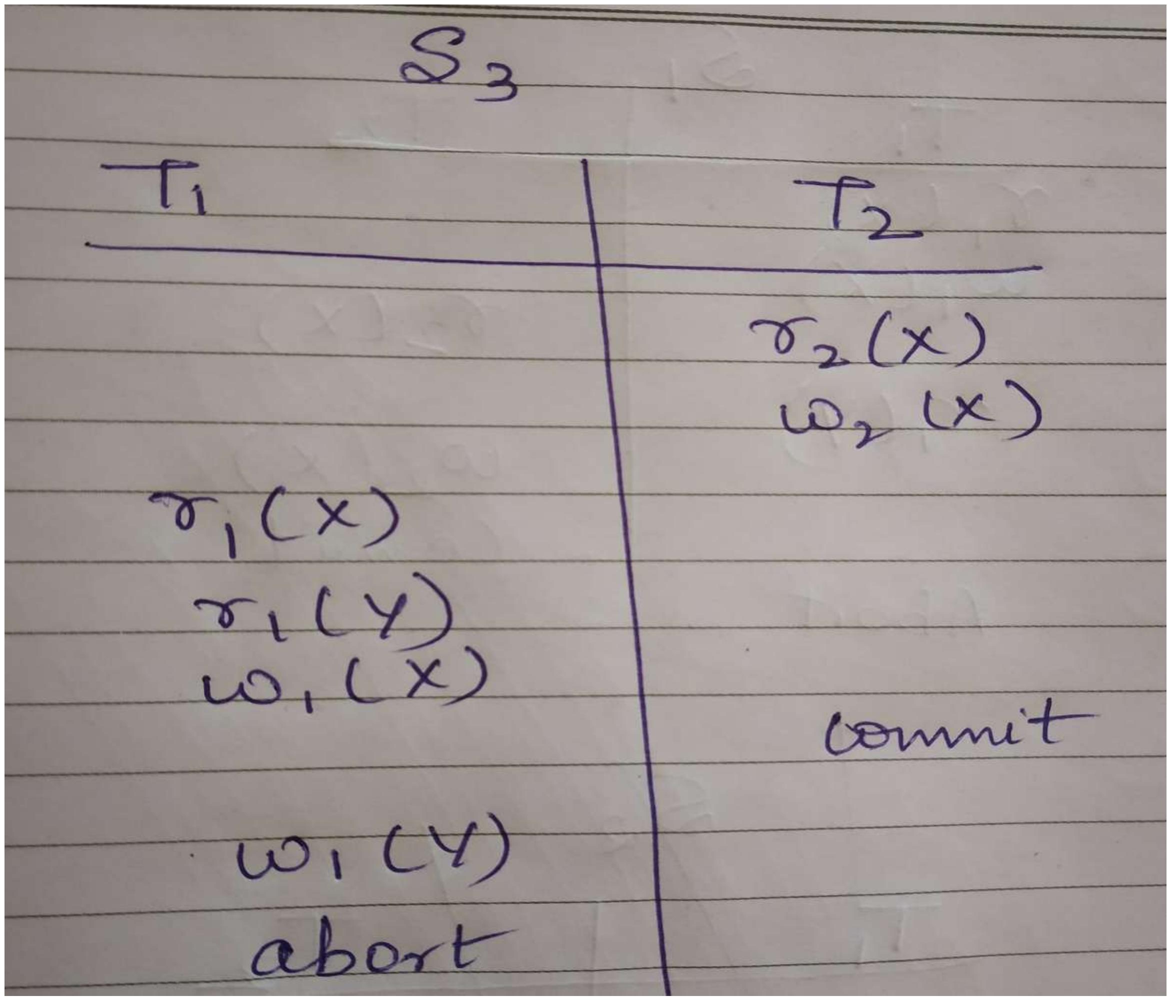


S2

T_1	T_2
$r_1(x)$	
	$r_2(x)$
$w_1(x)$	
$r_1(Y)$	
	$w_2(x)$
	commit
$w_1(Y)$	
abort	

Recoverable (T_2 does not have to be rolled back when T_1 aborts)

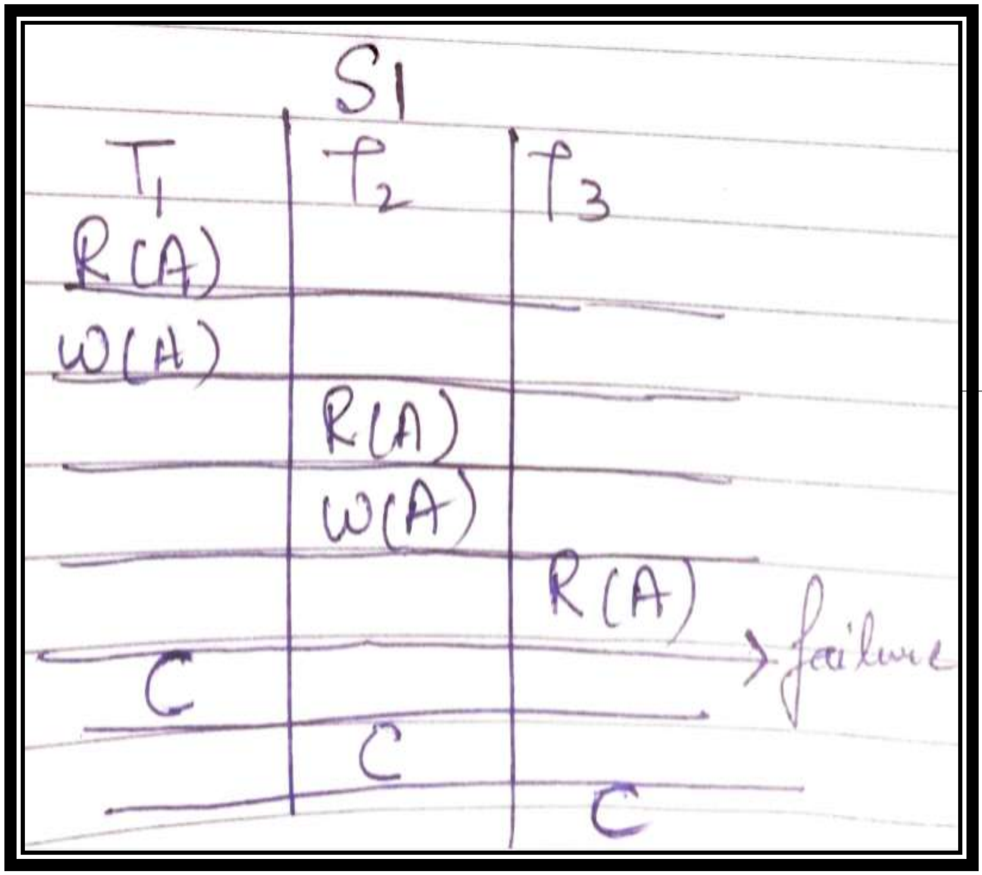
S3- $r_2(X)$ $w_2(X)$ $r_1(X)$ $r_1(Y)$ $w_1(X)$ c_2 $w_1(Y)$ a_1



S_3

T_1	T_2
	$r_2(X)$
	$w_2(X)$
$r_1(X)$	
$r_1(Y)$	
$w_1(X)$	
	commit
$w_1(Y)$	
abort	

Recoverable (T_2 does not have to be rolled back when T_1 aborts)

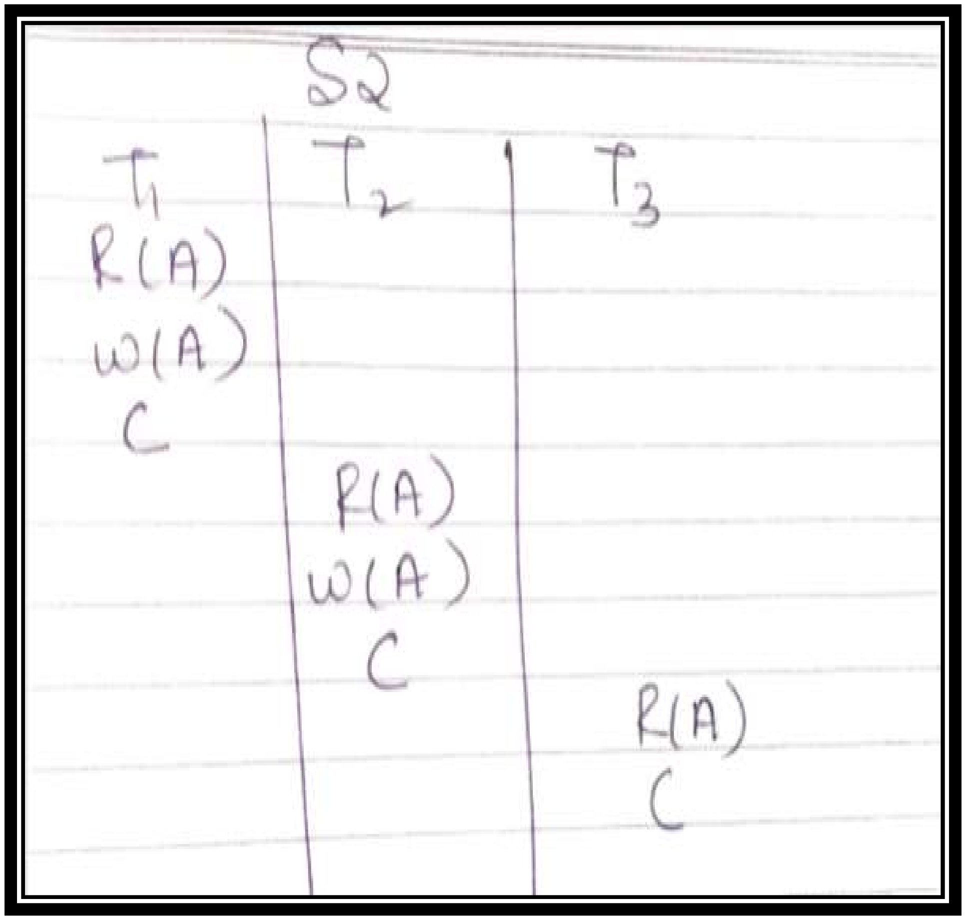


S1

T1	T2	T3	
R(A)			
W(A)			
	R(A)		
	W(A)		
		R(A)	
C			→ failure
	C		
		C	

Cascading Rollback

Cascadeless Schedule



S2

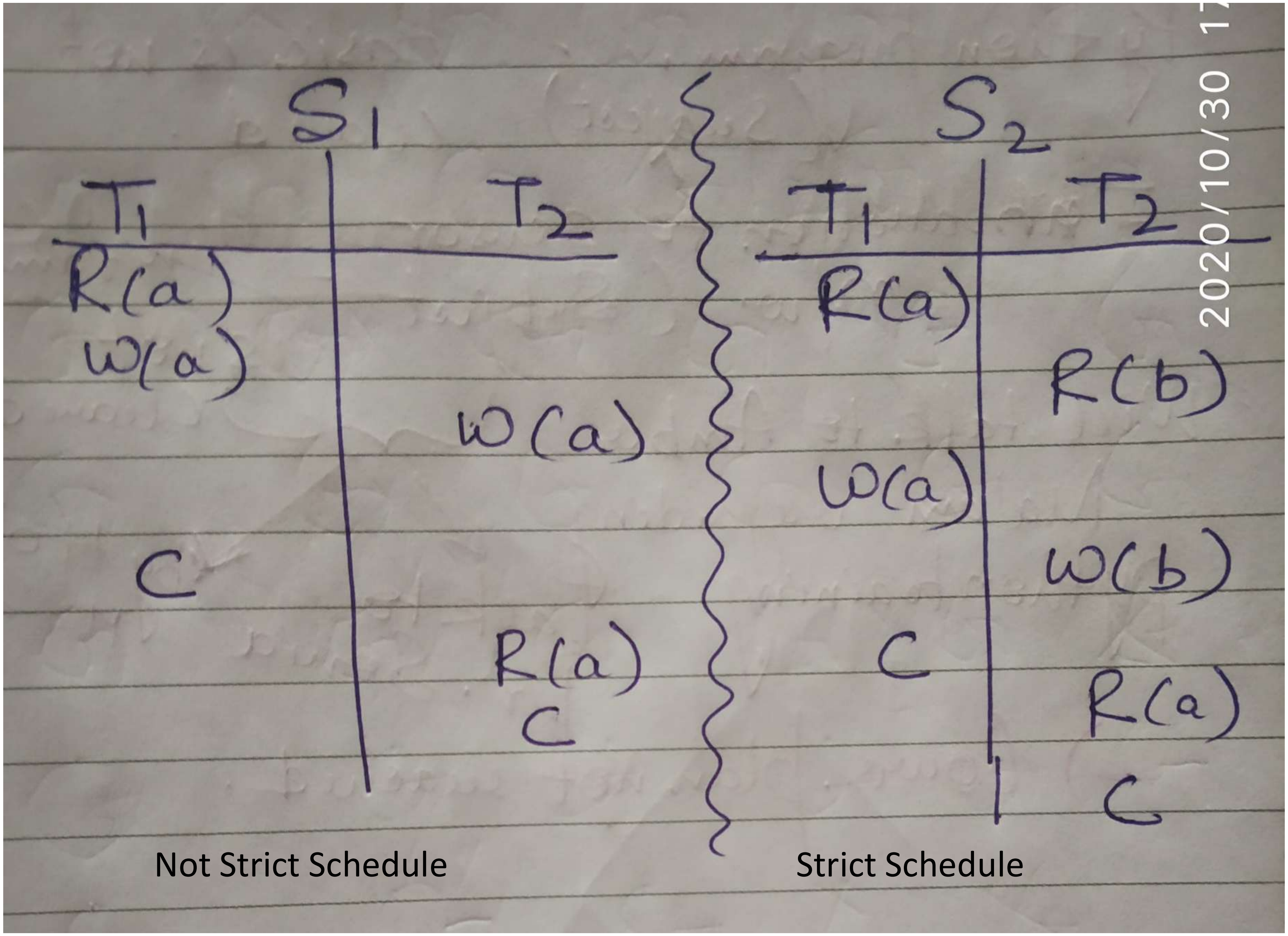
T1	T2	T3
R(A)		
W(A)		
C		
	R(A)	
	W(A)	
	C	
		R(A)
		C

Schedules requiring cascaded rollback:

- A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.

Strict Schedules:

- A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.



S_1

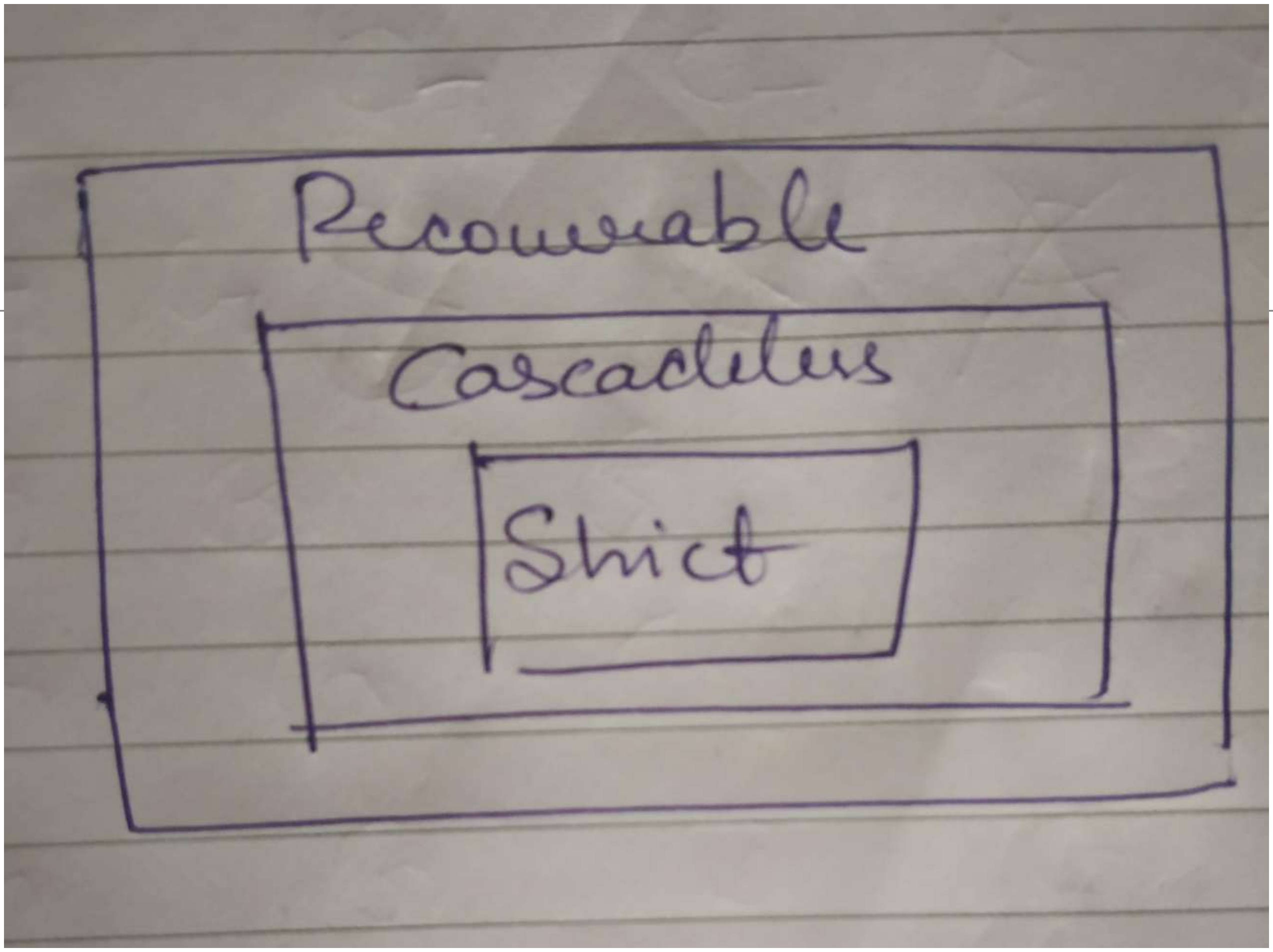
T_1	T_2
R(a)	
W(a)	
	W(a)
C	
	R(a)
	C

Not Strict Schedule

S_2

T_1	T_2
R(a)	
	R(b)
W(a)	
	W(b)
C	
	R(a)
	C

Strict Schedule



1. Cascadeless schedules are recoverable
2. Strict schedules are cascadeless and recoverable
3. More stringent condition means easier to do recovery from failure but less concurrency

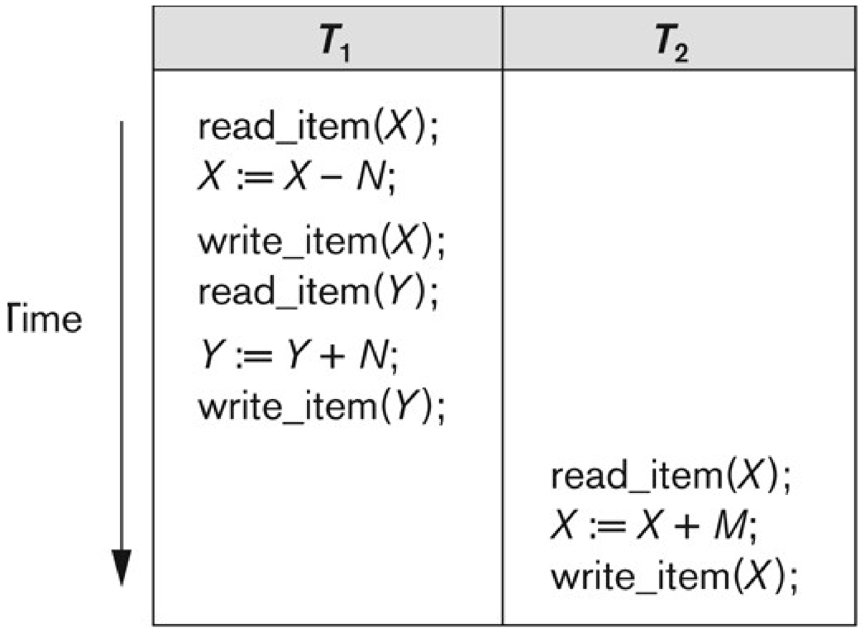
Characterizing Schedules based on Serializability

Serial schedule:

- A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
  - Otherwise, the schedule is called non-serial schedule.

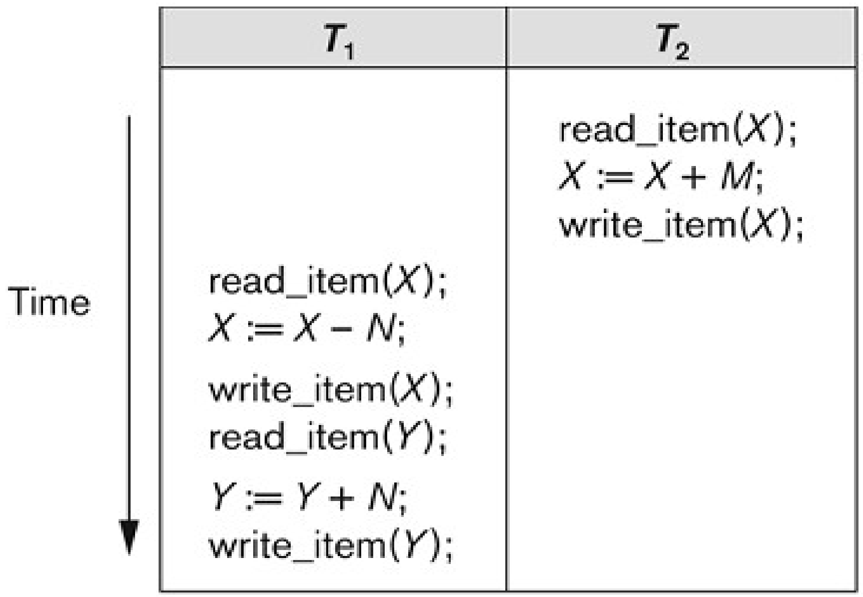
Serializable schedule:

- A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions



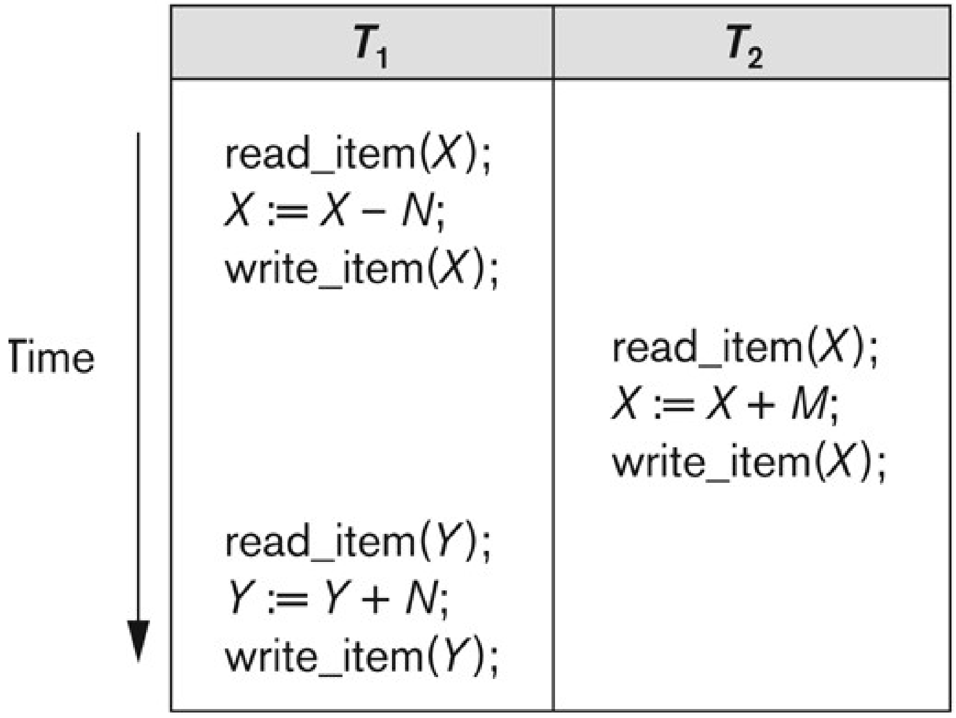
T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A



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

Schedule B



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

Schedule D

Schedules based on Serializabilty

1. **Conflict Serializable**
2. **View Serializable**

Conflict Serializable

A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule of the same transactions

Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules

Two operations in a transaction schedule are in conflict if

- They belong to different transactions
- They access the same data item
- At least one of them is a write operation

Conflict

We say that *Ii* and *Ij* conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

Conflict equivalent

If a schedule S1 can be transformed into a schedule S2 by a series of swaps of non-conflicting instructions, we say that S1 and S2 are conflict equivalent.

If we are able to swap the instructions in schedule S1 to form another schedule S2, and S1 and S2 produces same consistent database then we say that the schedules are conflict equivalent.

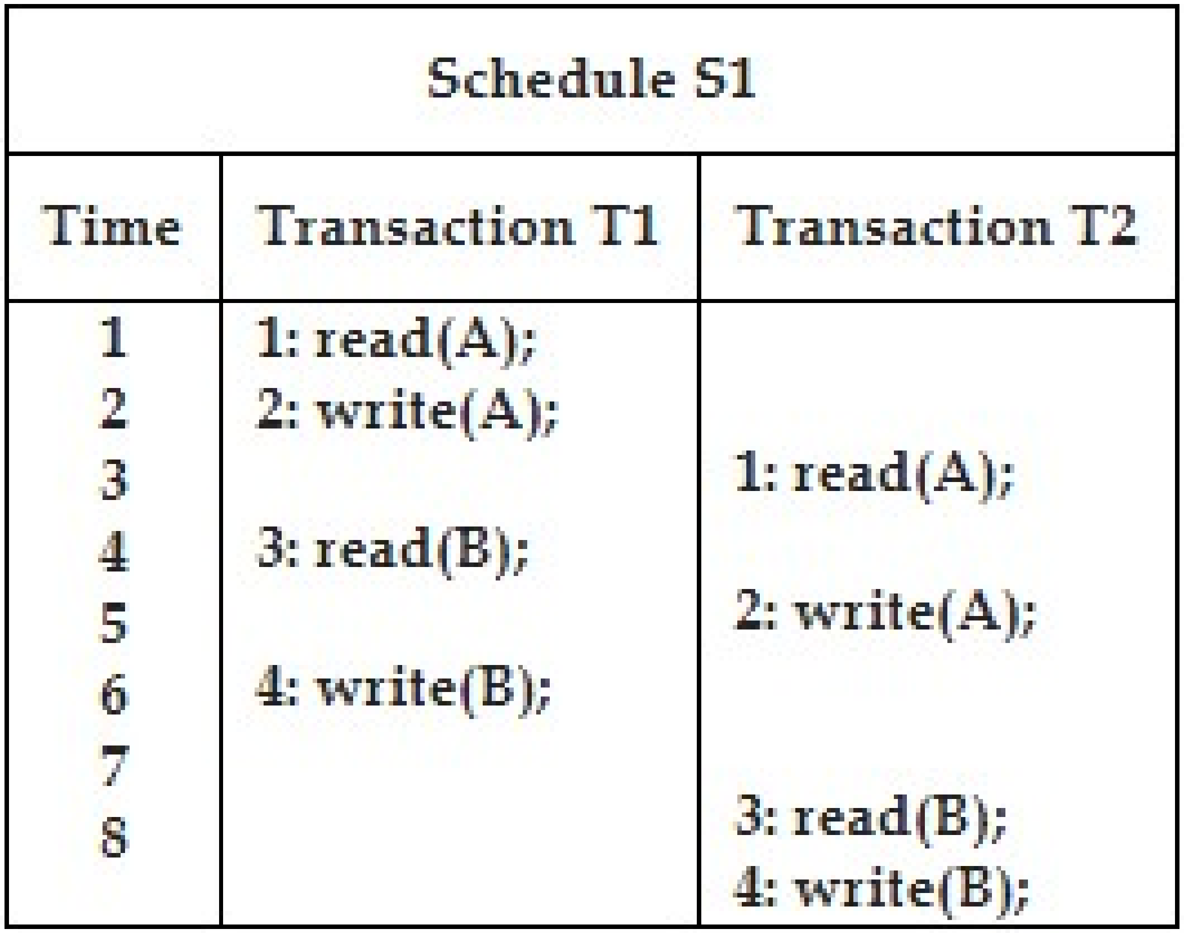
Conflict serializable

A schedule S1 is said to be conflict serializable if it is conflict equivalent to a serial schedule.

Schedule S		***T1***	
		R(X)	W(X)
T2	R(X)	**OK**	**Not OK**
	W(X)	Not OK	**Not OK**

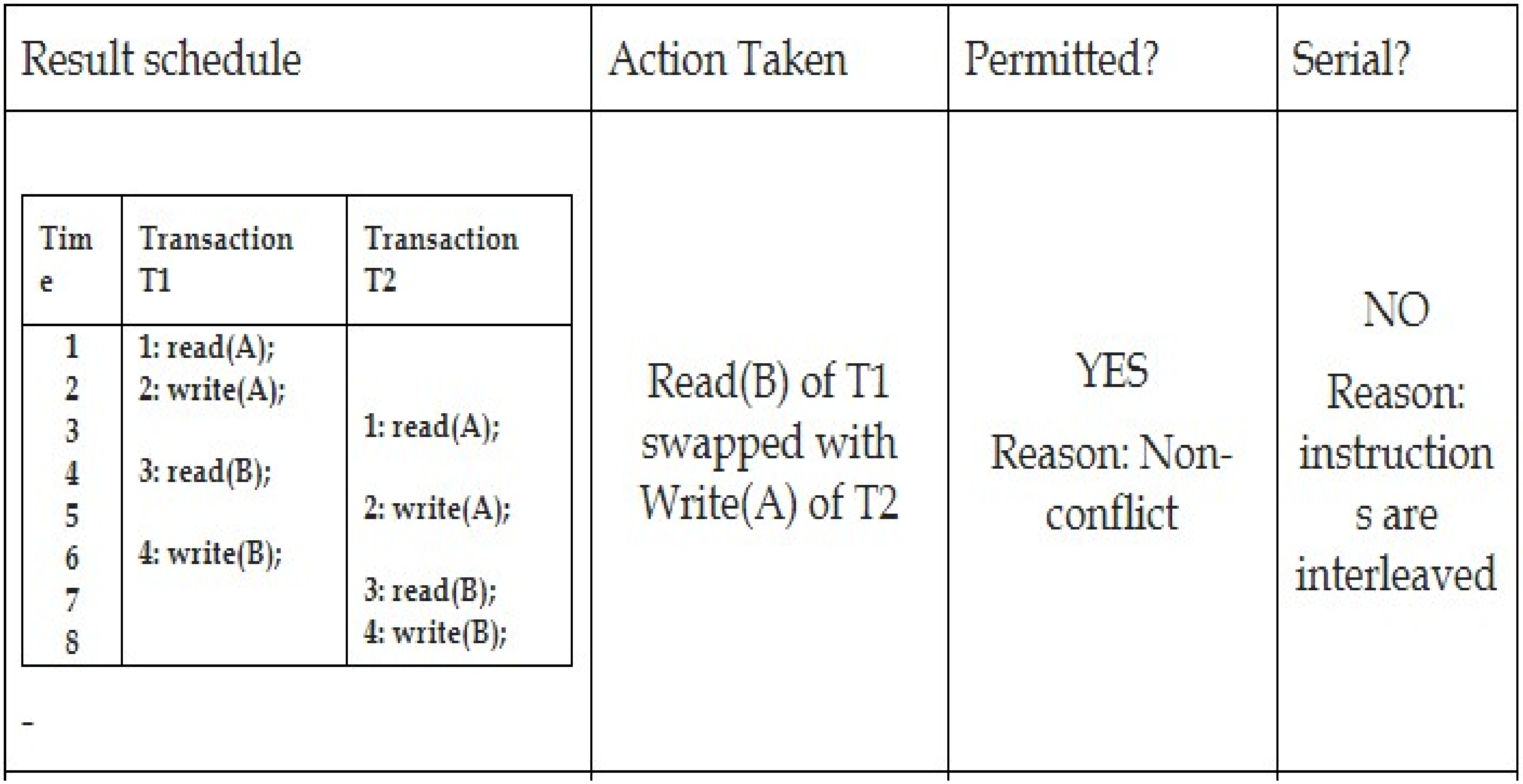
If a schedule is conflict serializable, we can reorder the nonconflicting operations until we form an equivalent serial schedule

Is S1 Conflict Serializable Schedule?



Schedule S1		
Time	Transaction T1	Transaction T2
1	1: read(A);	
2	2: write(A);	
3		1: read(A);
4	3: read(B);	
5		2: write(A);
6	4: write(B);	
7		3: read(B);
8		4: write(B);

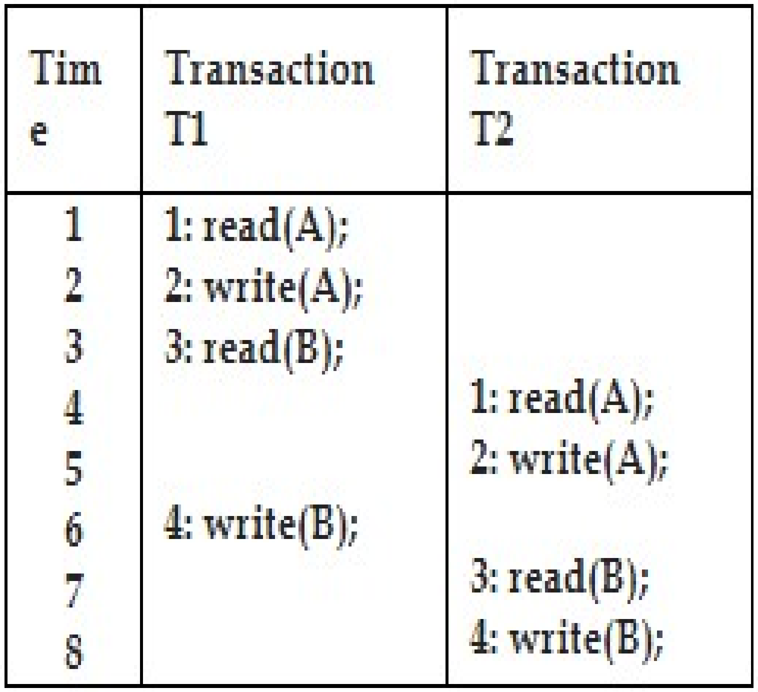
Step 1



Result schedule	Action Taken	Permitted?	Serial?
(schedule below)	Read(B) of T1 swapped with Write(A) of T2	YES Reason: Non-conflict	NO Reason: instructions are interleaved

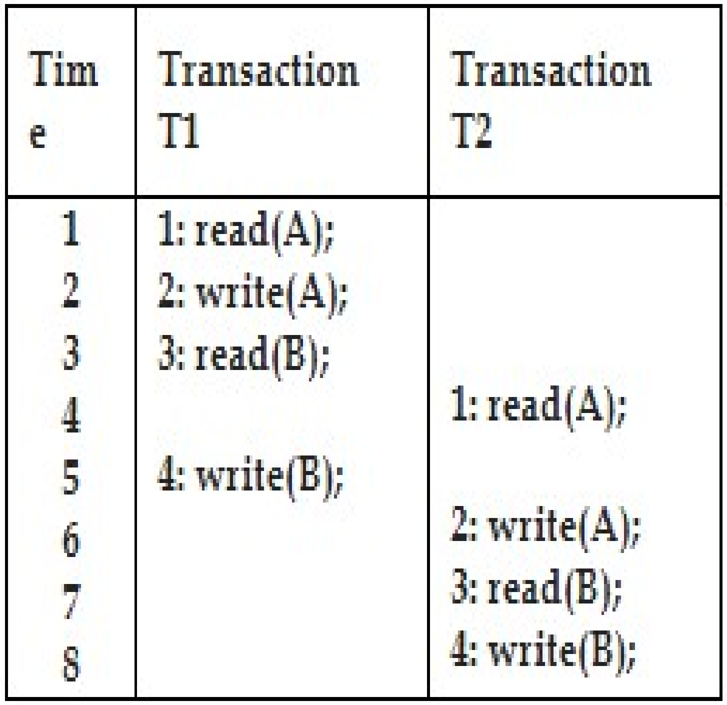
Time	Transaction T1	Transaction T2
1	1: read(A);	
2	2: write(A);	
3		1: read(A);
4	3: read(B);	
5		2: write(A);
6	4: write(B);	
7		3: read(B);
8		4: write(B);

Step 2

 -	Read(B) of T1 swapped with Read(A) of T2	YES	NO

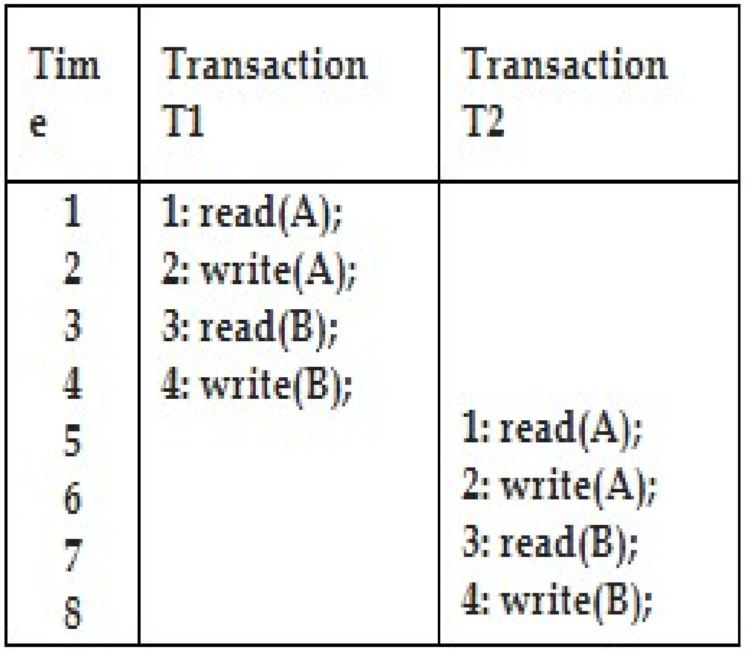
Time	Transaction T1	Transaction T2
1	1: read(A);	
2	2: write(A);	
3	3: read(B);	
4		1: read(A);
5		2: write(A);
6	4: write(B);	
7		3: read(B);
8		4: write(B);

Step 3

 -	Write(B) of T1 swapped with Write(A) of T2	YES	NO

Time	Transaction T1	Transaction T2
1	1: read(A);	
2	2: write(A);	
3	3: read(B);	
4		1: read(A);
5	4: write(B);	
6		2: write(A);
7		3: read(B);
8		4: write(B);

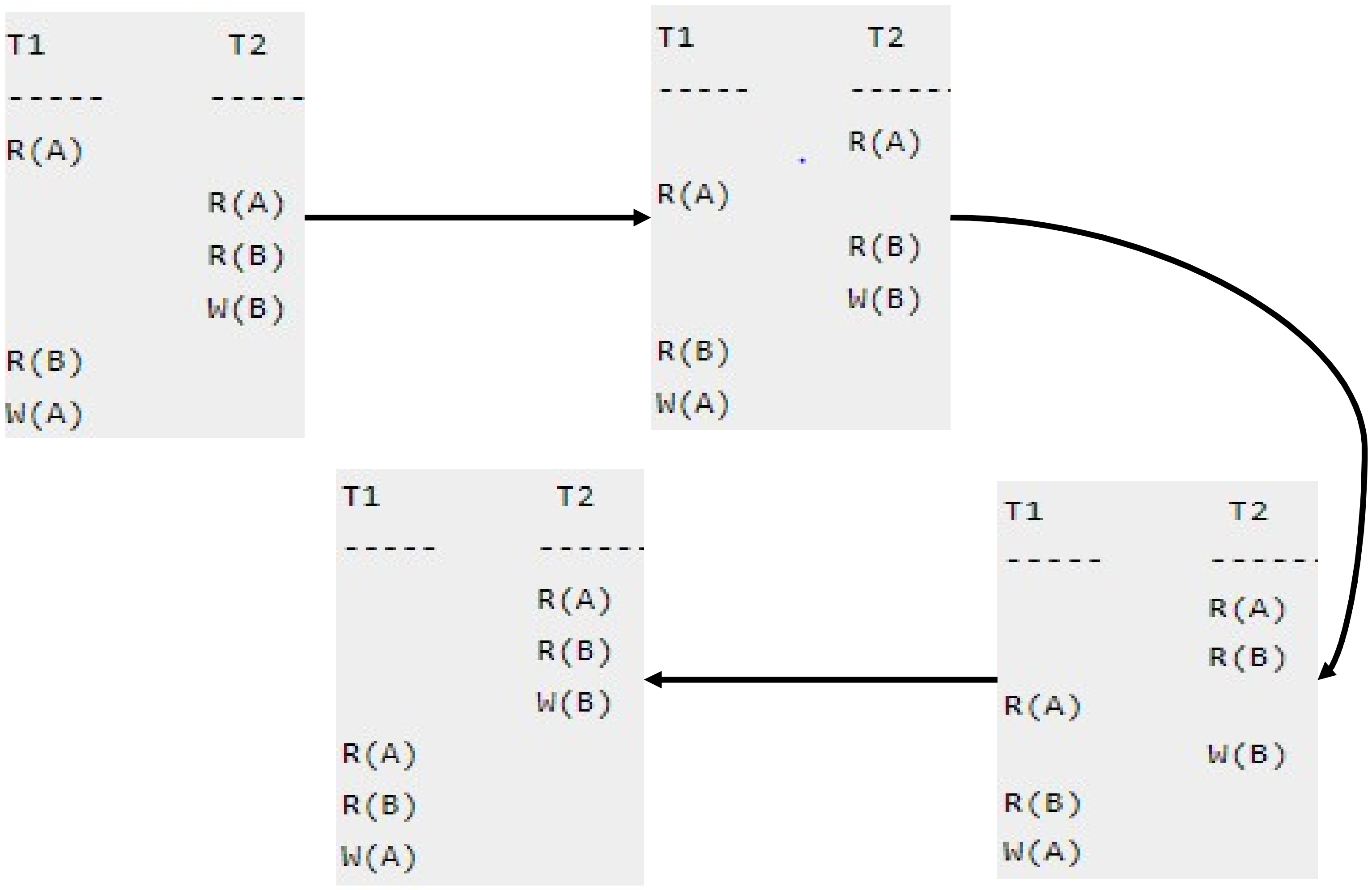
Step 4

	Write(B) of T1 swapped with Read(A) of T2	YES	YES T1 then T2 is the serial order

Tim e	Transaction T1	Transaction T2
1	1: read(A);	
2	2: write(A);	
3	3: read(B);	
4	4: write(B);	
5		1: read(A);
6		2: write(A);
7		3: read(B);
8		4: write(B);

Final answer: schedule S1 is conflict serializable to the serial schedule T1-T2. that is, the concurrent schedule S1 is equivalent to executing T1 first then T2.

Another Example



```
T1          T2
-----       -----
R(A)
            R(A)
            R(B)
            W(B)
R(B)
W(A)
```

```
T1          T2
-----       -----
            R(A)
R(A)
            R(B)
            W(B)
R(B)
W(A)
```

```
T1          T2
-----       -----
            R(A)
            R(B)
R(A)
            W(B)
R(B)
W(A)
```

```
T1          T2
-----       -----
            R(A)
            R(B)
            W(B)
R(A)
R(B)
W(A)
```

Testing for Conflict Serializability

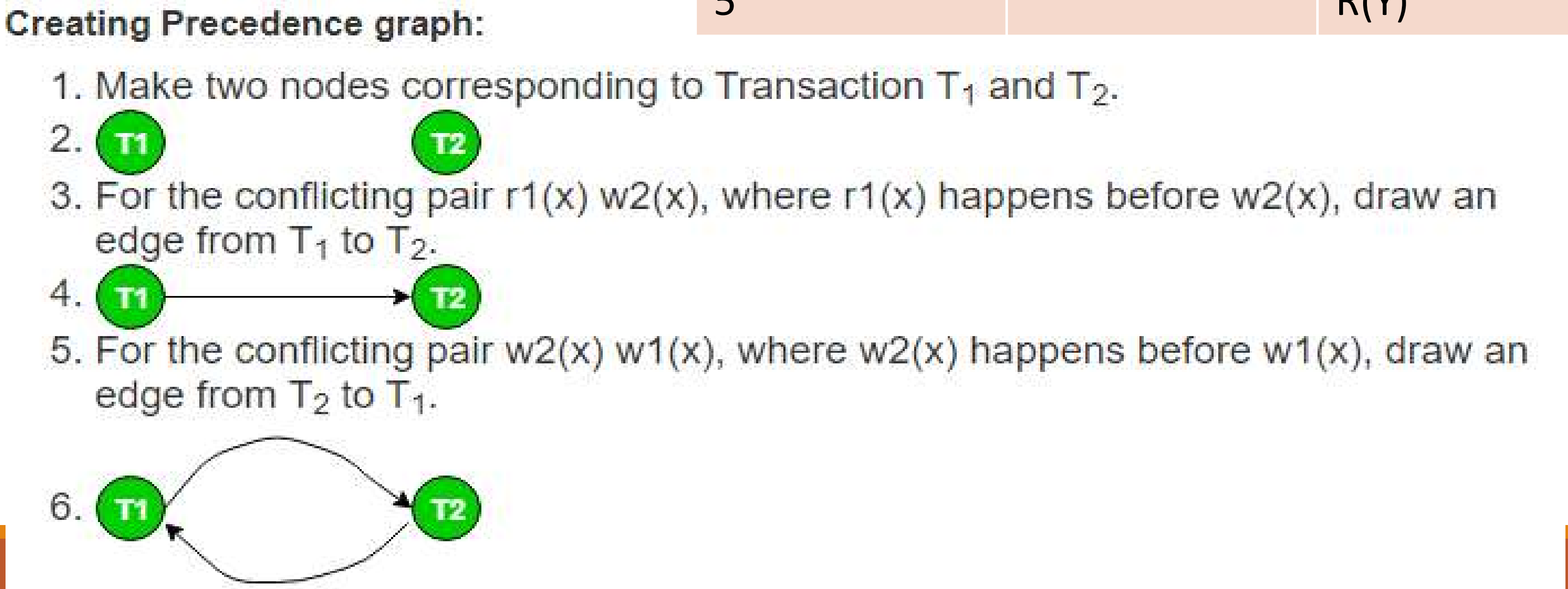
Construct a precedence graph (serialization graph) where

- Nodes are the transactions
- A directed edge is created from T_i to T_j if one of the operations in T_i appears before a conflicting operation in T_j
  - Create edge $T_i \rightarrow T_j$ if schedule contains $w_i(X)$ $r_j(X)$
  - Create edge $T_i \rightarrow T_j$ if schedule contains $r_i(X)$ $w_j(X)$
  - Create edge $T_i \rightarrow T_j$ if schedule contains $w_i(X)$ $w_j(X)$

A schedule is conflict serializable if and only if the precedence graph has no cycles.

S : r1(x) r1(y) w2(x) w1(x) r2(y)

S. No.	T1	T2
1	R(X)	
2	R(Y)	
3		W(X)
4	W(X)	
5		R(Y)

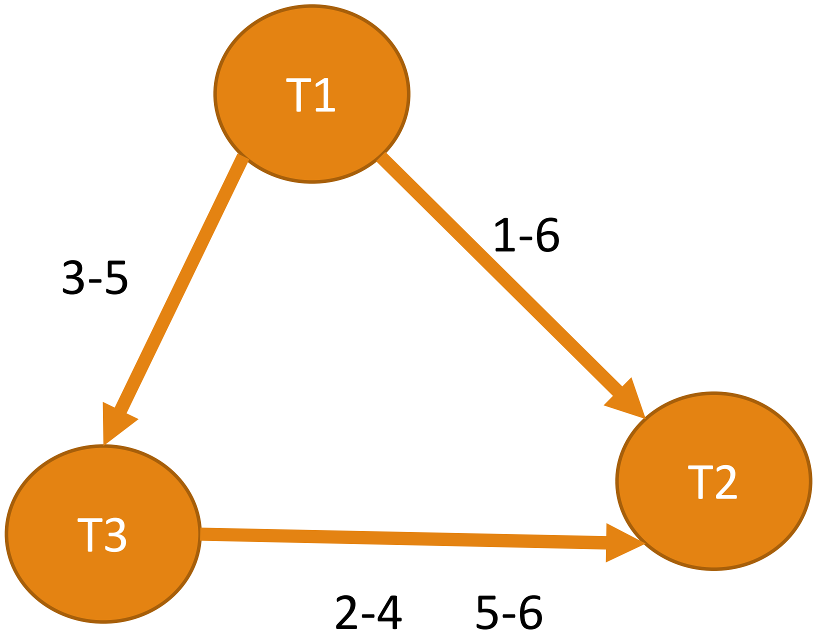


Creating Precedence graph:

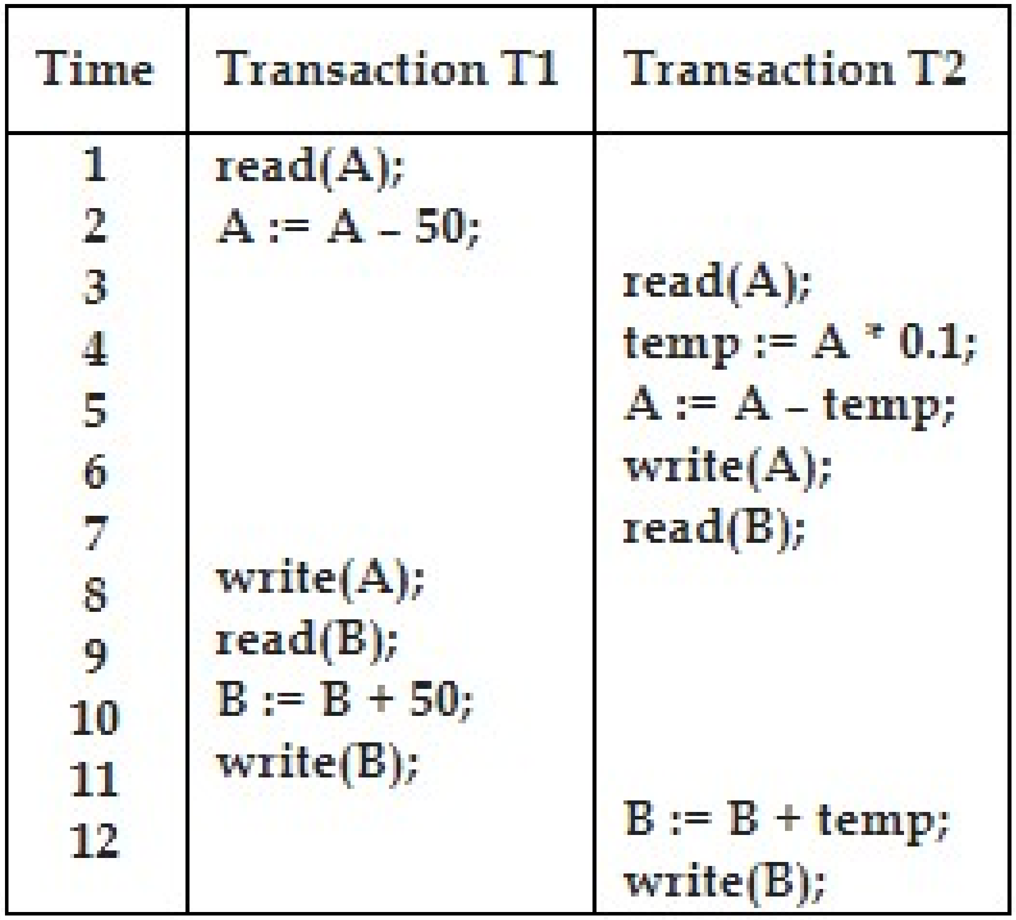
1. Make two nodes corresponding to Transaction T_1 and T_2.
2. $$\begin{tikzcd} T1 & & T2 \end{tikzcd}$$
3. For the conflicting pair r1(x) w2(x), where r1(x) happens before w2(x), draw an edge from T_1 to T_2.
4. $$\begin{tikzcd} T1 \arrow[rr] & & T2 \end{tikzcd}$$
5. For the conflicting pair w2(x) w1(x), where w2(x) happens before w1(x), draw an edge from T_2 to T_1.
6. $$\begin{tikzcd} T1 \arrow[rr, bend left] & & T2 \arrow[ll, bend left] \end{tikzcd}$$

S1: r1(x) r3(y) w1(x) w2(y) r3(x) w2(x)

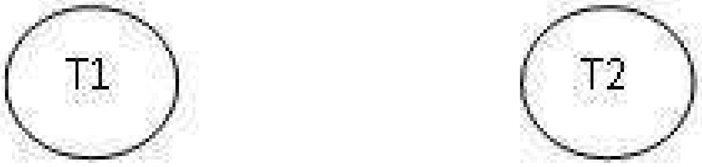
S.no.	T1	T2	T3
1	R(X)		
2			R(Y)
3	W(X)		
4		W(Y)	
5			R(X)
6		W(X)	



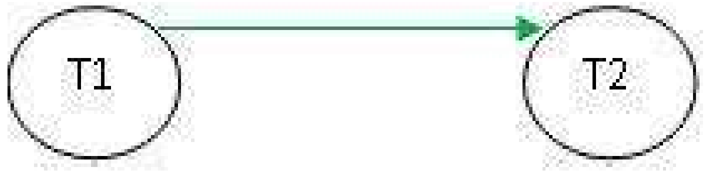
Precedence Graph does not contain cycle so it is conflict serializable



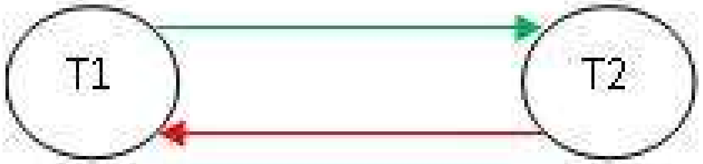
Time	Transaction T1	Transaction T2
1	read(A);	
2	A := A – 50;	
3		read(A);
4		temp := A * 0.1;
5		A := A – temp;
6		write(A);
7		read(B);
8	write(A);	
9	read(B);	
10	B := B + 50;	
11	write(B);	
12		B := B + temp;
		write(B);



(a)



(b)



(c)

View Serializabilty

View serializability is a concept that is used to compute whether schedules are View-Serializable or not. A schedule is said to be View-Serializable if it is view equivalent to a Serial Schedule (where no interleaving of transactions is possible).

Condition of schedules to View-equivalent

1) Initial Read
If a transaction T1 reading data item A from initial database in S1 then in S2 also T1 should read A from initial database.

2)Updated Read /Intermediate Read
If Ti is reading A which is updated by Tj in S1 then in S2 also Ti should read A which is updated by Tj.

3)Final Write operation
If a transaction T1 updated A at last in S1, then in S2 also T1 should perform final write operations.

T1	T2
Read(A)	
	Write(A)

Schedule S1

T1	T2
	Write(A)
Read(A)	

Schedule S2

T1	T2	T3
Write(A)		
	Write(A)	
		Read(A)

Schedule S1

T1	T2	T3
	Write(A)	
Write(A)		
		Read(A)

Schedule S2

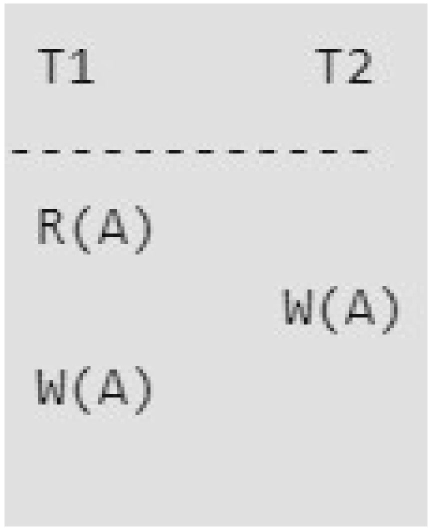
T1	T2	T3
Write(A)		
	Read(A)	
		Write(A)

Schedule S1

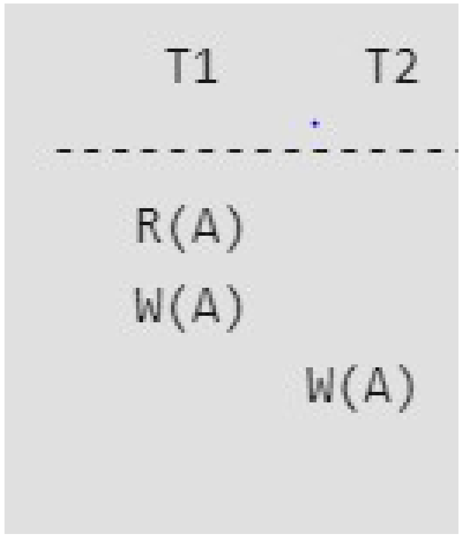
T1	T2	T3
	Read(A)	
Write(A)		
		Write(A)

Schedule S2

These two schedules are view equivalent or not?



T1	T2
R(A)	
	W(A)
W(A)	

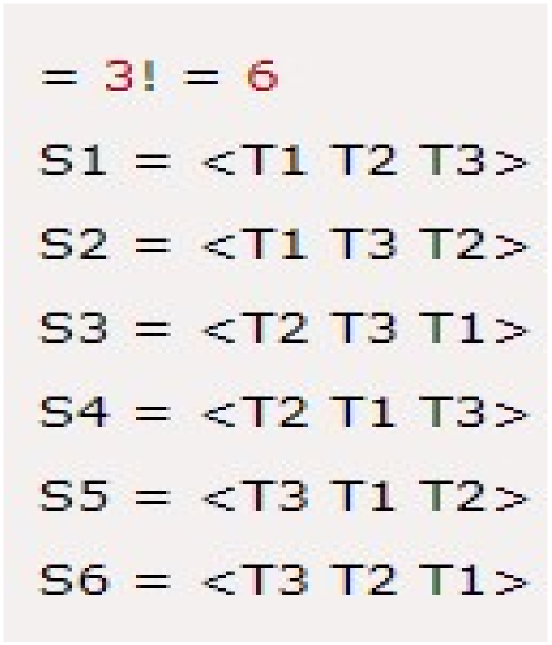


T1	T2
R(A)	
W(A)	
	W(A)

Above two schedules are not view-equivalent as Final write operation in S1 is done by T1 while in S2 done by T2.

Schedule S

With 3 transactions, the total number of possible serial schedule



= 3! = 6

S1 = <T1 T2 T3>

S2 = <T1 T3 T2>

S3 = <T2 T3 T1>

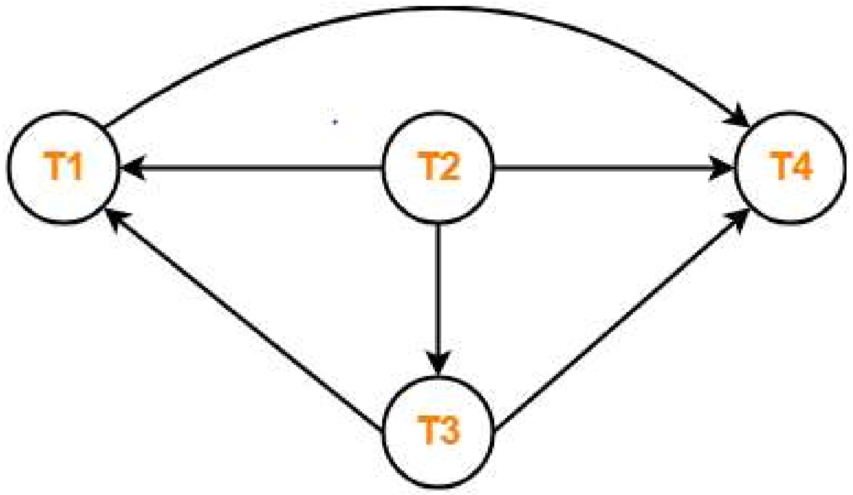
S4 = <T2 T1 T3>

S5 = <T3 T1 T2>

S6 = <T3 T2 T1>

Check whether the given schedule S is conflict serializable and recoverable or not-

T1	T2	T3	T4
	R(X)		
		W(X)	
		Commit	
W(X)			
Commit			
	W(Y)		
	R(Z)		
	Commit		
			R(X)
			R(Y)
			Commit


T1
T2
T4
T3

Practice Questions

S1 : $R_1(A)$, $R_2(A)$, $R_1(B)$, $R_2(B)$, $R_3(B)$, $W_1(A)$, $W_2(B)$