Database Design & Applications (SS ZG 518)

BITS Pilani
Hyderabad Campus

Prof.R.Gururaj
CS&IS Dept.

Transaction Processing(Ch.20)

innovate achieve lead

Content

- *What is Transaction Model*
- *Significance of Transaction Model*
- *States of a transaction*
- *ACID Properties*
- *Concurrent Transactions*
- *Transaction Schedule*
- *Serial and Concurrent Schedules*

Railway Reservation

Reserve x number of berths

1. If $x \leq A$
2. $A = A - x$
3. Make payment for x berths
4. Print tickets
5. If $x > A$, Then give error message "required berths not available"

Database

Ex

If $x \leq A$

$A = A - x$

make payment

print ticket - Done

else print message

"Required berths not available"

User1

$x = 5$

$x \leq A$ True

$A = 10 - 5 = 5$

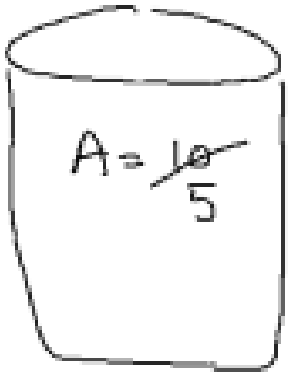
— X —

System crash

User2

$x = 10$

$x \leq A$ false



This is because the berths (5 no) tried by User1 are not given to him but changes are done to database. Thats why U2 is not able to get them.

Transaction Model

A *Transaction* is an executing program that forms a logical unit of database processing.

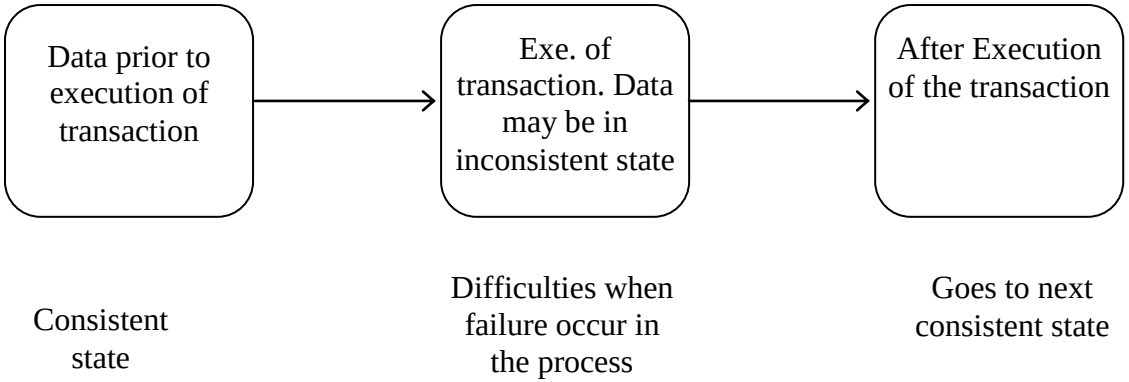
A transaction includes one or more database access operations – read/write/delete etc.

A *transaction* is an atomic unit of work that is either completed in its entirety or not done at all.

For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.

A *transaction* is a program unit that access and update several data items.

Read ( ) *and* *Write ( )* are the basic operations.



Hence, as a result of failure, state of the system will not reflect the state of the real world that the database is supposed to capture.

We call that state as *inconsistent state*.

It is important to define transactions such that they preserve consistency.

ACID Properties of a Transaction

Transaction should possess the following properties called as ***ACID*** properties**.**

***A**tomicity:* A transaction is an atomic unit of processing. It is either performed in its entirety or not performed at all.

***C**onsistency Preservation:* The successful execution of a transaction takes the database from one consistent state to another.

***I**solation:* A transaction should be executed as if it is not interfered by any other transaction.

***D**urability:* The changes applied to the data by a transaction must be permanent.

Transaction States

Active State: Initial state when a transaction starts.

Partially committed State: This state is reached when the last statement is executed and the outcome is not written to the DB.

Failed State: After discovering that the normal execution cannot be continued a transaction is aborted and reaches failed state.

Comitted State: Is reached after successful completion of the transaction.

Terminated State: Is reached after failure or success of the transaction.

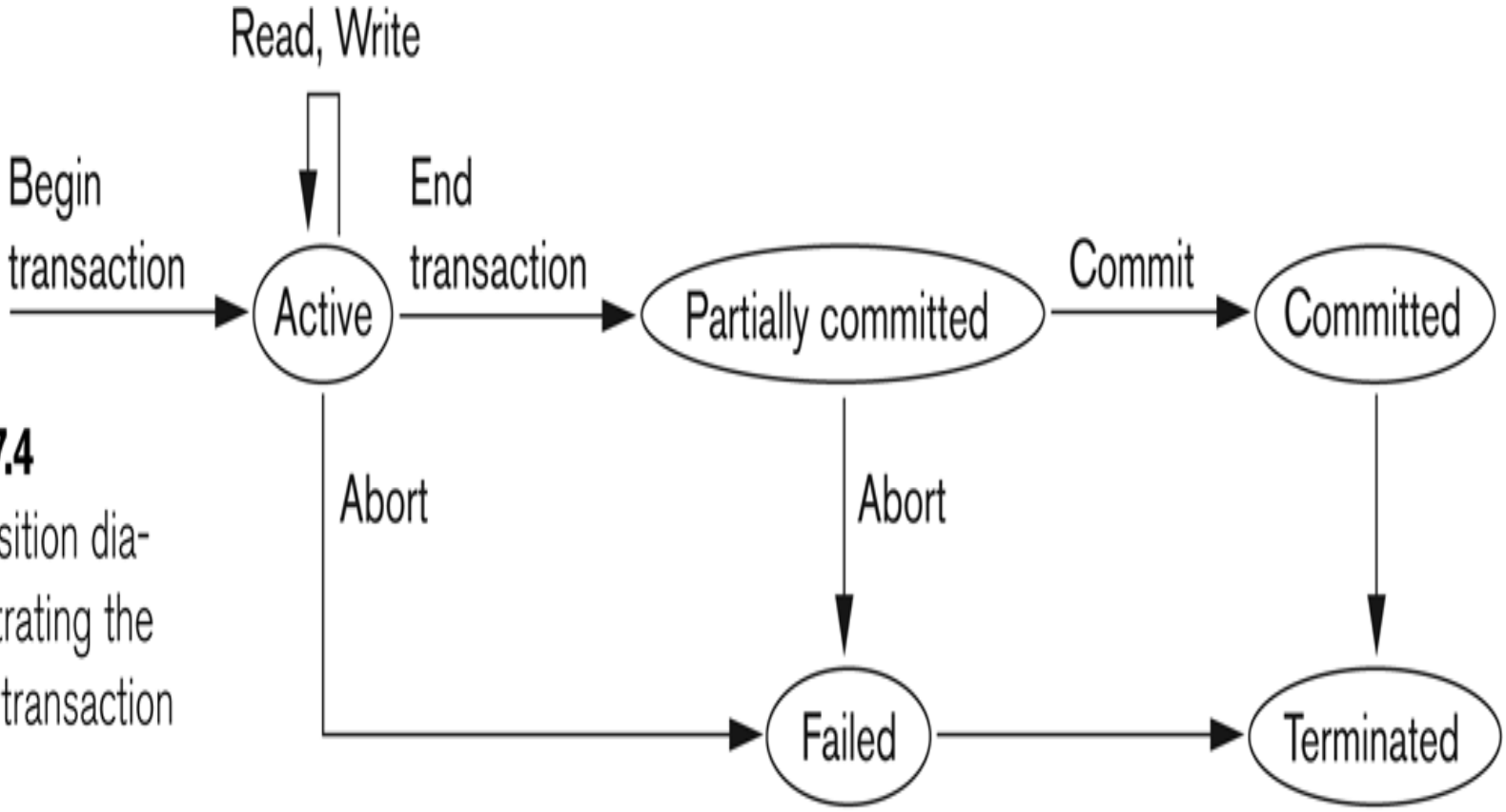


Figure 17.4
State transition diagram illustrating the states for transaction execution.

Transaction Schedule and Concurrency

The descriptions that specify the execution sequence of instructions in a set of transactions are called as *schedules*.

Hence schedule can describe the execution sequence of more than one transaction.

Here, in the above schedule the transactions T_1 & T_2 are executed in a serial manner i.e., first all the instructions of the transaction T_1 are executed, and then the instructions of T_2 are executed. Hence the above schedule is known as *serial schedule*.

T_1	T_2
Read (A) A = A + 50 Read (B) B = B + A Write (B)	Read (B) B = B + 75 Write (B)

T_1 and T_2 are transaction

Read (A) – Reads data item A

Write (B) – Writes the data item B

In a *serial schedule*, instructions belonging to one single transaction appear together.

A serial schedule does not exploit the concurrency. Hence, it is less efficient.

If the transactions are executed *concurrently* then the resources can be utilized more efficiently hence more throughput is achieved.

A *serial schedule* always results in correct database state that reflect the real world situations.

When the instructions of different transactions of a schedule are executed in an interleaved manner use call such schedules are called *concurrent schedules*.
This kind of concurrent schedules may result in incorrect database state.

Problems with Concurrency

Figure 17.3

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

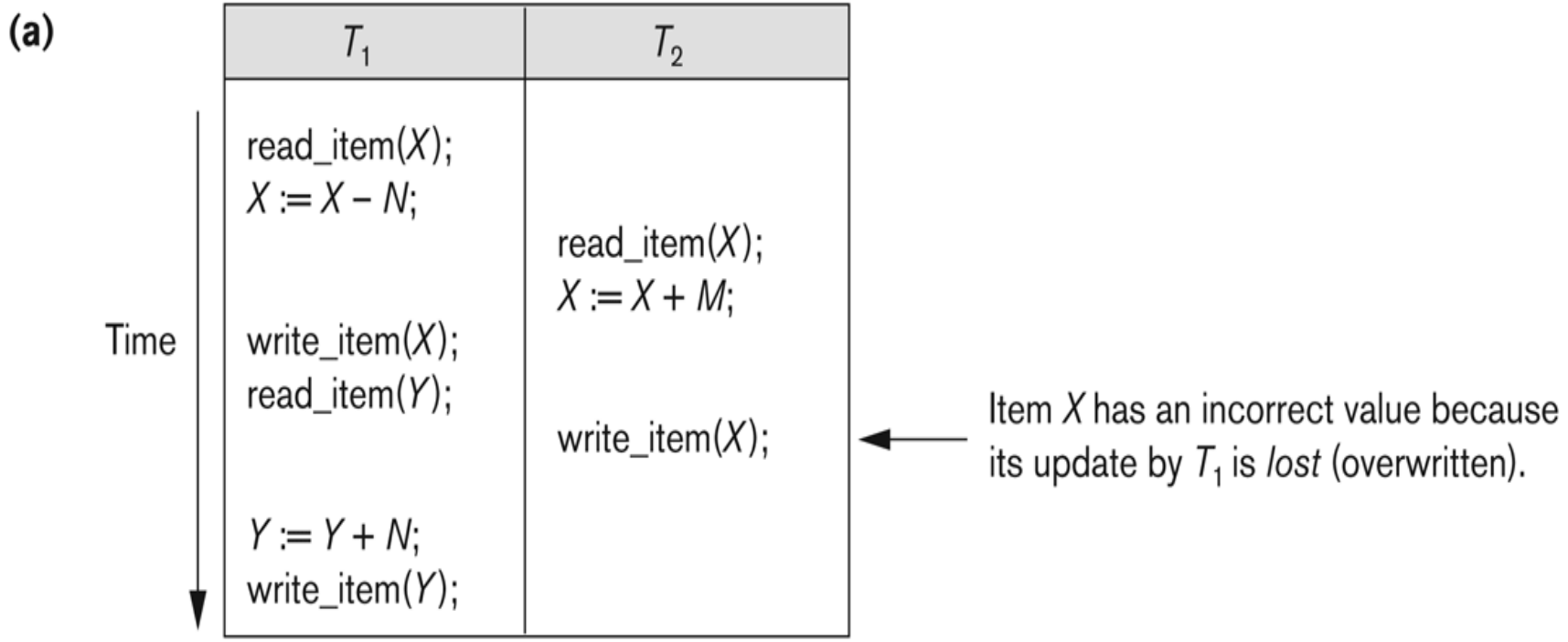


Figure 17.3

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(b)

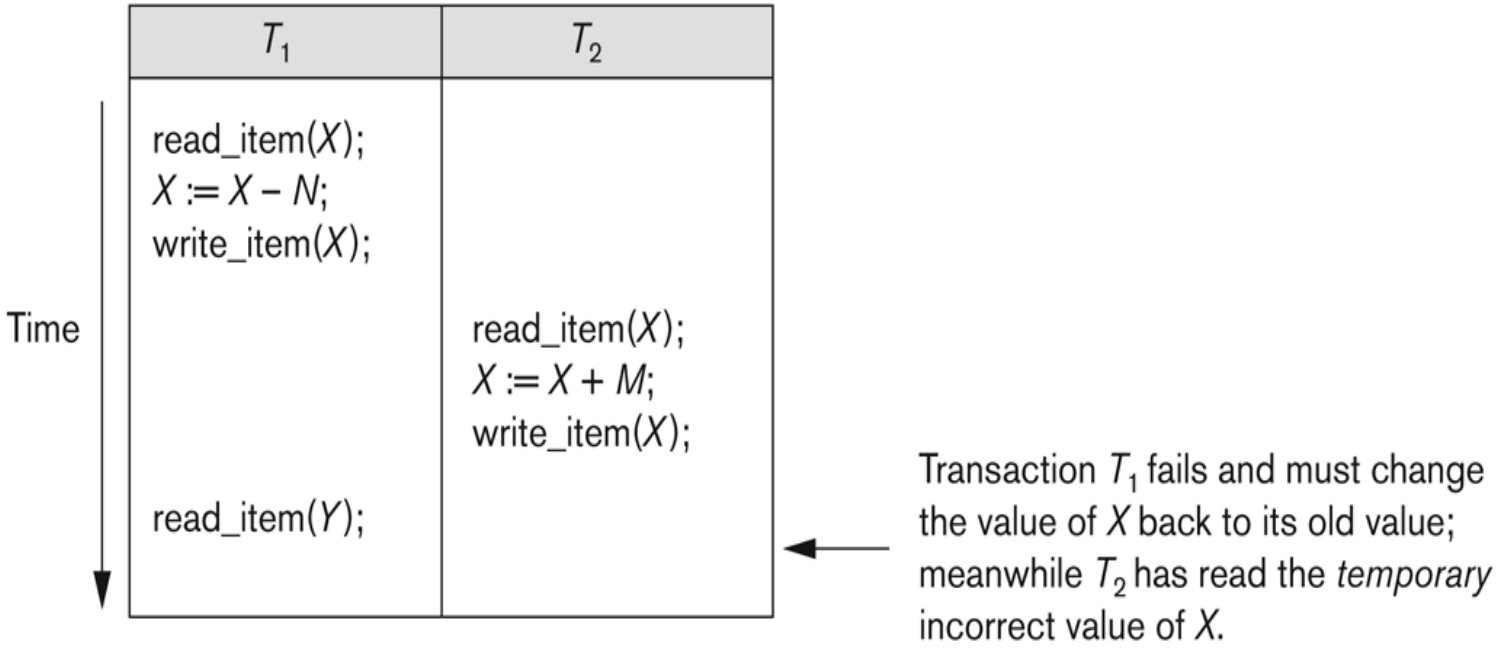


Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(c)

T_1	T_3

	sum := 0;
	read_item(*A*);
	sum := *sum* + *A*;
	• • •

read_item(*X*);
X := *X* − *N*;
write_item(*X*);	

	read_item(*X*);
	sum := *sum* + *X*;
	read_item(*Y*);
	sum := *sum* + *Y*;

read_item(*Y*);
Y := *Y* + *N*;
write_item(*Y*);	

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

Time

T_1	T_2
Read (A)	
A = A + 50	
	Read (A)
	A = A + 100
Write (B)	
	Write (A)

Concurrent Schedule

T_1	T_2
Read (A)	
A = A + 50	
Read (B)	
B = B + A	
Write (B)	
	Read (B)
	B = B + 75
	Write (B)

Seral Schedule

T_1 and T_2 are transaction

Read (A) – Reads data item A

Write (B) – Writes the data item B

Conflicting Pair of operations in a schedule ***S***:

1. Must belong to two different transactions.
2. Must access same data item.
3. One of them must be write operation.

Conflict Equivalent Schedules

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*.

Further, we say that a schedule *S* is *conflict serializable* if it is conflict equivalent to a serial schedule.

Conflict Serializable schedules

Hence it is evident that if we swap non-conflicting operations of a concurrent schedule, it will not affect the final result.

Look at the following example.

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

(S_1)

Concurrent schedule with T_1 & T_2 accessing A, B (data item)

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

(S_2)

Swap W(A) in T_2 with R(B) in T_1 (because they are non conflicting)

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

(S_3)

Swap R(A) of T_2 with R(B) of T_1 and W(A) in T_2 with W(B) in T_1 (Since non conflicting)

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

(S_4)

Swap R(A) of T_2 with W(B) of T_1

Now, the final schedule is a serial schedule

In this example, S_4 in the above example is a serial schedule and is conflict equivalent to S_1. Hence S_1 is a conflict serializable schedule.

T_1	T_2
R(θ)	
	W(θ)
W(θ)	

In this schedule we cannot perform any swap between instructions of T_1 and T_2. Hence it is not conflict serializable

Test for Conflict Serializability

Let S be a schedule.

We construct a precedence graph.

Each transaction participating in the schedule will become a vertex.

The set of edges consist of all edges $T_i \rightarrow T_j$ for which one of the following three conditions hold-

T_i executes W(Q) before T_j executes R(Q)
T_i executes R(Q) before T_j executes W(Q)
T_i executes W(Q) before T_j executes W(Q)

T_0	T_1
R(A)	
	R(A)
W(A)	
	W(A)
	R(B)
R(B)	
W(B)	

Infact their schedule is non conflict serializable

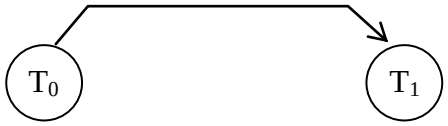
T_1 writes(A) after T_0 writes(A) hence we draw on edge from $T_0 \rightarrow T_1$
T_1 reads(B) before T_0 write (B) hence we can draw an edge from $T_1 \rightarrow T_0$

At any moment of time, while developing the graph in the above manner, if we see a cycle then the schedule is *not conflict serializable*. If no cycles at the end, then it is *conflict serializable.* Hence the above schedule is not serializable.

Now let us consider the following transaction which is *conflict serializable* and discussed earlier.

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

Now, let us draw a precedence graph for the above schedule –
To write A before T_1 reads A. hence we have $T_0 \rightarrow T_1$.



We have only one edge in this graph, and no cycles. Hence it is *conflict serializable*.

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 nonserial schedule.

Serializable schedule:

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

- Being serializable is not the same as being serial
- Being serializable implies that the schedule is a correct schedule.
  - It will leave the database in a consistent state.
  - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

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, for each data item Q,
  1. If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.
  2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S'.

A schedule *S* is ***view serializable*** if it is view equivalent to a serial schedule.

- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

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

Recoverable Schedules

Recoverable schedule — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.

- The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read

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

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

Cascading Rollbacks

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)
abort		

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.

Cascadeless Schedules

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

Weak Levels of Consistency

Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable

- E.g. a read-only transaction that wants to get an approximate total balance of all accounts
- E.g. database statistics computed for query optimization can be approximate.
- Such transactions need not be serializable with respect to other transactions.

- Tradeoff accuracy for performance

A schedule S is *recoverable* if no transaction T commits until all transactions T' that have written an item that T reads has committed.

Otherwise it is *unrecoverable*.

Schedule S1: :

{$r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $commit_2$; $w_1(Y)$; c_1}=Is recoverable.

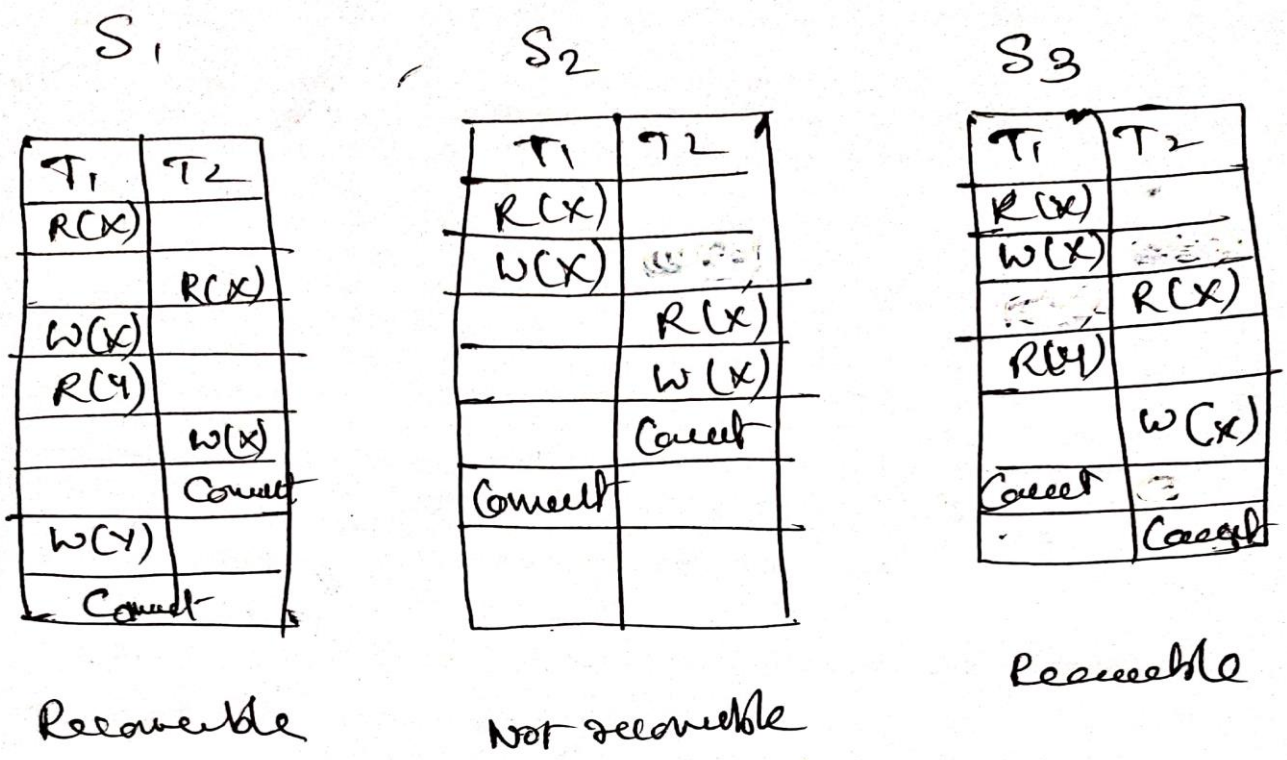
Schedule S2: :

{$r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $commit_2$; $commit_1$ }=Is unrecoverable.

Schedule S3: :

{$r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; c_2 ;c_2}=Is recoverable.

In a recoverable schedule no committed transaction is ever required to roll-back.



Schedule S1: :

{$r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $commit_2$; $w_1(Y)$; c_1}=Is recoverable.

Schedule S2: :

{$r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $commit_2$; $commit_1$ }=Is unrecoverable.

Schedule S3: :

{$r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; c_2 ;c_2}=Is recoverable.

Problem solving

Look at the following schedules for the concurrent transactions 1 and 2. The data items are A and B.

Schedule 1 : $r_1(A)$; $r_1(B)$; $r_2(A)$; $w_1(B)$; $w_2(A)$; $w_1(B)$; $R_2(B)$; $r_1(B)$; $w_2(A)$; $r_1(A)$;

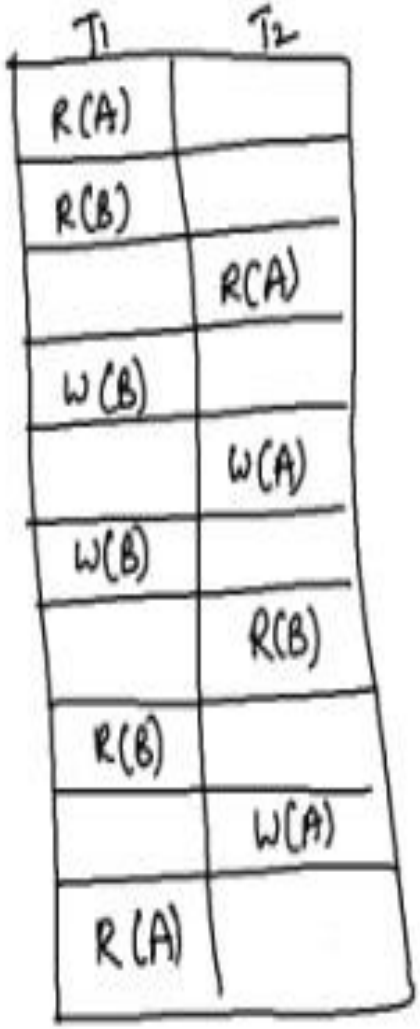
Schedule 2 : $r_1(B)$; $r_1(A)$; $r_2(A)$; $w_1(B)$; $w_2(A)$; $R_2(A)$; $w_1(B)$; $r_2(B)$; $R_1(B)$; $w_2(A)$;

Here, $r_1(A)$; - means that the transaction-1 reads data item A

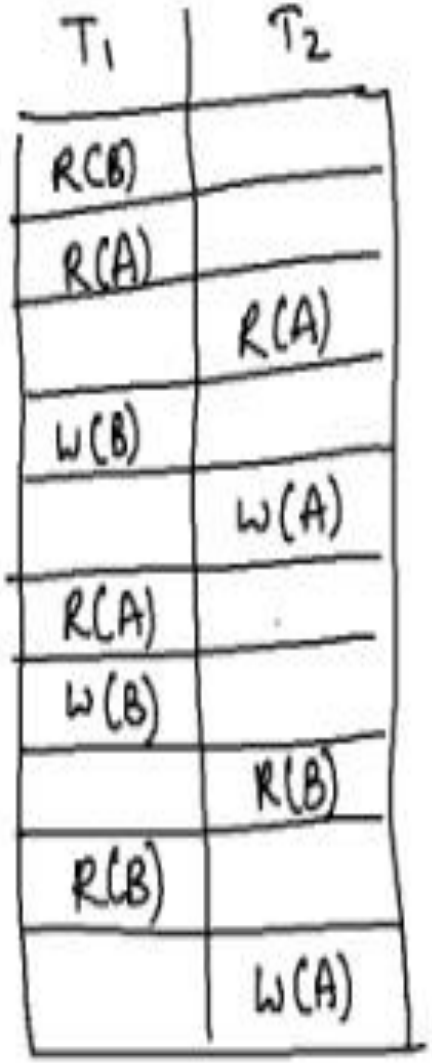
$w_2(B)$; - means that the transaction-2 reads data item B

For each of the above schedules draw the precedence graph and find if they are conflict serializable.

Schedule 1



Schedule-2



Summary

- ✓ *What is a transaction*
- ✓ *Properties of a transaction*
- ✓ *States of a transaction*
- ✓ *Transaction execution and the database consistency*
- ✓ *What are concurrent Schedules*
- ✓ *What are Serial and Concurrent Schedules*
- ✓ *Recoverabilty*