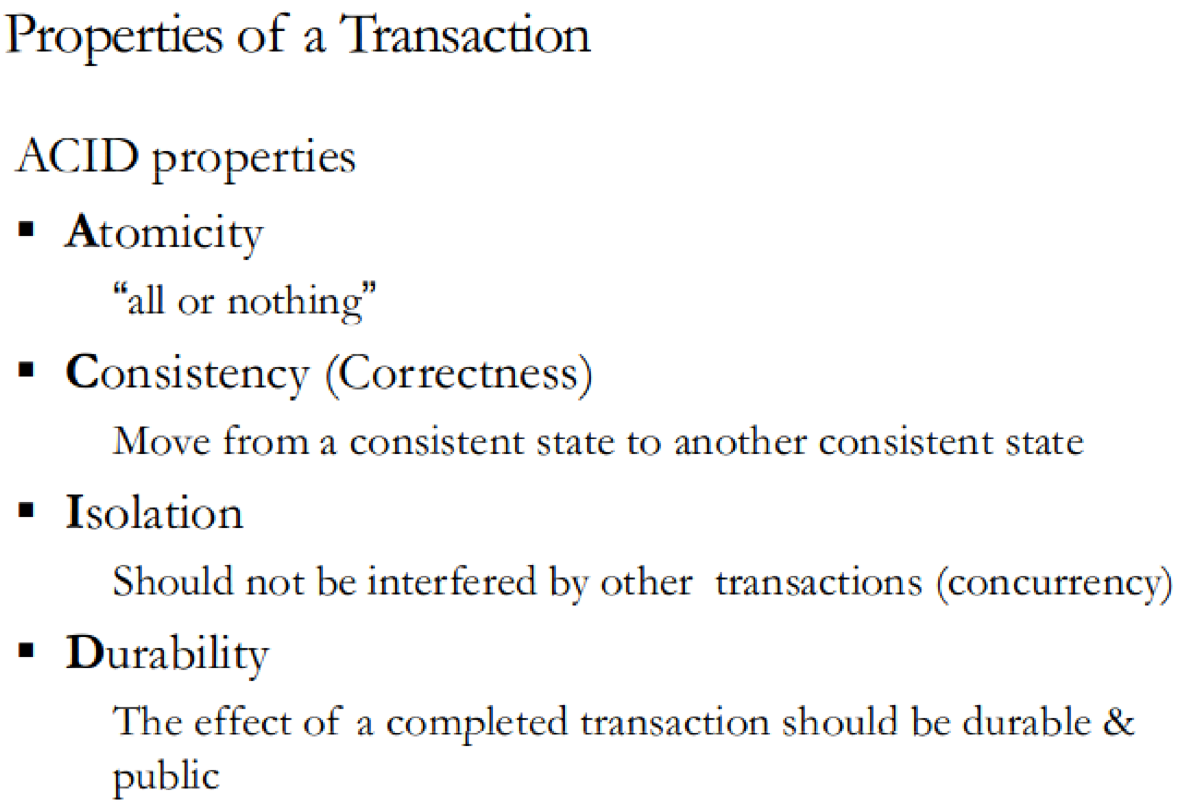


Properties of a Transaction

ACID properties

- **Atomicity**

  "all or nothing"
- **Consistency (Correctness)**

  Move from a consistent state to another consistent state
- **Isolation**

  Should not be interfered by other transactions (concurrency)
- **Durability**

  The effect of a completed transaction should be durable & public

Transaction Atomicity and Durability

- ***a transaction may not always complete its execution successfully-aborted***

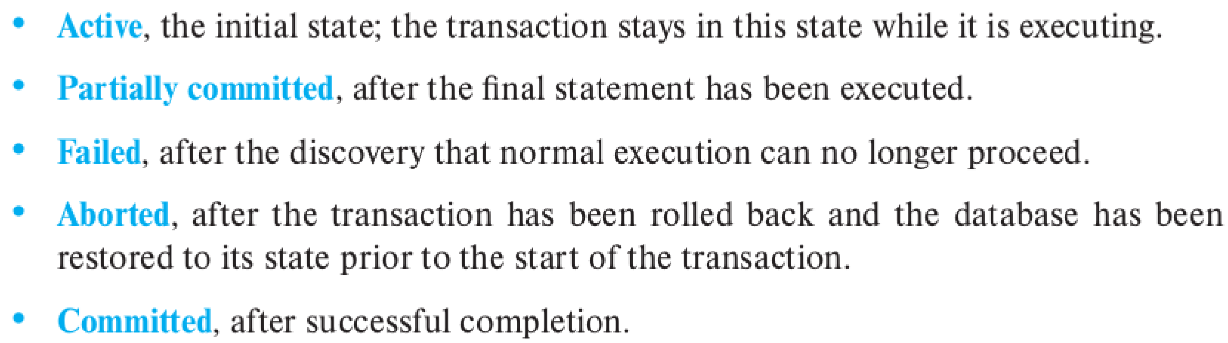
- ***Once the changes caused by an aborted transaction have been undone, we say that the transaction has been rolled back(responsibility of the recovery scheme)***

- ***Each database modification made by a transaction is first recorded in the log.***

  - ➢ ***identifier of the transaction***
  - ➢ ***identifier of the data item being modified,***
  - ➢ ***both the old value (prior to modification)***
  - ➢ ***the new value (after modification) of the data item***

- ***A transaction that completes its execution successfully is said to be committed***

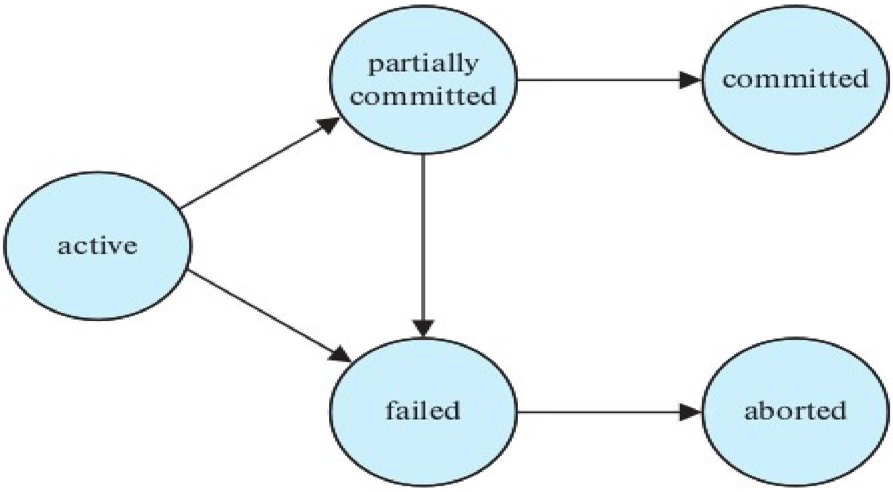
- ***Once a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a compensating transaction***

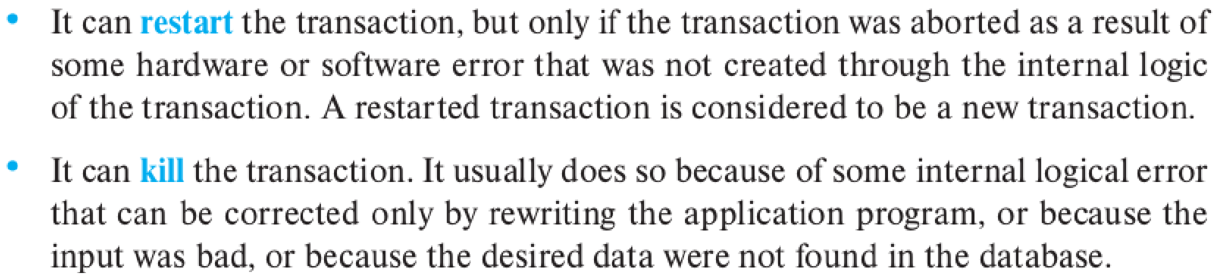


- **Active**, the initial state; the transaction stays in this state while it is executing.
- **Partially committed**, after the final statement has been executed.
- **Failed**, after the discovery that normal execution can no longer proceed.
- **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- **Committed**, after successful completion.

A transaction is said to have terminated if it has either committed or aborted.

After abort state,





- It can **restart** the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
- It can **kill** the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

observable external writes, such as writes to a user's screen, or sending email.

Most systems allow such writes to take place only after the transaction has entered the committed state.

Transaction Isolation

- ***Transaction-processing systems usually allow multiple transactions to run concurrently.***
- ***Allowing multiple transactions to update data concurrently causes several complications with consistency of the data***
- ***it is far easier to insist that transactions run serially.***
- ***But there are two good reasons for allowing concurrency.***

1. ***Improved throughput and resource utilization.***

   - ***The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel.***
   - ***increases the throughput of the system— that is, the number of transactions executed in a given amount of time***
   - ***the processor and disk utilization also increase***

2. ***Reduced waiting time.***
   - ***If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete***
   - ***Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the average response time: the average time for a transaction to be completed after it has been submitted.***

concurrency-control schemes:database system must control the interaction among the concurrent trans-actions to prevent them from destroying the consistency of the database.

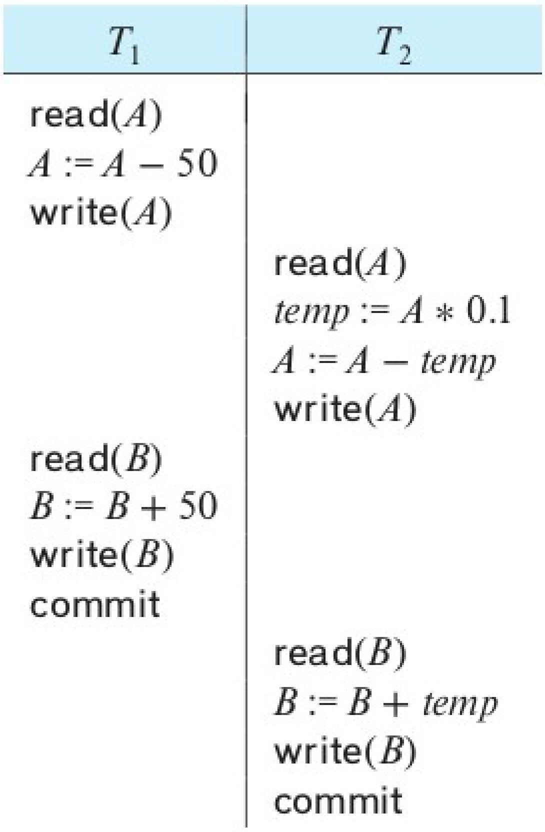
schedules-a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed

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

2 Schedule 1—a serial schedule in which T_1 is followed by T_2.

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

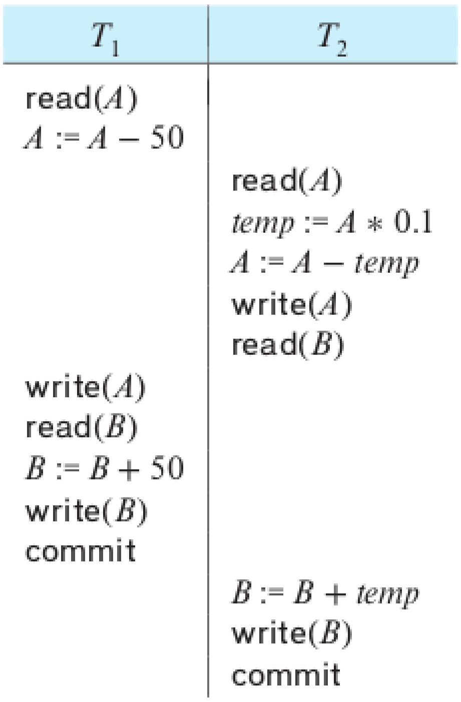
2 Schedule 1—a serial schedule in which T_1 is followed by T_2.



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

Schedule 3—a concurrent schedule equivalent to schedule 1.

It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent state. The concurrency-control component of the database system carries out this task.



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

Schedule 4—a concurrent schedule resulting in an inconsistent state.

Conflict Serializable: A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

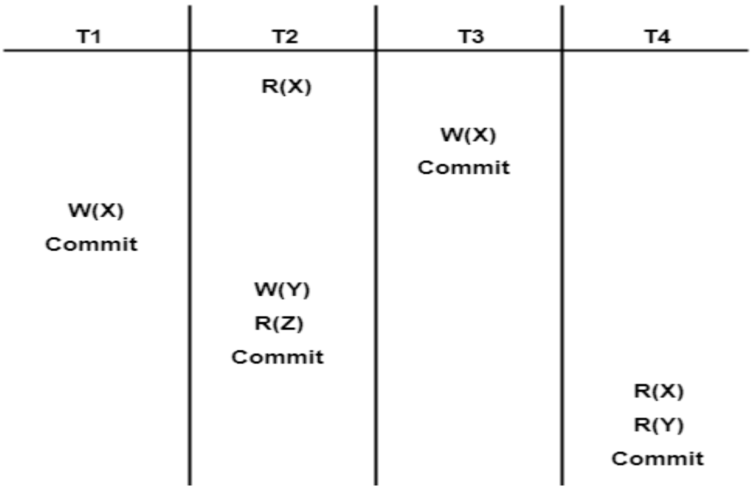
Conflicting operations: Two operations are said to be conflicting if all conditions satisfy:

- They belong to different transactions
- They operate on the same data item
- At Least one of them is a write operation

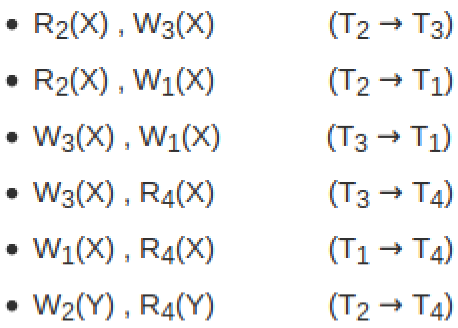
Conflict Serializable: A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

Conflicting operations: Two operations are said to be conflicting if all conditions satisfy:

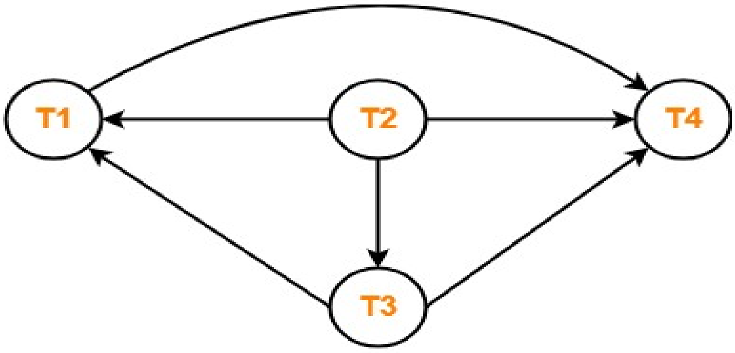
- They belong to different transactions
- They operate on the same data item
- At Least one of them is a write operation



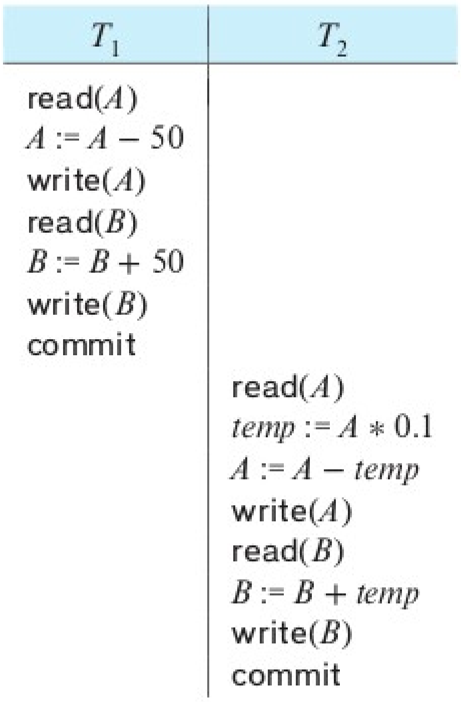
T1	T2	T3	T4
	R(X)		
		W(X)	
		Commit	
W(X)			
Commit			
	W(Y)		
	R(Z)		
	Commit		
			R(X)
			R(Y)
			Commit



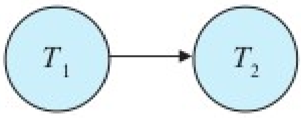
• $R_2(X)$, $W_3(X)$	$(T_2 \rightarrow T_3)$
• $R_2(X)$, $W_1(X)$	$(T_2 \rightarrow T_1)$
• $W_3(X)$, $W_1(X)$	$(T_3 \rightarrow T_1)$
• $W_3(X)$, $R_4(X)$	$(T_3 \rightarrow T_4)$
• $W_1(X)$, $R_4(X)$	$(T_1 \rightarrow T_4)$
• $W_2(Y)$, $R_4(Y)$	$(T_2 \rightarrow T_4)$



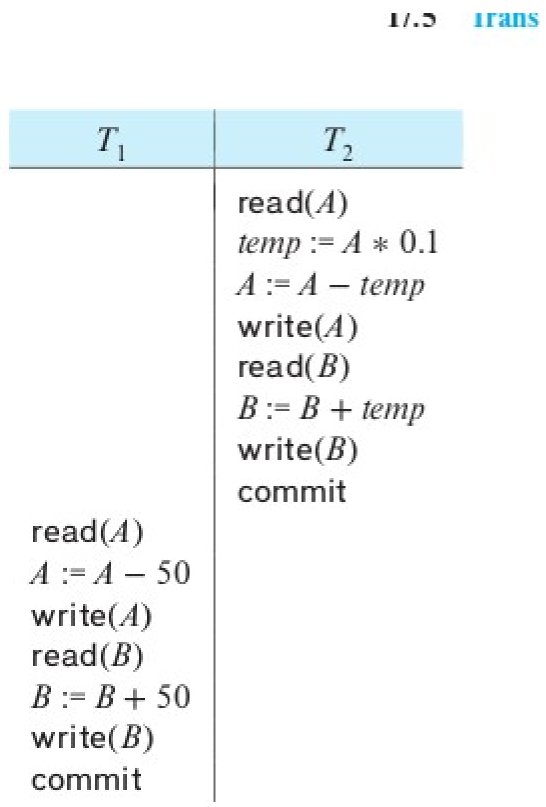
https://www.gatevidyalay.com/conflict-serializability-practice-problems/



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

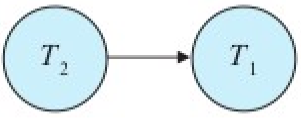


Schedule is conflict serializable schedule.

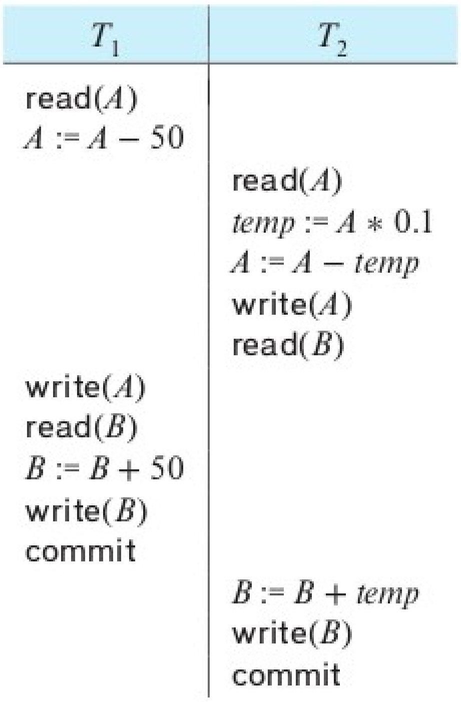


17.5 Trans

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



***Schedule is conflict serializable schedule*.**



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

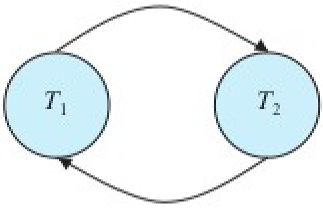


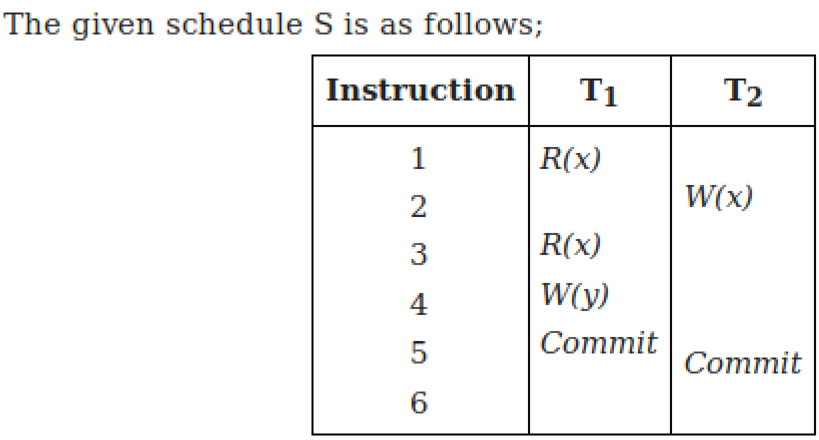
Figure 17.11 Precedence graph for schedule 4.

Schedule 4 is not conflict serializable schedule.

Consider a schedule S with two transactions T_1 and T_2 as follows;

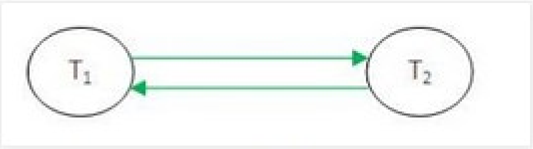
S: $R_1(x);W_2(x);R_1(x);W_1(y)$;commit1;commit2;

Is the schedule S conflict serializable?

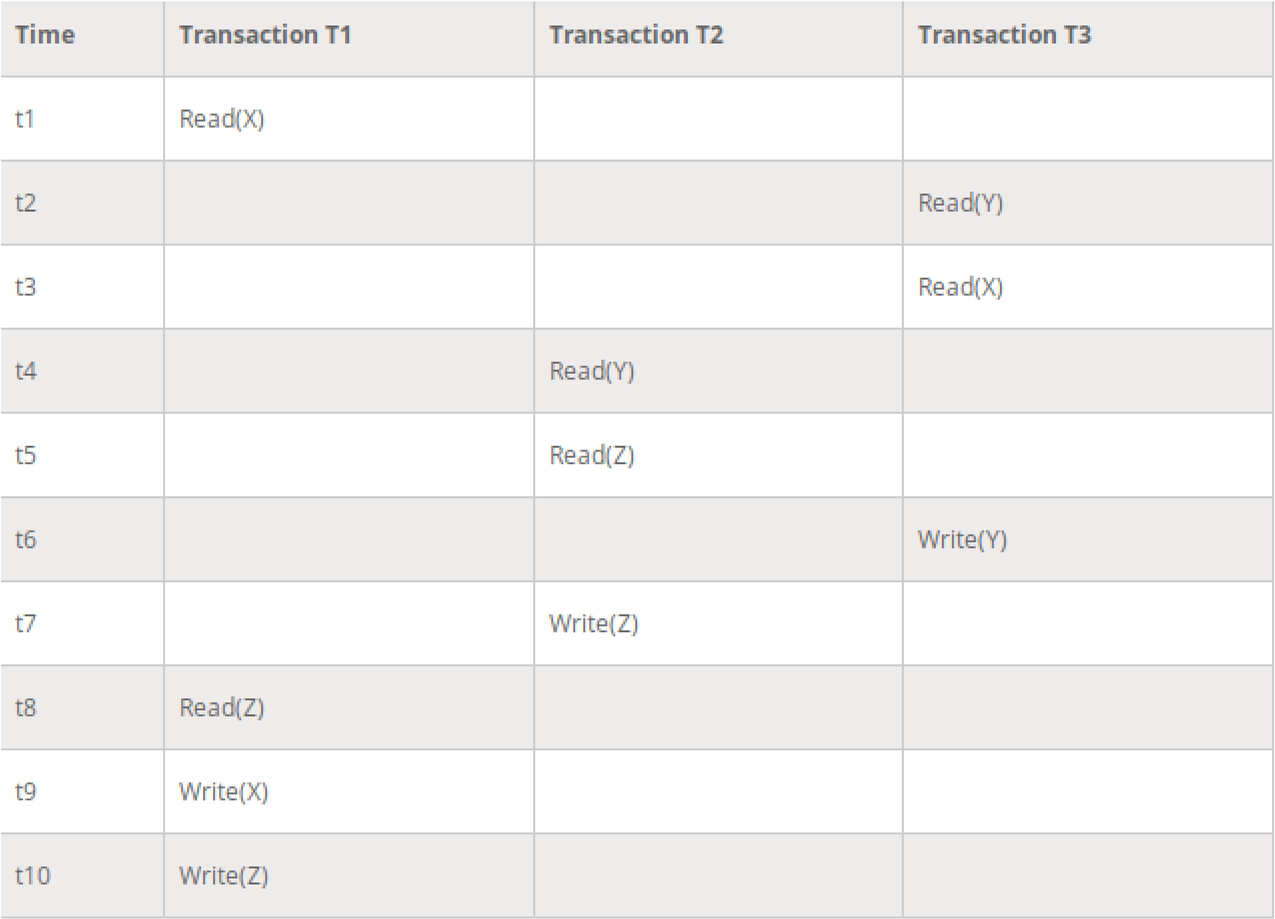


The given schedule S is as follows;

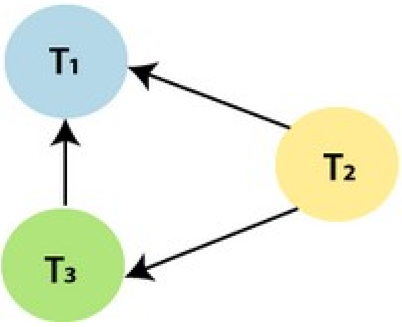
Instruction	T_1	T_2
1	*R(x)*	
2		*W(x)*
3	*R(x)*	
4	*W(y)*	
5	*Commit*	
6		*Commit*



Schedule S is not conflict serializable schedule.

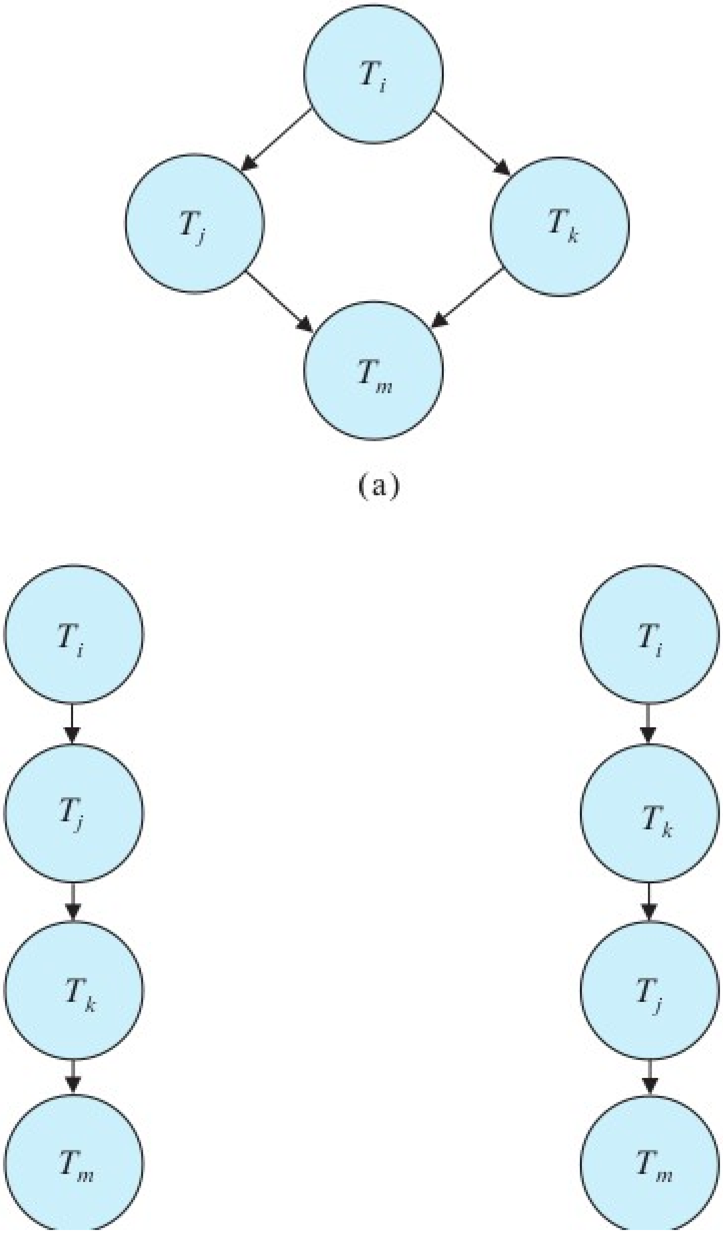


Time	Transaction T1	Transaction T2	Transaction T3
t1	Read(X)		
t2			Read(Y)
t3			Read(X)
t4		Read(Y)	
t5		Read(Z)	
t6			Write(Y)
t7		Write(Z)	
t8	Read(Z)		
t9	Write(X)		
t10	Write(Z)		



<u>schedule is conflict serializable</u>

A serializability order of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called topological sorting.

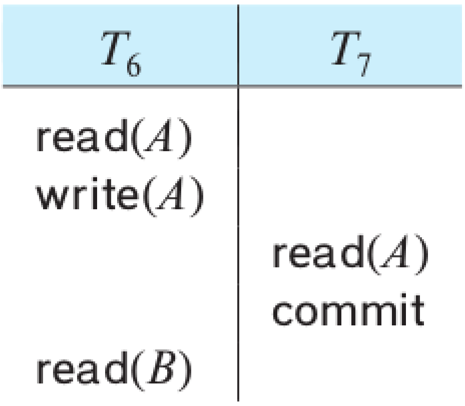


(a)

Transaction Isolation and Atomicity

If a transaction T i fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction

Recoverable Schedules



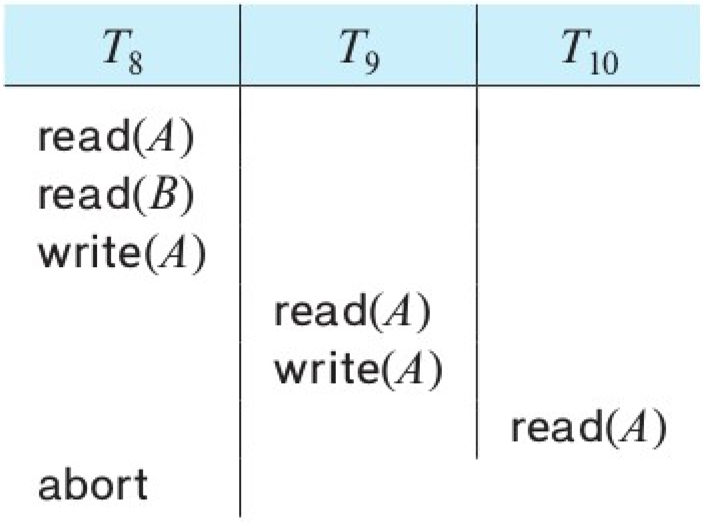
T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

Schedule 9, a nonrecoverable schedule.

A recoverable schedule is one where, 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 commit operation of T j .

Cascadeless Schedules

phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called cascading rollback.



T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

Formally, a cascadeless schedule is one where, 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 .

Transaction Isolation Levels

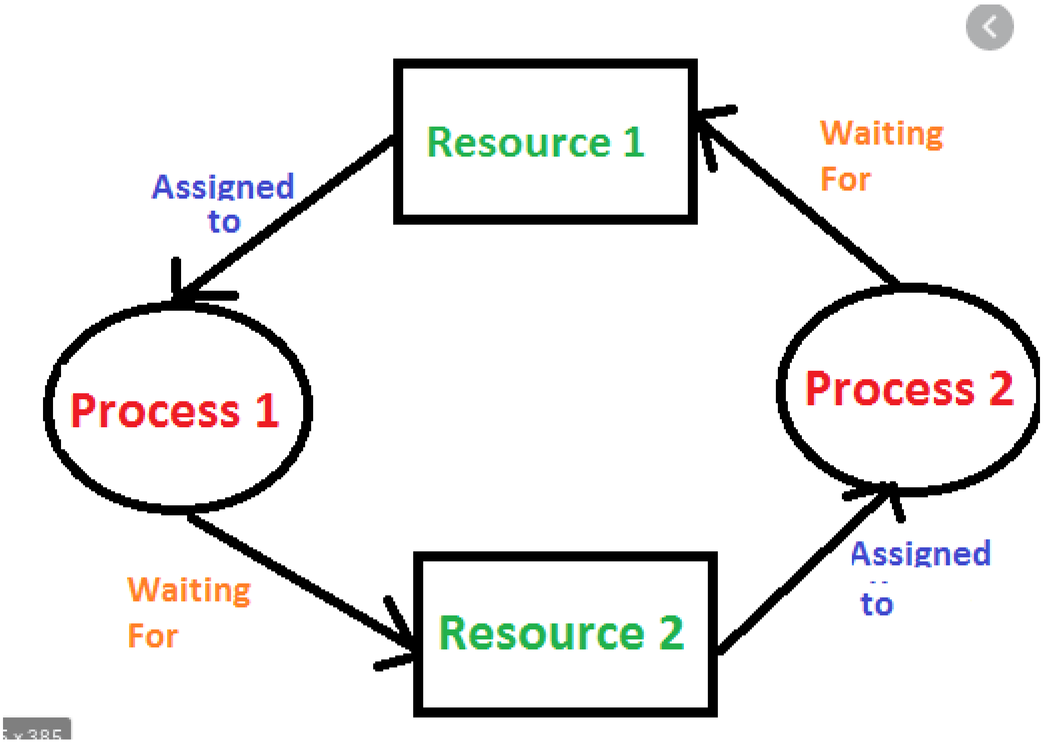
- ***concurrency-control policies** that we can use to ensure that, even when multiple transactions are executed concurrently, only acceptable schedules are generated.*

- ***The goal of concurrency-control policies is to provide a high degree of concurrency,while ensuring that all schedules that can be generated are conflict or view serializable,recoverable, and cascadeless.***

The isolation levels specified by the SQL standard are as follows:

1. ***Serializable***
2. ***Repeatable read***
3. ***Read committed***
4. ***Read uncommitted***

deadlock


Resource 1
Assigned to
Waiting For
Process 1
Process 2
Waiting For
Resource 2
Assigned to