BITS, PILANI – K. K. BIRLA GOA CAMPUS

Database Systems (CS F212)

by

Dr. Mrs. Shubhangi Gawali

Dept. of CS and IS

TRANSACTION MANAGEMENT

Concepts

- Database system Examples
- Definition
- Properties of a Transaction

Examples

- Banking system
- Reservation system
- Library management system

Definition

- Transaction is a unit of program that access and updates various data items in data base.

ATM System

Transaction: Withdraw some amount

Steps:

START the transaction

1. Prompt: Enter the amount to be withdrawn

2.Checking enough balance

3. if yes then withdraw

(balance=balance – amount)and

Give the money

3. if no display the message

4. print the receipt

END the transaction

ATM System

Transaction: Withdraw some amount

Steps:

START the transaction

1. Prompt: Enter the amount to be withdrawn

2.Checking enough balance

3. if yes then withdraw

(balance=balance – amount)and

print the receipt

3. if no display the message

4. Give the money

END the transaction

ATM System

Transaction: Withdraw some amount

Steps:

START the transaction

1. Prompt: Enter the amount to be withdrawn

2.Checking enough balance

3. if yes then withdraw

(balance=balance – amount)and

print the receipt

3. if no display the message

TRANSACTION FAILS

4. Give the money

END the transaction

ATM System

Transaction: Withdraw some amount

Steps:

START the transaction

1. Prompt: Enter the amount to be withdrawn

2.Checking enough balance

3. if yes then withdraw

(balance=balance – amount)and

Give the money

3. if no display the message

TRANSACTION FAILS

4. print the receipt

END the transaction

Definition

- Unit of program that accesses and updates various data items
- Set of instructions
- Sequential order

Banking system

AccNo.	Customer Name	Balance amt
A1	Manish	1000/-
A2	Suman	2000/-

Transaction: Transfer Rs 500/- from account A1 to account A2

<u>Steps:</u>

```
BEGIN
1. Read (A1, A1balance)
2  A1balance = A1balance - 500
3. Read (A2, A2balance)
4. A2balance = A2balance + 500
5. Write (A1, A1balance)
6. Write (A2, A2balance)
END
```

ATOMICITY

AccNo.	**Customer Name**	**Balance amt**
A1	Manish	1000/-
A2	Suman	2000/-

Transaction: Transfer Rs 500/- from account A1 to account A2

Steps:

```
START
1. Read (A1, A1balance)
2. A1balance = A1balance - 500
TRANSACTION FAILS
3. Read(A2, A2balance)
4. A2balance = A2balance + 500
5. Write (A1, A1balance)
6. Write(A2,A2balance)
END
```

SOLUTION:

(UNDO : A1balance=A1balance+500)

ALL OR NONE

CONSISTENCY

AccNo.	Customer Name	Balance amt
A1	Manish	1000/-
A2	Suman	2000/-

Transaction: Transfer Rs 500/- from account A1 to account A2

Steps:

```
BEGIN
1. Read (A1, A1balance)
2  A1balance=A1balance-500
3. Read(A2, A2balance)
4. A2balance=A2balance+500
5. Write (A1, A1balance)
6. Write(A2,A2balance)
END
```

Q. How do you know whether transaction failed/left incomplete?

SOLUTION: check the sum of A1balance and A2balance before starting the transaction. It is Rs. 3000.

check the sum of A1balance and A2balance after completing the transaction It is also Rs. 3000 .(same as before means data is consistent)

ISOLATION

Transaction: both C1 and C2 are accessing joint account

Transaction1: C1 is withdrawing money.

Transaction 2: C2is checking balance and withdrawing money

T1	T2
	balance = 500
Withdraw 500	
	trying to withdraw but cannot

He/she is the only one doing the transaction.

DURABILITY

- What happens if the database server crashes before the changed data is written onto a stable storage?

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

Solution: client to server, transaction logs, backup etc.

ACID Properties of a transaction

- Atomicity
- Consistency
- Isolation
- Durability

Definition of Transcation

- Unit of program that accesses and updates various data items
- Set of instructions
- Sequential order
- ACID properties

Responsible Components

Property of transaction	**Component**
Atomicity	Transaction Management Component
Consistency	Application Programmer
Isolation	Concurrency Control Component
Durability	Recovery Management Component

Types of failures

- **Computer Failure :** A H/W or S/W error in the computer during transaction execution.
- **Transaction Failure :** Failure caused by an operation in the transaction e.g. Division by zero.
- **Local errors :** Conditions that cause the cancellation of transaction e.g. data needed for transaction is not found.

Types of errors

- **Concurrency control enforcement** : Either of the transaction need to be aborted by CC methods.

  e.g withdrawal from ATM and Funds transfer.

- **Disk Failure** : Loss of data in disk blocks during a transaction due to say a disk read/write head.

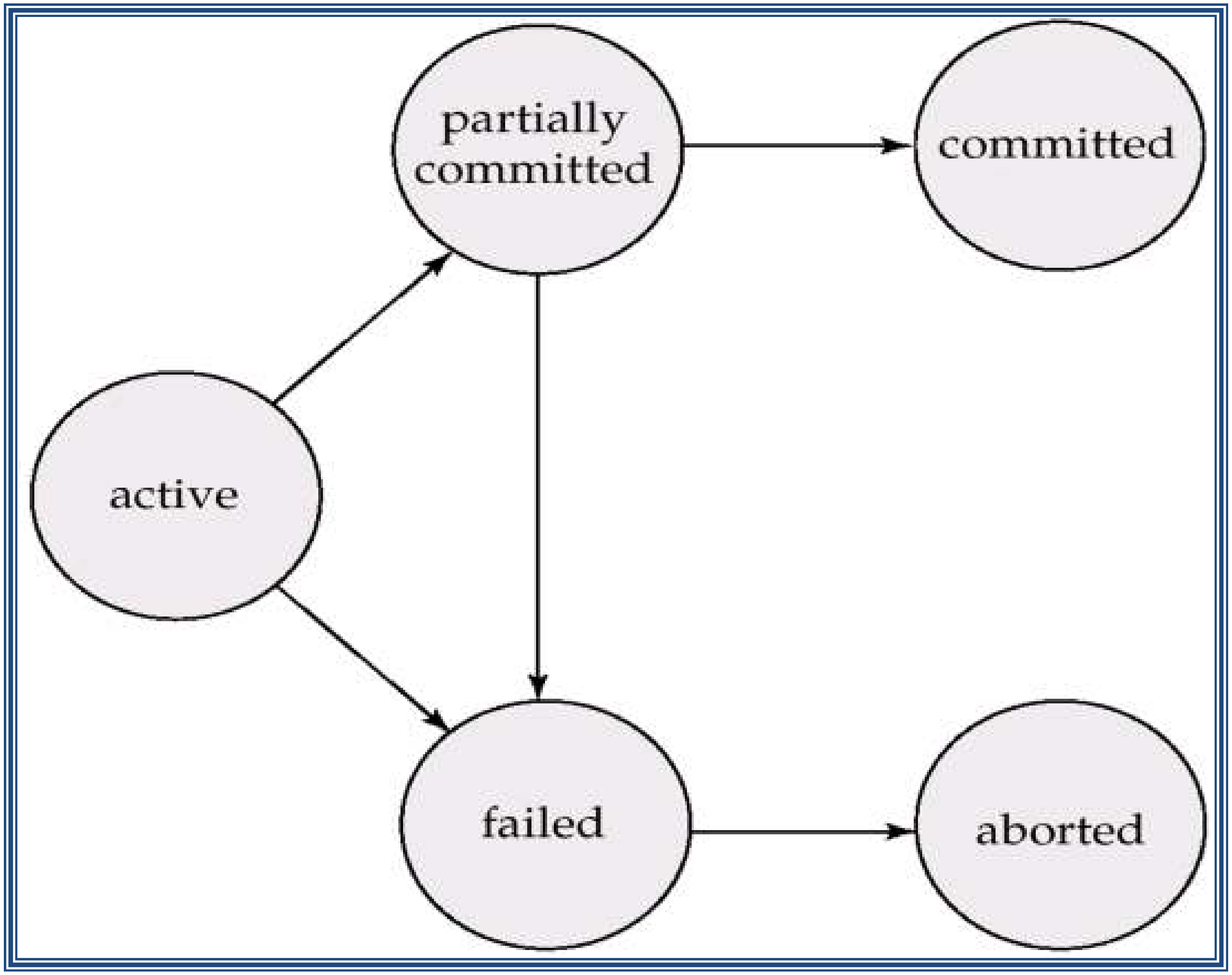
- **Physical problems & catastropes** : problems like power failure, earthquake i.e beyond human control.

TCL Commands

TCL : Transaction Control Language

- COMMIT : save work done to the server end
- SAVEPOINT : identify a point in a transaction to which we can later roll back
- ROLLBACK : Undo the transaction operations till last savepoint or till mentioned savepoint

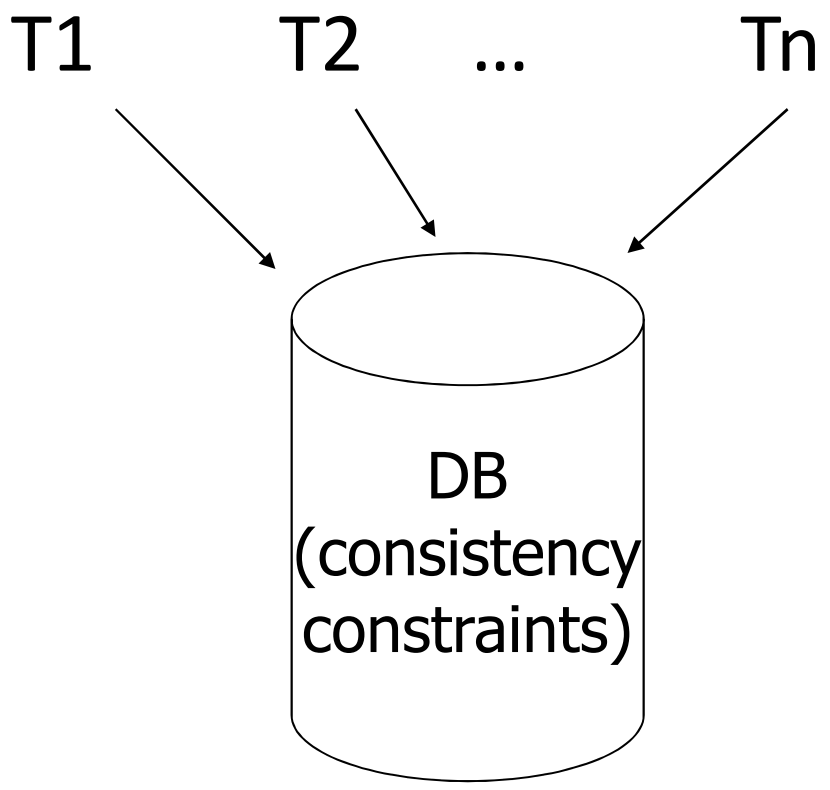
Transaction State Diagram



Transaction State Diagram

- **Active,** the initial state; the transaction stays in this state while it is executing
- **Partially committed,** before 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.

Concurrent Executions



Multiple transactions are allowed to run concurrently in the system.

Concurrent Executions

- Advantages :
  - **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.

Concurrent Executions

Concurrency control schemes – mechanisms to achieve isolation,

i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- Schedules – sequences that indicate the order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.

Example

- Consider two transactions (transactions) :

```
T1: BEGIN   A=A+1500,   B=B -1500   END
T2: BEGIN   A=1.06*A,    B=1.06*B   END
```

- Intuitively, the first transaction is transferring Rs1500 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.

- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.

Anomalies due to interleaved executions

- Reading Uncommitted data (WR Conflicts) (Dirty read problem) (due to temporary updates)
  - Eg: T2 reads the data updated by T1 and T1 fails before completing. Thus T2 reads dirty data.
- Unrepeatable Reads (RW Conflicts)(Incorrect summary problem)
  - Eg: giving bonus and transferring funds to/from same account
  - Eg: finding total seats available and cancelling ticket for same train.
- Overwriting Uncommitted Data (WW Conflicts)(The Lost Update Problem)
  - Eg: transferring the funds and debiting the amount from same account

Example (Contd.)

- Consider a possible interleaving (*schedule*):

```
T1:  A=A+1500,            B=B-1500
T2:             A=1.06*A,            B=1.06*B
```

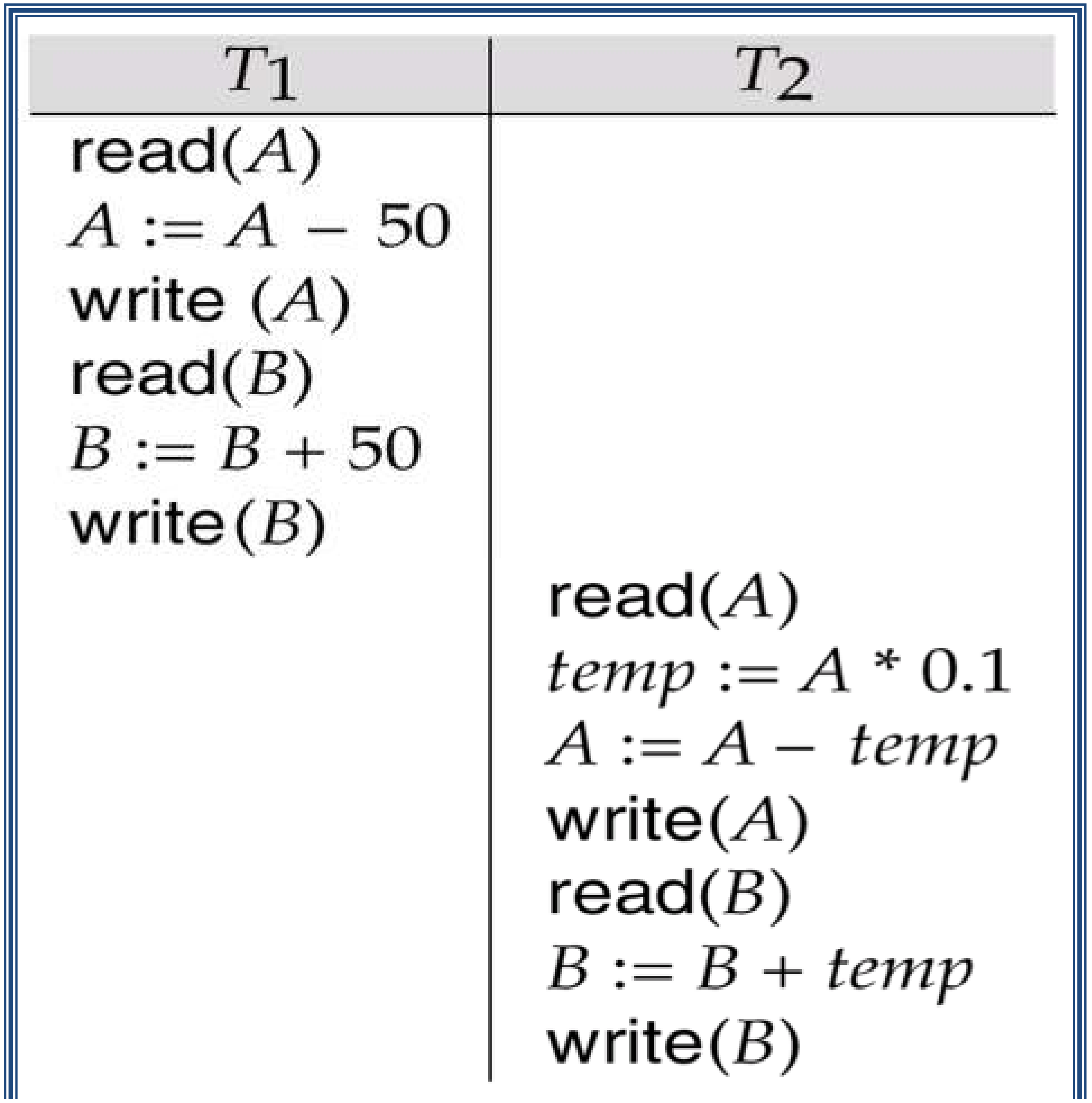
- The above one is OK. But what about:

```
T1: A=A+1500,                               B=B-1500
T2:             A=1.06*A, B=1.06*B
```

- The DBMS's view of the second schedule:

```
T1: R(A), W(A),                                  R(B), W(B)
T2:              R(A), W(A), R(B), W(B)
```

Example serial schedule 1: T1 is followed by T2



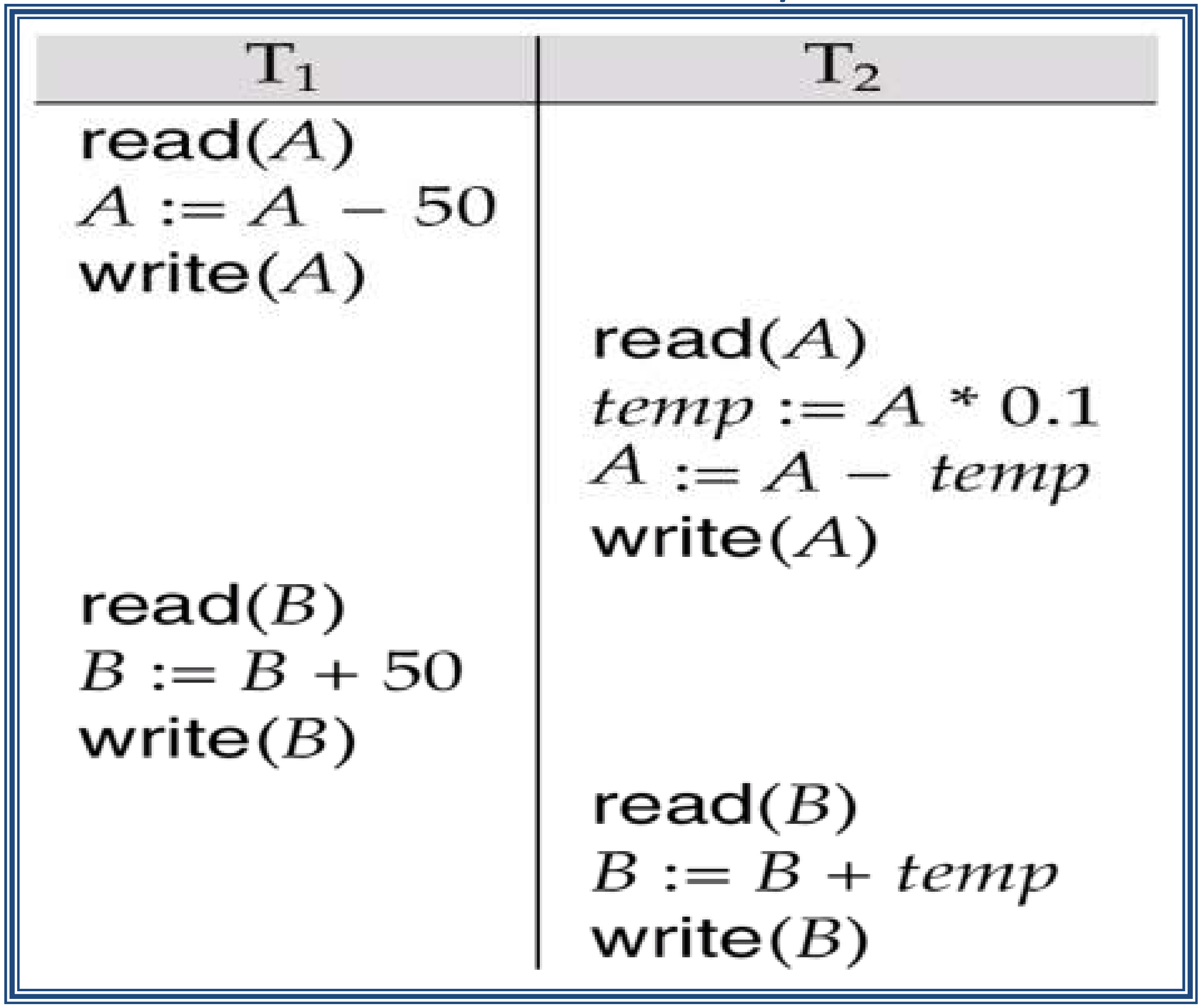
T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Example serial schedule 2: T2 is followed by T1

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

Example Schedule (Cont.)

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

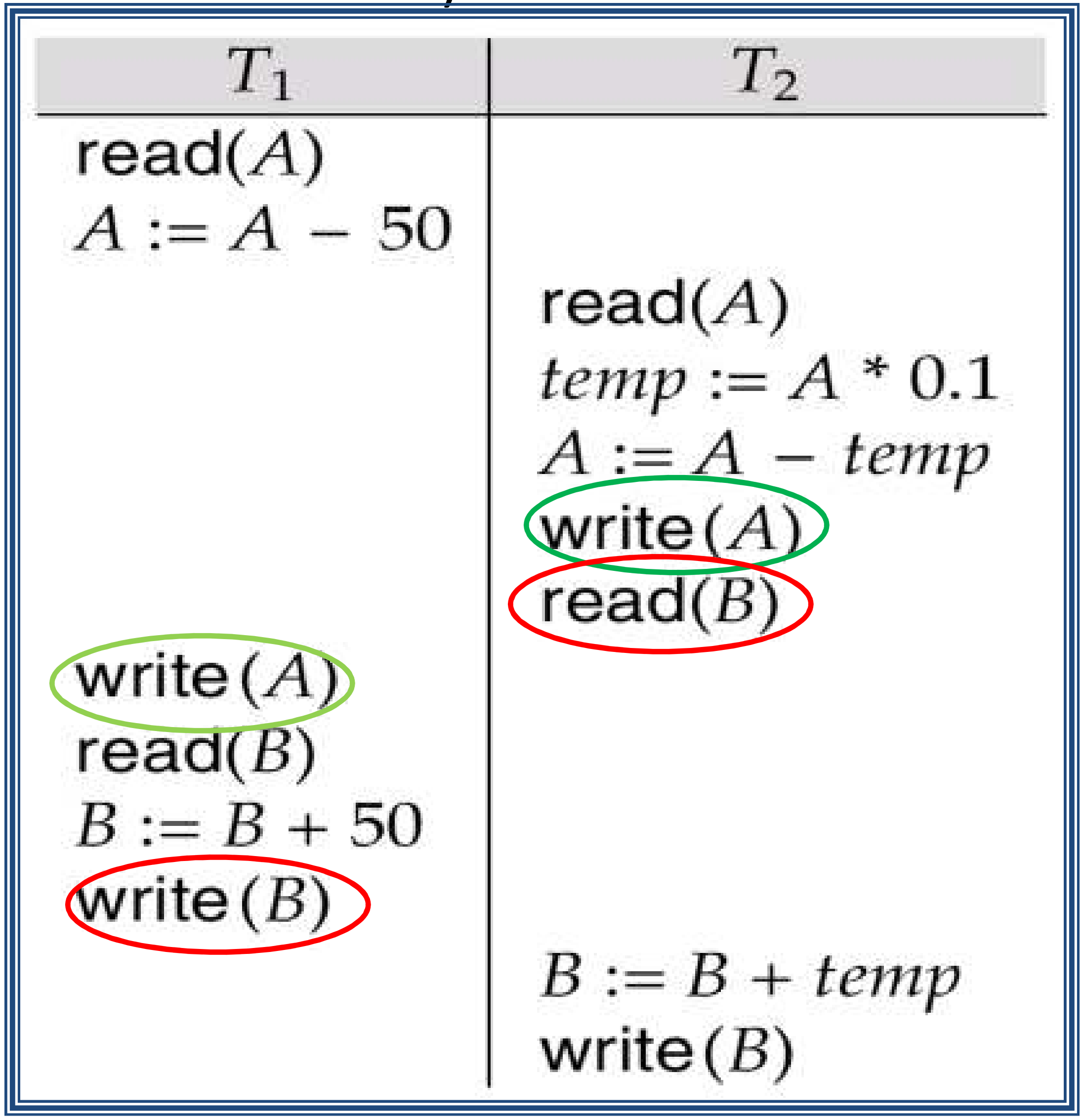


T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

In both Schedules , the sum A + B is preserved.

Example Schedules (cont...)

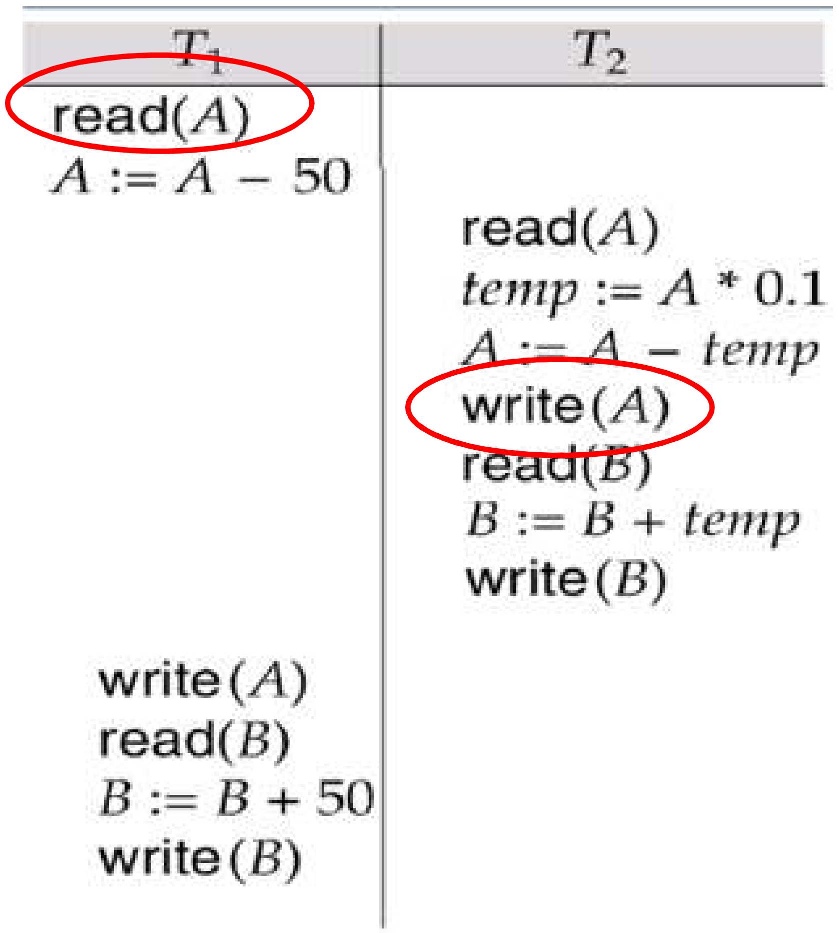
This concurrent schedule does not preserve the value of the the sum *A* + *B*. (WW and RW conflict)



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

Example Schedules (cont...)

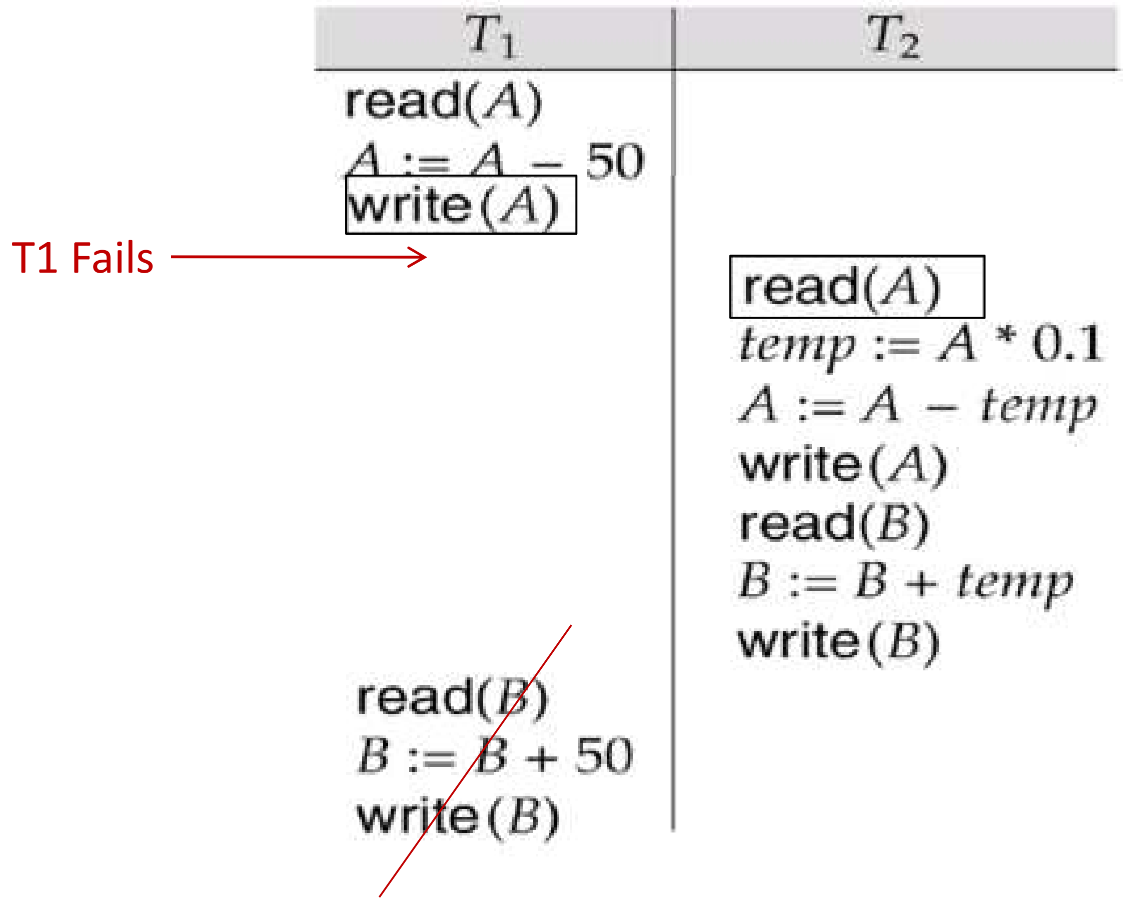
This concurrent schedule does not preserves the value of the the sum *A* + *B*. (RW Conflict)



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

Example Schedules (cont…)

This concurrent schedule does not preserve the value of the the sum *A* + *B*. (WR Conflict)



T1 Fails

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

Anomalies due to interleaved executions

- Reading Uncommitted data (WR Conflicts) (Dirty read problem) (due to temporary updates)
  - Eg: T2 reads the data updated by T1 and T1 fails before completing. Thus T2 reads dirty data.
- Unrepeatable Reads (RW Conflicts)(Incorrect summary problem)
  - Eg: giving bonus and transferring funds to/from same account
  - Eg: finding total seats available and cancelling ticket for same train.
- Overwriting Uncommitted Data (WW Conflicts)(The Lost Update Problem)
  - Eg: transferring then funds and debiting the amount from same account

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.

- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.

- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

Conflict Serializability

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.

  1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
  2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
  3. l_i = **write**(Q), l_j = **read**(Q). They conflict
  4. l_i = **write**(Q), l_j = **write**(Q). They conflict

Conflict Serializability

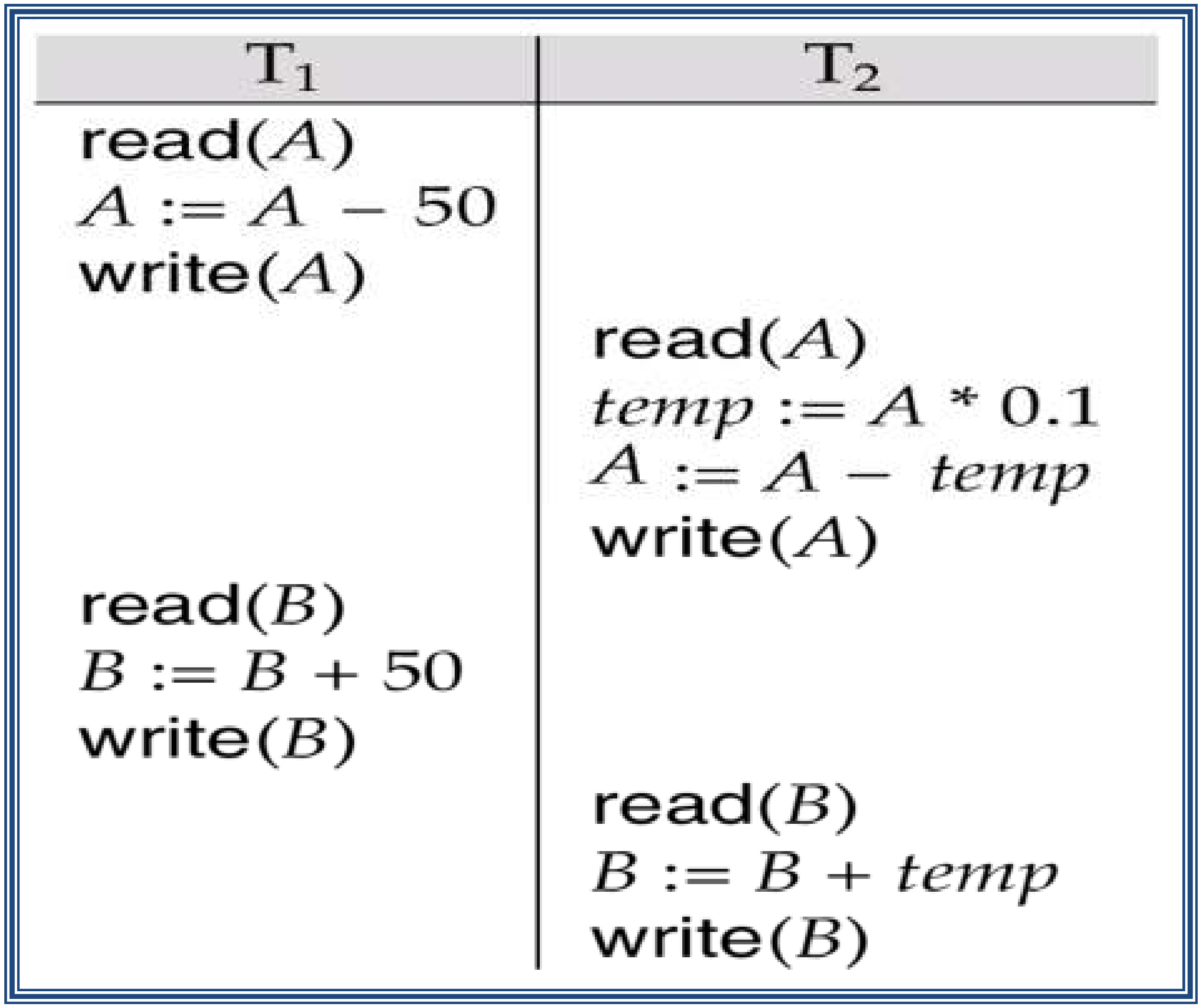
- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them. If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability (cont...)

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

- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Example schedule



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)	
	read(B)
	$B := B + temp$
	write(B)

Conflict Serializability (cont…)

This Schedule can be transformed into Schedule 1, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore this schedule is conflict serializable.

T1	T2
Read (A)	
Write (A)	
	Read (A)
	Write (A)
Read (B)	
Write (B)	
	Read (B)
	Write (B)

Conflict Serializability (cont…)

This Schedule can be transformed into Schedule 1, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore this schedule is conflict serializable.

T1	T2
Read (A)	
Write (A)	
	Read (A)
Read (B)	
	Write (A)
Write (B)	
	Read (B)
	Write (B)

Conflict Serializability (cont…)

This Schedule can be transformed into Schedule 1, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore this schedule is conflict serializable.

T1	T2
Read (A)	
Write (A)	
	Read (A)
Read (B)	
Write (B)	
	Write (A)
	Read (B)
	Write (B)

Conflict Serializability (cont…)

This Schedule can be transformed into Schedule 1, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore this schedule is conflict serializable.

T1	T2
Read (A)	
Write (A)	
Read (B)	
	Read (A)
Write (B)	
	Write (A)
	Read (B)
	Write (B)

Conflict Serializability (cont…)

This Schedule can be transformed into Schedule 1, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore this schedule is conflict serializable.

T1	T2
Read (A)	
Write (A)	
Read (B)	
Write (B)	
	Read (A)
	Write (A)
	Read (B)
	Write (B)

Example : non conflict serializable schedule

- Example of a schedule that is not conflict serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Exercise : check whether the schedule is conflict serializable or not

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Testing for Serializability

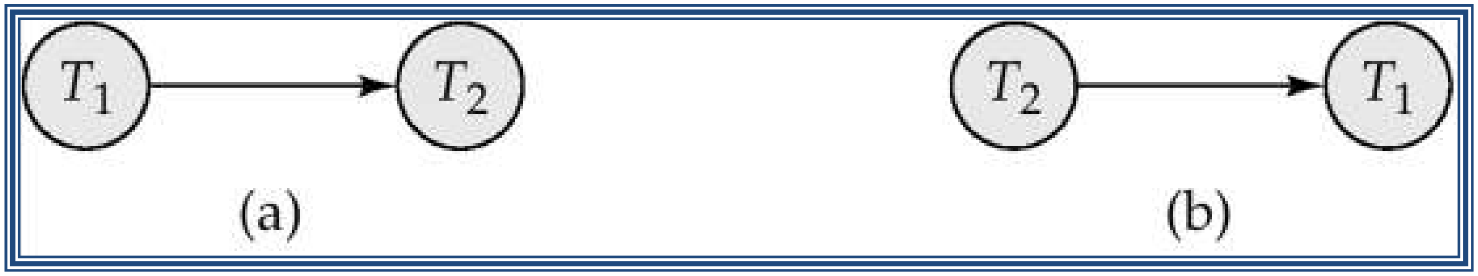
- Consider some schedule of a set of transactions T_1, T_2, …, T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- Draw an arc from T_i to $T_k$if the two transaction conflict (i.e one of the transaction writes to same data item), and T_i accessed the data item on which the conflict arose earlier.
- Label the arc by the item that was accessed.

Precedence Graph

- Given a schedule S, involving transactions T1 and T2, perhaps among other transactions, we say that T1 takes precedence over T2, written as T1 <T2, if there are actions A1 of T1 and A2 of T2, such that:

1. A1 is ahead of A2 in S,
2. Both A1 and A2 involve the same database element, and
3. At least one of A1 and A2 is a write action

Precedence Graph for (a) Schedule 1 and (b) Schedule 2



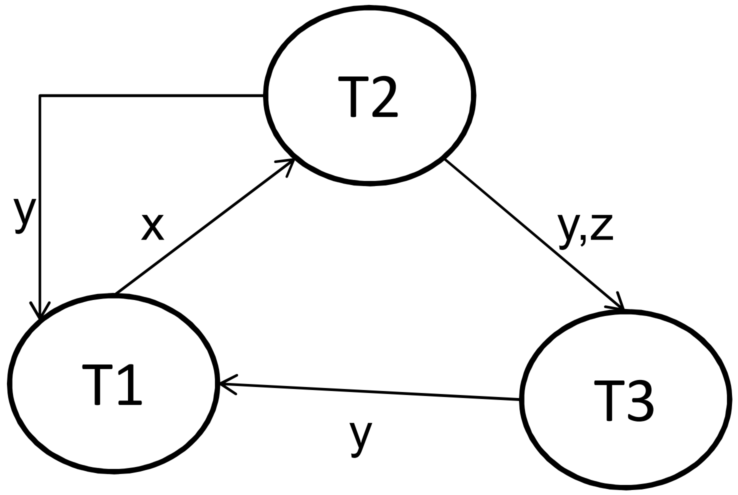
Example

	T1	T2	T3
t1		R(Z)	
t2		R(Y)	
t3		W(Y)	
t4			R(Y)
t5			R(Z)
t6	R(X)		
t7	W(X)		
t8			W(Y)
t9			W(Z)
t10		R(X)	
t11	R(Y)		
t12	W(Y)		
t13		W(X)	

Example

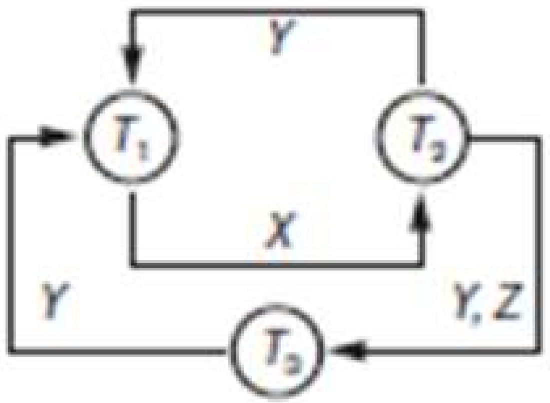
	T1	T2	T3
t1		R(Z)	
t2		R(Y)	
t3		W(Y)	
t4			R(Y)
t5			R(Z)
t6	R(X)		
t7	W(X)		
t8			W(Y)
t9			W(Z)
t10		R(X)	
t11	R(Y)		
t12	W(Y)		
t13		W(X)	

Precedence graph



Thus the given schedule is not serializable since the precedence graph is cyclic.

(d)



Equivalent serial schedules

None

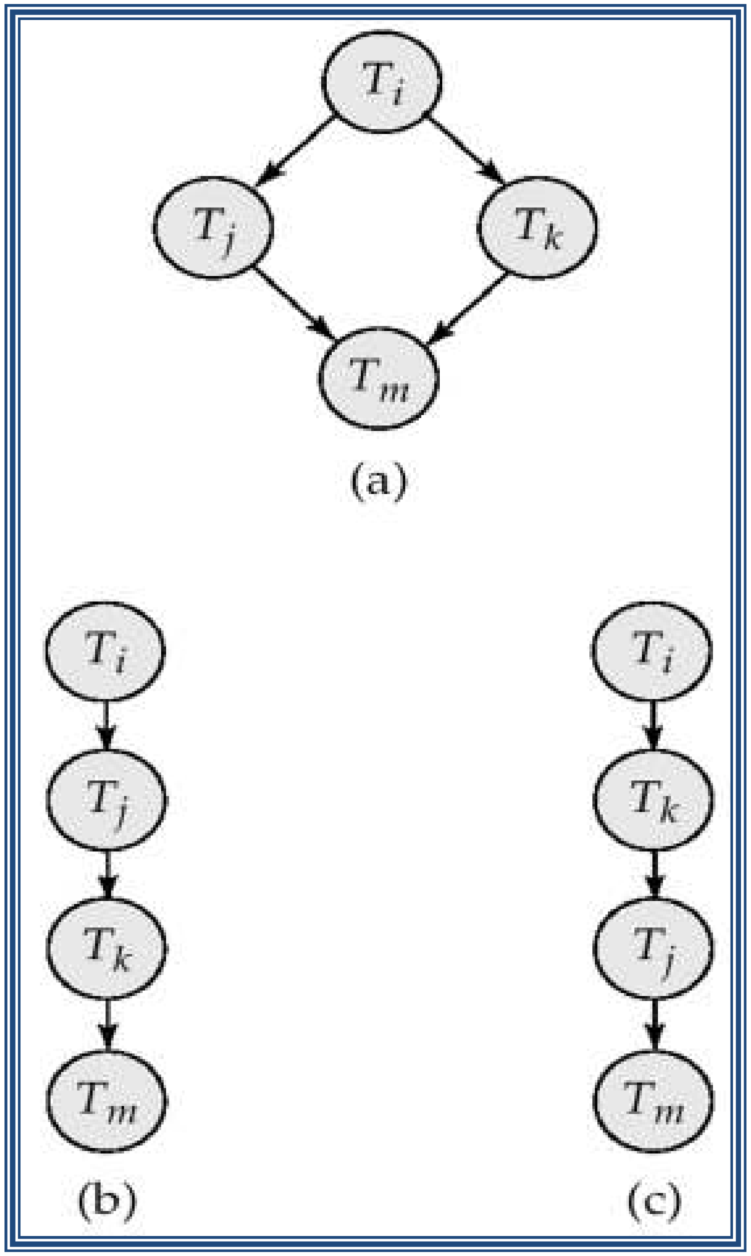
Reason

Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

Test for Conflict Serializability

- **A schedule is conflict serializable if and only if its precedence graph is acyclic.**
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph. (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.

Illustration of Topological Sorting

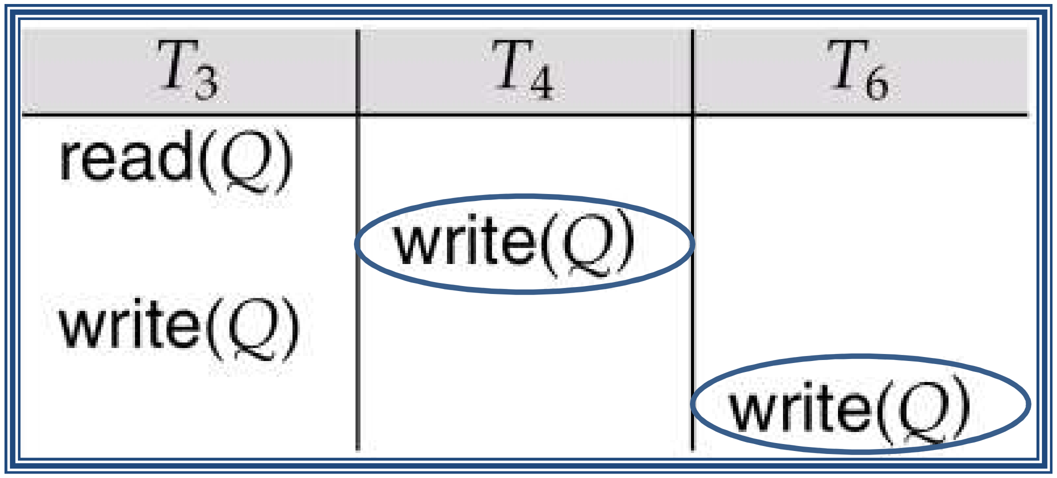


View Serializability

- Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are **view equivalent** if the following three conditions are met:
  1. For each data item *Q*, if transaction T_i reads the initial value of *Q* in schedule *S*, then transaction T_i must, in schedule *S´*, also read the initial value of *Q*.
  2. For each data item *Q* if transaction T_i executes **read**(*Q*) in schedule *S*, and that value was produced by transaction T_j (if any), then transaction T_i must in schedule *S´* also read the value of *Q* that was produced by transaction T_j .
  3. For each data item *Q*, the transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must perform the final **write**(*Q*) operation in schedule *S´*.

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

Blind writes

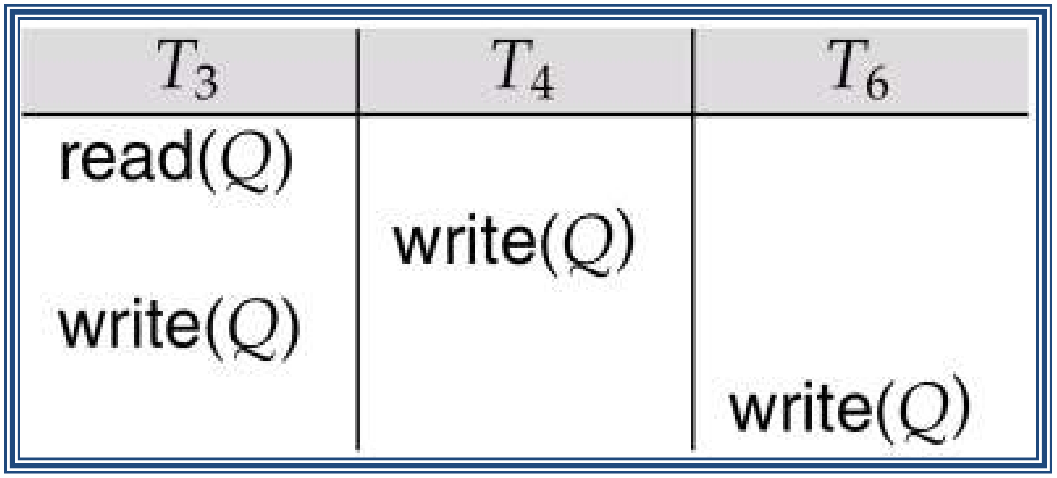


T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

A **blind write** is when a transaction writes to an object without ever reading the object.

View Serializability (Cont.)

- A schedule *S* is **view serializable** it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Schedule below - a schedule which is view-serializable but *not* conflict serializable.



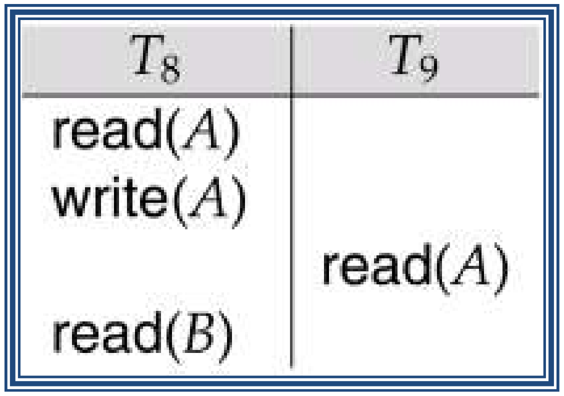
T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- Every view serializable schedule that is not conflict serializable has **blind writes.**

Recoverability

- Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction T_j reads a data items previously written by a transaction T_i , the commit operation of T_i appears before the commit operation of T_j.
- The following schedule is not recoverable if *T8* commits immediately after write(A)

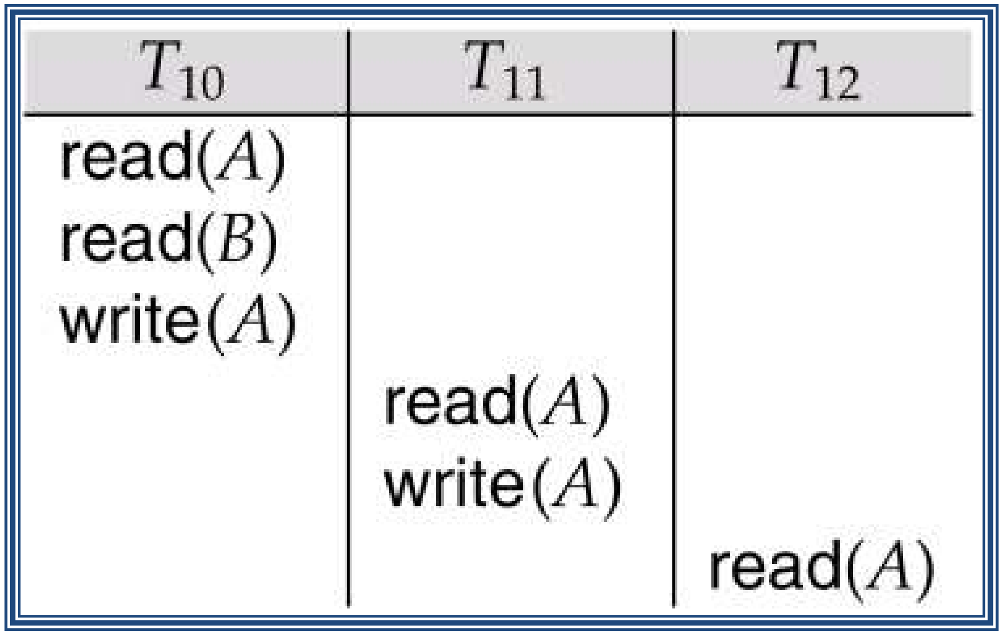


T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- If T_8 needs to abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence database must ensure that schedules are recoverable.

Recoverability (Cont.)

Cascading rollback – a single transaction failure leads to a series of transaction rollbacks.
Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)



T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

- If T_{10} fails, T_{11} and T_{12} must also be rolled back.
- Can lead to the undoing of a significant amount of work

Recoverability (Cont.)

- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.

- Every cascadeless schedule is also recoverable.

- It is desirable to restrict the schedules to those that are cascadeless.