Chapter 14: Transactions

Database System Concepts, 6th Ed.

Chapter 14: Transactions

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

Transactions

A series of one or more SQL statements that are logically related are termed as a Transaction.

E.g:

1. 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*)

synonym for SQL statement

for our study purpose

- A **transaction** is a **unit** of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data-manipulation language
- A transaction is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**. The transaction consists of all operations executed between the **begin transaction** and **end transaction**.
- This collection of steps must appear to the user as a single, indivisible unit. This “all-or-none” property is referred to as **atomicity**.

- the database system must take special actions to ensure that transactions operate properly without interference from concurrently executing database statements. This property is referred to as **isolation.**
- Even if the system ensures correct execution of a transaction, this serves little purpose if the system subsequently crashes and, as a result, the system "forgets" about the transaction. Thus, a transaction's actions must persist across crashes. This property is referred to as **durability.**
- if a transaction is run atomically in isolation starting from a consistent database, the database must again be consistent at the end of the transaction. This property is referred to as **consistency.**

Transfer A ⇒ B = sum(A+B)

ACID Properties

we require that the database system maintain the following properties of the transactions:

- **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.
- **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 *Ti* and *Tj,* it appears to *Ti* that either *Tj,* finished execution before *Ti* started, or *Tj* started execution after *Ti* finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

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. $B := B + 50$
   write(B)

Atomicity requirement

if the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state

Failure could be due to software(R.Error in Query) or hardware(S/W crash)

the system should ensure that updates of a partially executed transaction are not reflected in the database

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 even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

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*)

Consistency requirement in above example:
the sum of A and B is unchanged by the execution of the transaction

In general, consistency requirements include
- Explicitly specified integrity constraints such as primary keys and foreign keys
- Implicit integrity constraints
  - e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand

A transaction must see a consistent database.
During transaction execution the database may be temporarily inconsistent.
When the transaction completes successfully the database must be consistent
- Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 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).

	T1	T2
1.	**read**(A)	
2.	$A := A - 50$	
3.	**write**(A)	
		read(A), read(B), print(A+B)
4.	**read**(B)	
5.	$B := B + 50$	
6.	**write**(B)	

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.

- However, executing multiple transactions concurrently has significant benefits, as we will see later.

Storage Structure

- To understand how to ensure the atomicity and durability properties of a transaction, we must gain a better understanding of how the various data items in the database may be stored and accessed.
- **Volatile storage:** Information residing in volatile storage does not usually survive system crashes. Examples: main memory and cache memory.
- **Nonvolatile storage:** Information residing in nonvolatile storage survives system crashes. Ex: secondary storage devices such as magnetic disk and flash storage, used for online storage, and tertiary storage devices such as optical media, and magnetic tapes, used for archival storage.
- **Stable storage:** Information residing in stable storage is *never lost. Extremely unlikely to lose data- replicate information*

- For a transaction to be durable, its changes need to be written to stable storage.
- Similarly, for a transaction to be atomic, log records need to be written to stable storage before any changes are made to the database on disk.
- Clearly, the degree to which a system ensures durability and atomicity depends on how stable its implementation of stable storage really is.

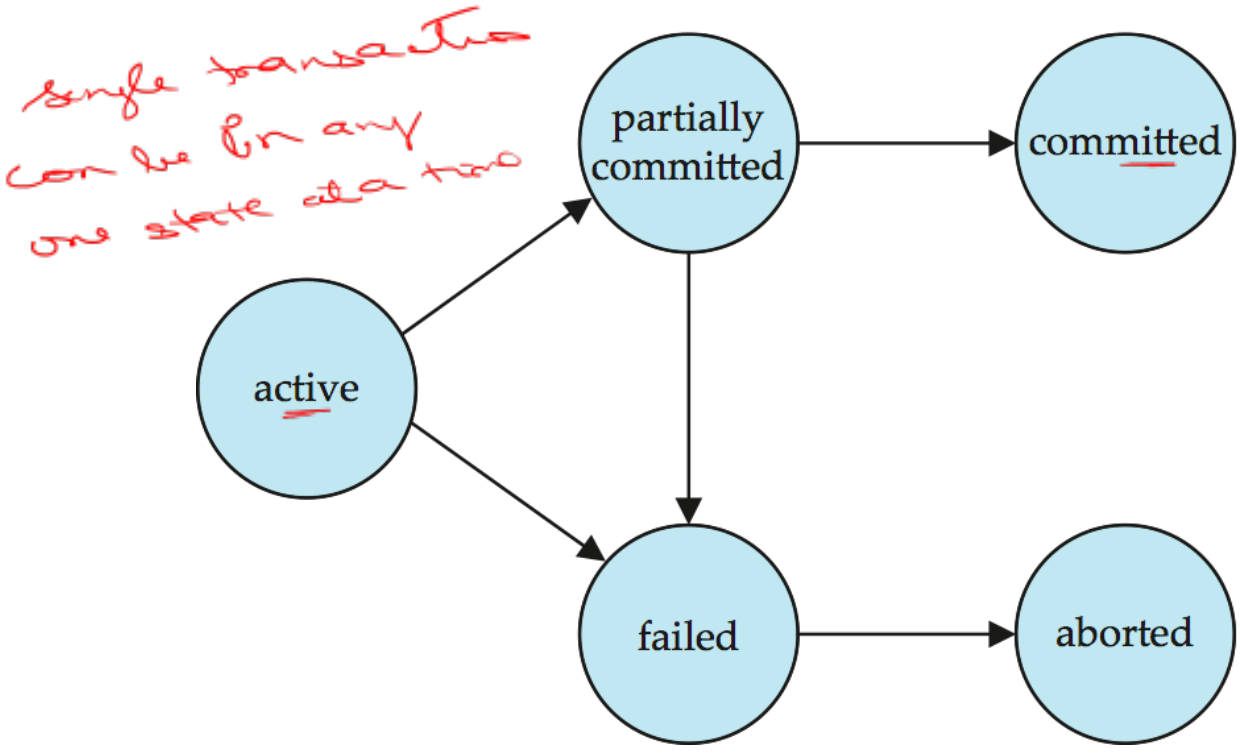
Transaction Atomicity and Durability

- A transaction may not always complete its execution successfully. Such a transaction is termed as **aborted**.
- Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**.
- It is part of the responsibility of the recovery scheme to manage transaction aborts. This is done typically by maintaining a **log**.
- Maintaining a log provides the possibility of redoing a modification to ensure atomicity and durability as well as the possibility of undoing a modification to ensure atomicity in case of a failure during transaction execution.
- A transaction that completes its execution successfully is said to be **committed**. The only way to undo the effects of a committed transaction is to execute a **compensating transaction.**

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
    - can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.

Transaction model



Transaction State (Cont.)

- The system has two options if the transaction is failed:
- 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.

Transaction Concept

COMMIT and ROLLBACK commands

- A transaction begins with the first executable SQL statement after a commit, rollback or connection made to the database.

- A transaction can be closed by using a commit or a rollback statement.

- **COMMIT** ends the current transaction and makes permanent changes made during the transaction
- **ROLLBACK** ends the transaction but undoes any changes made during the transaction

- Syntax: COMMIT; ROLLBACK ;

Example for usage of Commit, Rollback and SAVEPOINT

- Withdraw an amount 2,000 and deposit 10,000 for all the accounts. If the sum of balance of all accounts exceeds 2,00,000 then undo the deposit just made.

```
Declare
        Total_bal numeric;
Begin
      Update account set balance=balance-2000;
      Savepoint deposit;
      Update account set balance=balance+10000;
      Select sum(balance) into total_bal from account;
      If total_bal > 200000 then
            Rollback to savepoint deposit;
      End if;
      Commit;
End;
```

- **SAVEPOINT** marks and saves the current point in the processing of a transaction. When a savepoint is used with a ROLLBACK statement, parts of transaction can be undone.

Transaction isolation

- Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run **serially—that is, one at a time, each** starting only after the previous one has completed.
- Multiple transactions are allowed to run concurrently in the system.
  Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time/waiting time** for transactions: short transactions need not wait behind long ones.

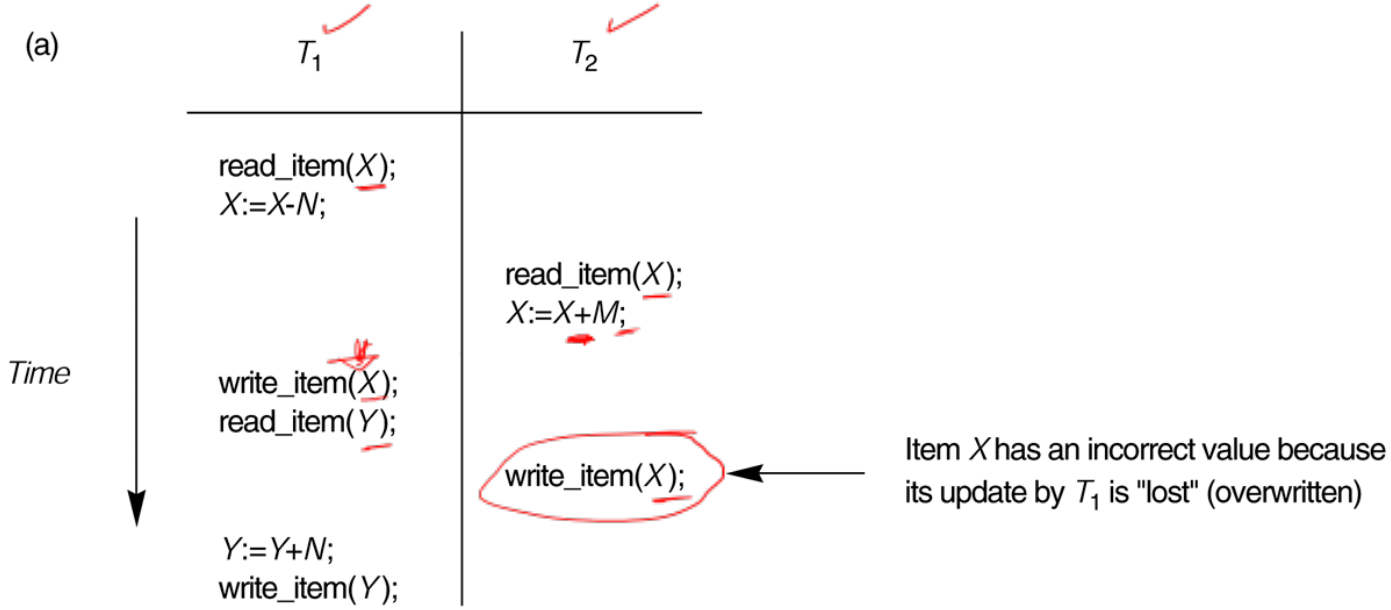
- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

Why Concurrency Control is needed

- Problems occur when concurrent transactions execute in an **uncontrolled** manner:
  - The Lost Update
  - The Temporary Update (or Dirty Read)
  - The Incorrect Summary
  - Unrepeatable Read:
    - A transaction T1 may read a given value. If another transaction later updates that value and T1 reads that value again, then T1 will see a different value.

The Lost Update Problem

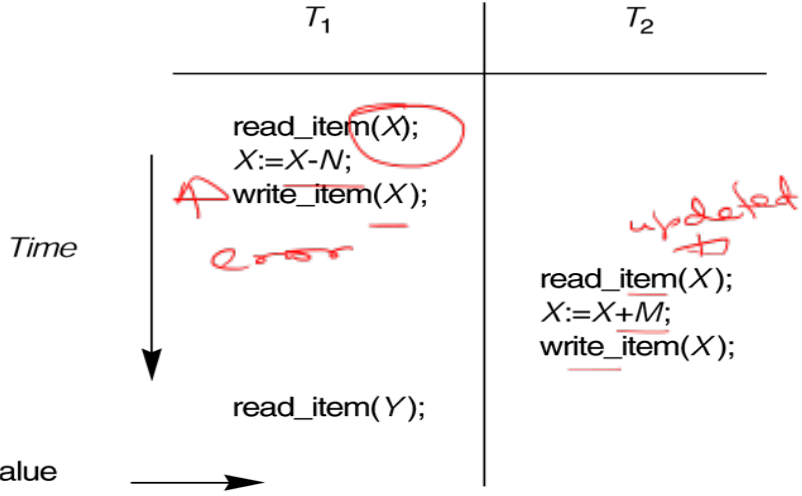
- This occurs when **two** transactions that access the **same** database items have their operations interleaved in a way that makes the value of some database item incorrect.



The Temporary Update Problem (Dirty Read)

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

(b)



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.

Incorrect Summary Problem

- If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

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.

(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).

Schedules

- **Schedule** – a sequences of instructions that specify the chronological 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.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedules

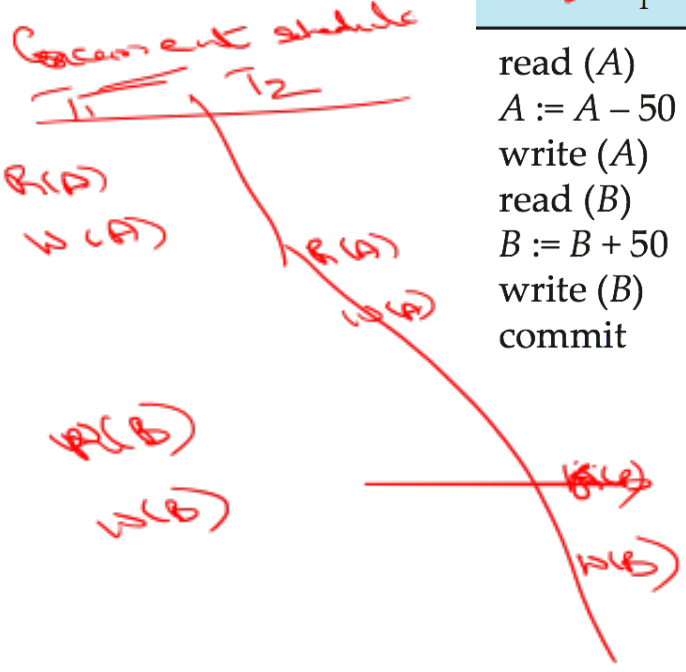
- A ***schedule*** *S* of *n* transactions *T*1, *T*2, ..., *T*n is an ordering of **all** the operations in these transactions subject to the constraint that:
  - for each transaction *Ti*, the operations of *Ti* in *S* must appear in the **same order** as they do in *Ti*.
    - Note, however, that operations from other transactions *Tj* can be interleaved with the operations of *Ti* in *S*.

- Example: Given
  *T*1 = **R**1(*Q*) **W**1(*Q*) & *T*2 = **R**2(*Q*) **W**2(*Q*)
  - a schedule: **R**1(*Q*) **R**2(*Q*) **W**1(*Q*) **W**2(*Q*)
  - not a schedule: **W**1(*Q*) **R**1(*Q*) **R**2(*Q*) **W**2(*Q*)

9/11/2020

Schedule 1

- Let *T*1 transfer $50 from *A* to *B*, and *T*2 transfer 10% of the balance from *A* to *B.*
- 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

Schedule 2

- A serial schedule where *T2* is followed by *T*1

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 3

- Let *T*1 and *T*2 be the transactions defined previously. The following schedule is not a serial schedule (It is CONCURRENT 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)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of ($A + B$). It has Lost Update problems.

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

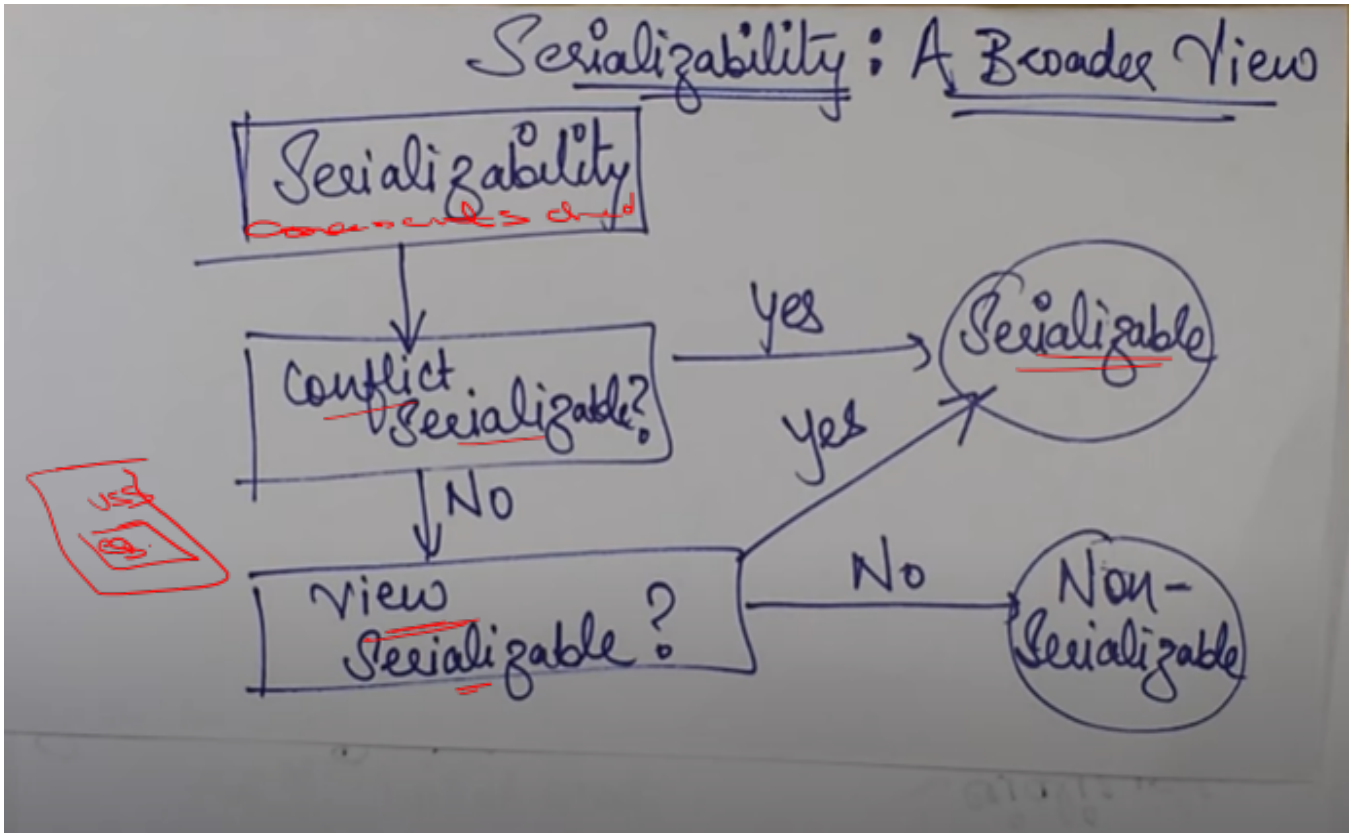
- We can ensure consistency of the database under concurrent execution by making sure that any schedule that is executed has the same effect as a schedule that could have occurred without any concurrent execution.
- That is, the schedule should, in some sense, be equivalent to a serial schedule. Such schedules are called **serializable** schedules.
- serial schedules are serializable, but if steps of multiple transactions are interleaved, it is harder to determine whether a schedule is serializable.
- Since transactions are programs, it is difficult to determine exactly what operations a transaction performs and how operations of various transactions interact.
- For this reason, we shall not consider the various types of operations that a transaction can perform on a data item, but instead consider only two operations: read and write.

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.

- A concurrent schedule is serializable if it is equivalent to a serial schedule.

1. **conflict serializability**
2. **view serializability**


Serializability : A Broader View
Serializability
Conflict Serializable?
yes
Serializable
No
yes
View Serializable ?
No
Non-Serializable

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflict Serializability

Conflicting Instructions

- Instructions *Ii* and *Ij* of transactions *Ti* and *Tj* respectively, **conflict** if and only if there exists some item *Q* accessed by both *Ii* and *Ij*, and at least one of these instructions wrote *Q*.
  1. *Ii* = **read**(*Q*), *Ij* = **read**(*Q*). *Ii* and *Ij* don't conflict.
  2. *Ii* = **read**(*Q*), *Ij* = **write**(*Q*). They conflict.
  3. *Ii* = **write**(*Q*), *Ij* = **read**(*Q*). They conflict
  4. *Ii* = **write**(*Q*), *Ij* = **write**(*Q*). They conflict

- Intuitively, a conflict between *Ii* and *Ij* forces a (logical) temporal order between them.
  - If *Ii* and *Ij* 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.

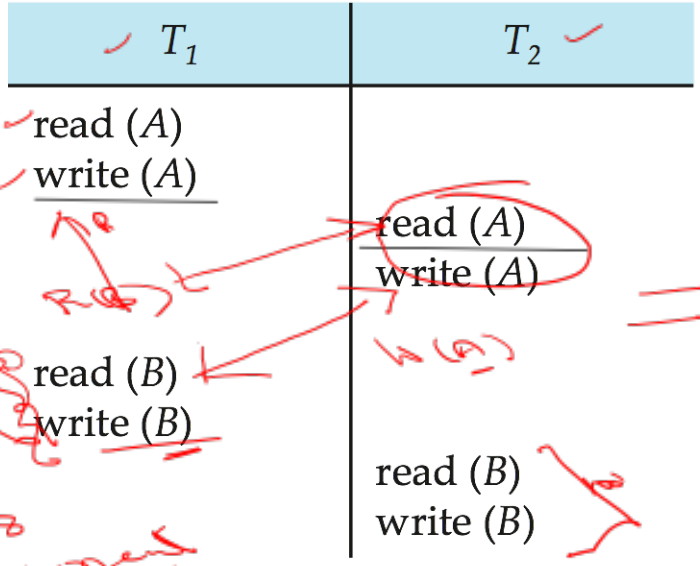
Conflict Serializability

- 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**.

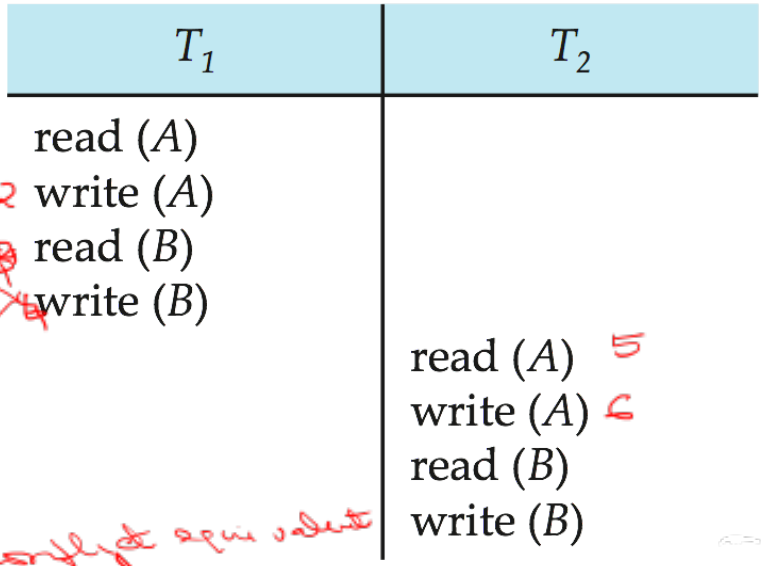
- We say that a schedule S is **conflict Serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where *T*2 follows *T*1, by series of swaps of non-conflicting instructions.
- Therefore Schedule 3 is conflict serializable.



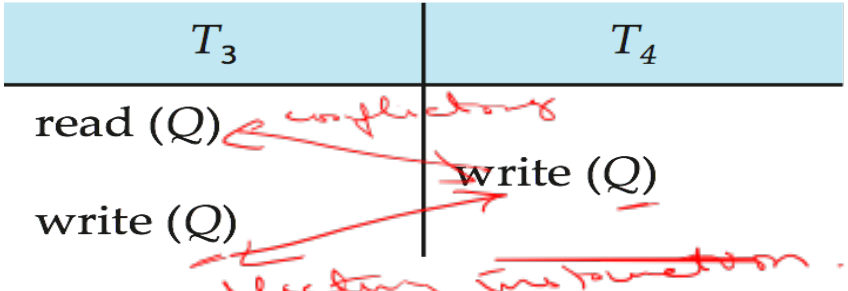
Schedule 3



Schedule 6

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:



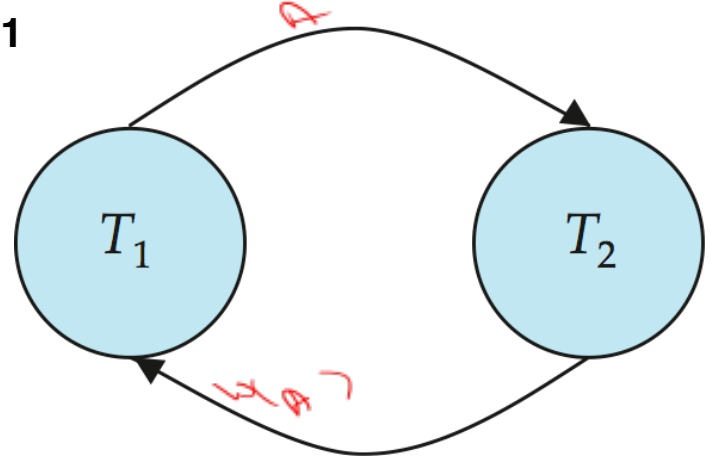
- We are unable to swap instructions in the above schedule to obtain either the serial schedule < $T3$, $T4$ >, or the serial schedule < $T4$, $T3$ >.

Check Conflict Serializable, using Precedence Graph

- We now present a simple and efficient method for determining conflict serializability of a schedule. Consider a schedule *S*. We construct a directed graph, called a **precedence graph**, from *S*.
- This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges
- $T_i \rightarrow T_j$ for which one of three conditions holds:
  1. T_i executes write(Q) before T_j executes read(Q).
  2. T_i executes read(Q) before T_j executes write(Q).
  3. T_i executes write(Q) before T_j executes write(Q).

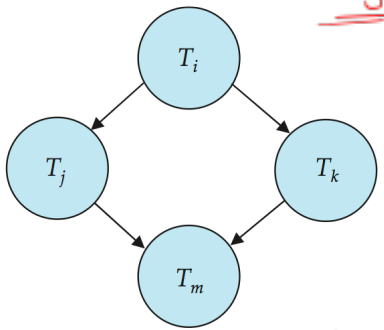
Testing for Serializability

- Consider some schedule of a set of transactions *T*1, *T*2, ..., *Tn*
- **Precedence graph** — a directed graph where the vertices are the transactions (names).
- We draw an arc from *Ti* to *Tj* if the two transaction conflict, and *Ti* accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**

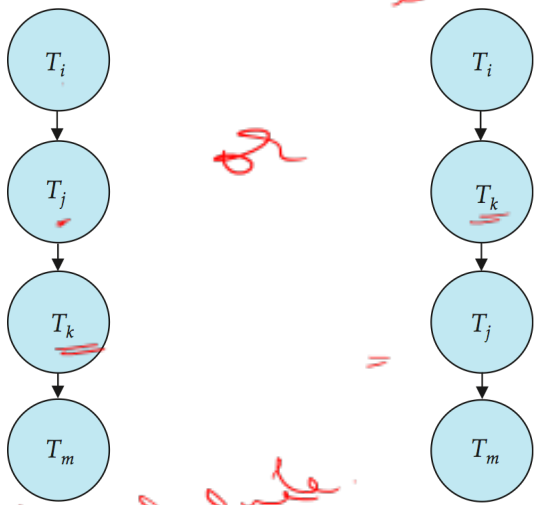


Test for Conflict Serializability

- 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.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be
    *T*5 □ *T*1 □ *T*3 □ *T*2 □ *T*4
  - Are there others?



(a)

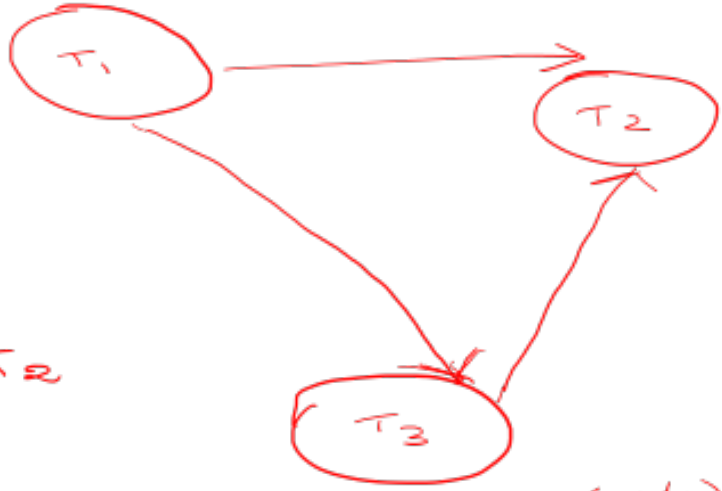


(b) (c)

Concurrent S ⇒ S' serial schedule.

T1	T2	T3
Read (X)		
	Read (Y)	
		Read (Y)
	Write (Y)	
Write (X)		
		Write (X)
	Read (X)	
	Write (X)	

No cycles ∴ conflict serial schedule



Topological sort T_1, T_3, T_2

serial schedule

R(x) W(x) , R(y) W(y) , R(y) W(y) R(x) & W(x)

T_1 T_3 T_2

$T_1 \Rightarrow T_3 \Rightarrow T_2$

T_1	T_2	T_3
$W_1(A)$		
	$R_2(A)$	
		$W_3(A)$

S_1

T_1	T_2	T_3
	$W_2(A)$	
		$W_3(A)$
	$R_2(A)$	
$W_1(A)$		

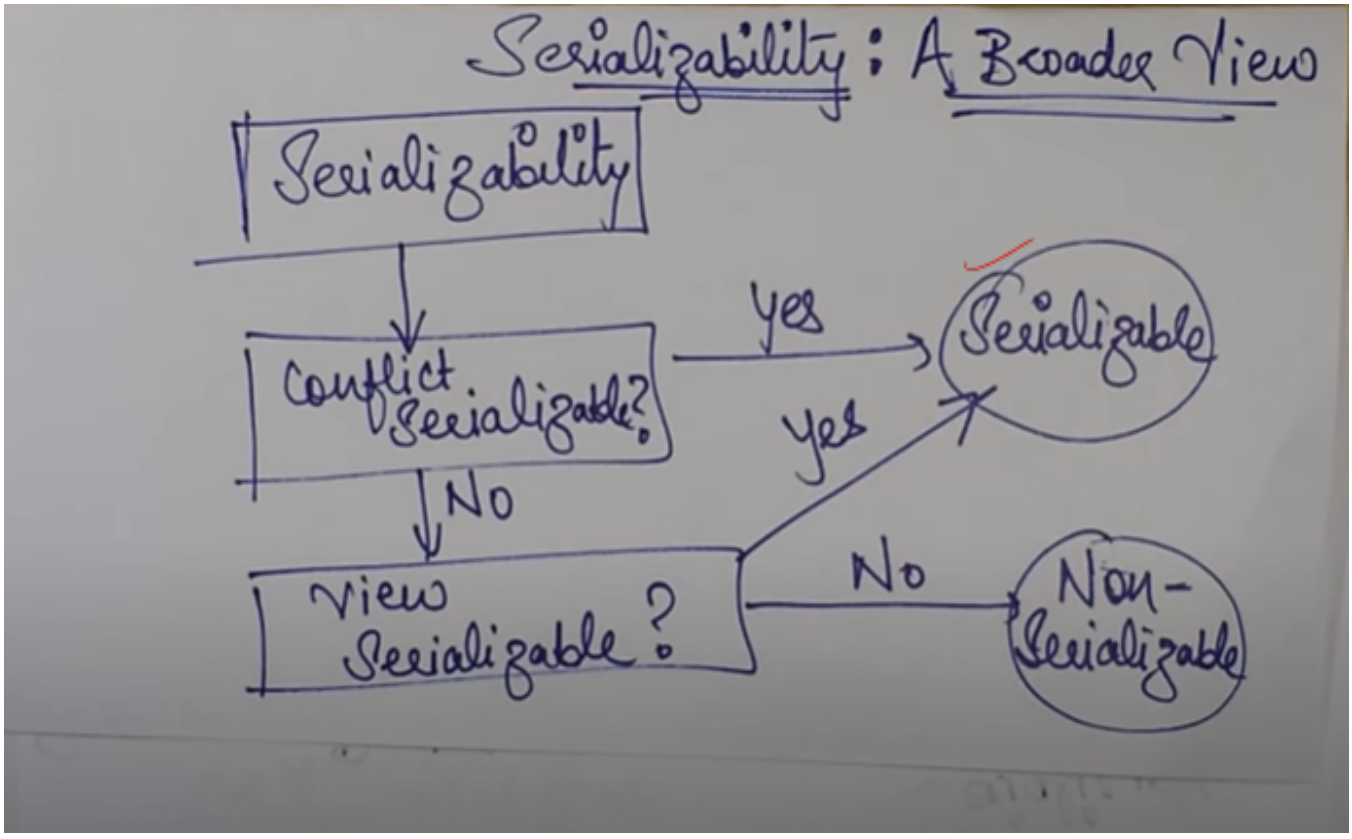
S_2

Without loss of S_1 conflict serializable

S_2 conflict serializable

S_1 : Hint :- serializable

S_2 : not serializable.


Serializability : A Broader View
Serializability
Conflict Serializable?
yes
Serializable
No
yes
View Serializable ?
No
Non-Serializable

View Serializability

- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

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

S1

T1:	R(A),W(A)		
T2:		W(A)	
T3:			W(A)

S2

View Serializability (Cont.)

- 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.

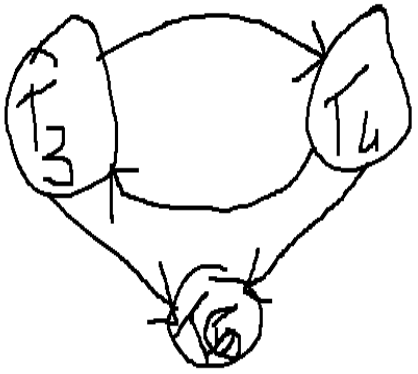
T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

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

: without reading the item, that write takes place

ex1

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



Cylus
not conflict serial

Is it view serializable?

1. Initial read Q
2. Final update T_6
3 writes T_4 T_3 T_6

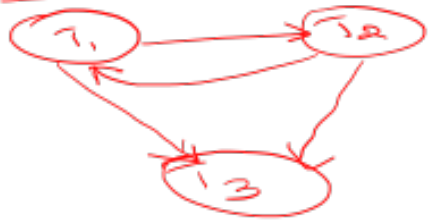
$T_3 \rightarrow T_4 \rightarrow T_6$

Problem :- Can this schedule be converted to a serial schedule? If so provide one.

T_1	T_2	T_3
R(A)		
	W(A)	
W(A)		

Step 1: Test if C.S.S. If yes then serializable if not step 2

Step 2: Find View serial schedule. If not possible then no serial schedule possible



Cycles are present

∴ no C.S.S possible

Step 2 :-	A
Initial Read	T_1
Final update	T_3
writes	T_2 T_1 T_3

$T_1 \Rightarrow T_2 \Rightarrow T_3$ ✓

R_1A W_1A W_2A W_3A

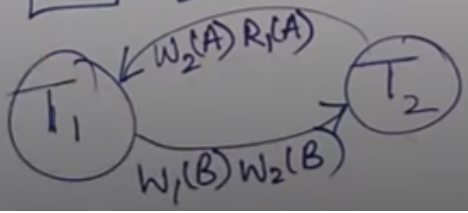
T_1 T_2 T_3

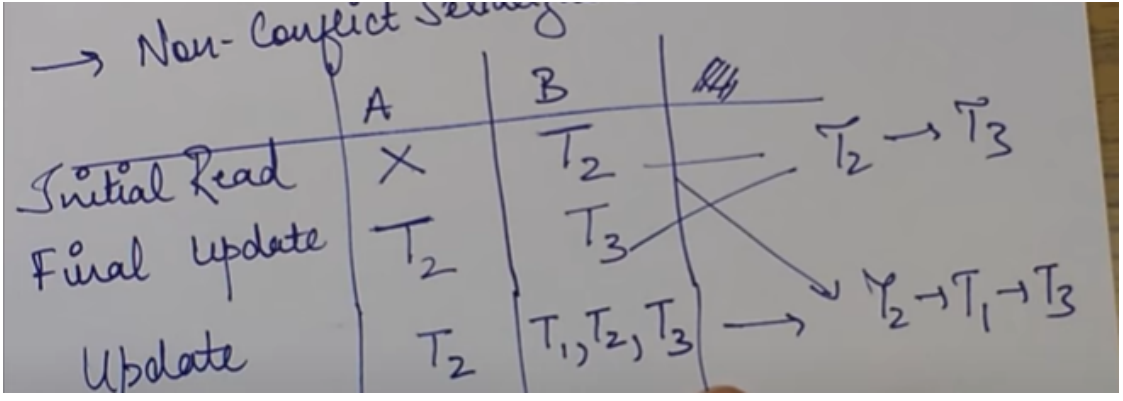
ex2

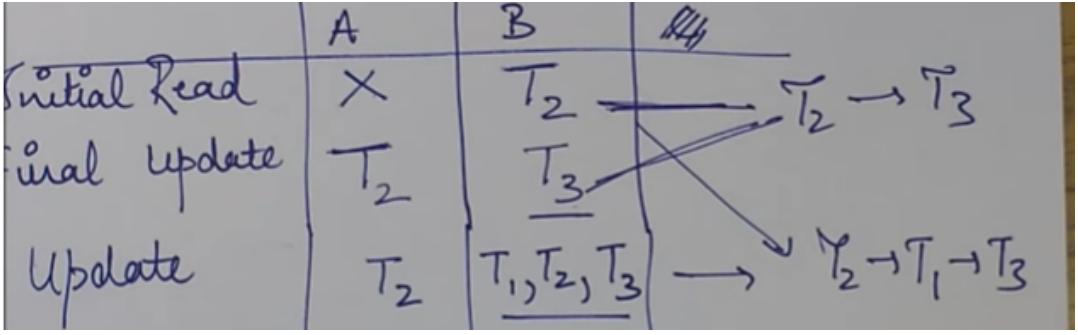
Schedule S, then T_i should only update finally the data item X in the view equivalent Schedule S'.

Example :-

S: $R_2(B)\ W_2(A)\ R_1(A)\ R_3(A)\ W_1(B)\ W_2(B)\ W_3(B)$




→ Non-Conflict
A
B
Initial Read
X
T2
Final Update
T2
T3
Update
T2
T1, T2, T3
T2 → T3
T2 → T1 → T3


A
B
Initial Read
X
T_2
Final Update
T_2
T_3
Update
T_2
T_1, T_2, T_3
$T_2 \rightarrow T_3$
$T_2 \rightarrow T_1 \rightarrow T_3$

Eg2 : for View serializability

T_1 T_2 T_3

R_B

W_A

R_A R_A

W_B W_B

Student work out

Step1. Find C.S.S

Not conflict serializable

Step2:

	A	B
Initial Read	T_2	T_2
Final update	T_2	T_3
write	T_2	T_1 T_2 T_3

Solution: A: T_2

B: $T_2 \rightarrow T_1 \rightarrow T_3$

View serial schedule $T_2 \rightarrow T_1 \rightarrow T_3$

$R_2(B)\, W_2(A)\, W_2(B)\, R_1(A)\, W_1(B)\, R_3(A)\, W_3(B)$

Solve:

Not Conflict serializable because of ~~the~~ cycles seen in precedence cycle

	A	B
Initial Read	T_1 T_2	T_2 T_1
Final update	T_1	T_1
writes	T_2 T_1	T_1

A: $T_2 \rightarrow T_1$
B: $T_2 \rightarrow T_1$
$\therefore$ Final serial schedule

$T_2 \rightarrow T_1$

T_1 T_2 T_3

R_A

R_A

W_A

R_B

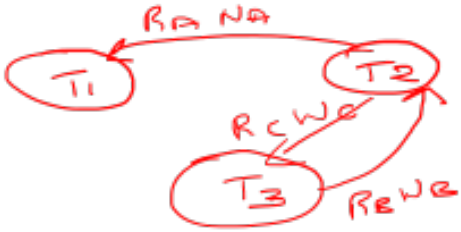
R_C

R_B

W_B

W_C

NOT Conflict serializable?



due to presence of cycle

Step 2:

	A	B	C
Initial Read	T_1 T_2	T_3 T_2	T_2
Final update	T_1	T_2	T_3
writes	T_1	T_2	T_3

A : $T_2 \rightarrow T_1$

B : $T_3 \rightarrow T_2$

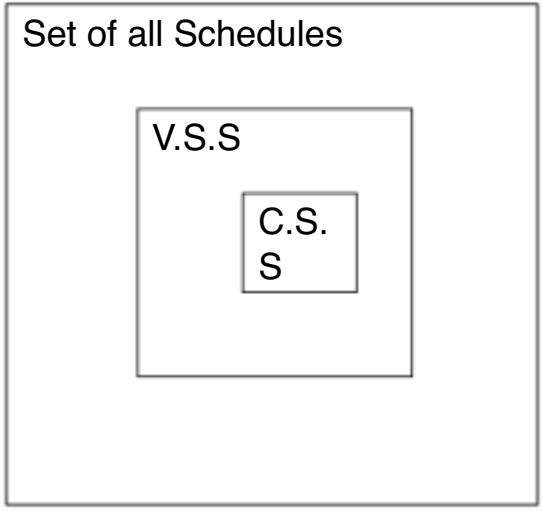
C : $T_2 \rightarrow T_3$

Conflicting ~~serial~~ schedule wrt B & C

∴ Not view serial schedule possible

ex3

Example:



T1 T2

R(A)

R(A)

W(A)

R(B)

W(A)

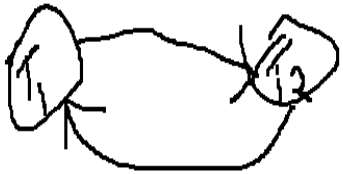
R(B)

W(B)

Check for View serializable with <T1, T2> and <T2, T1>

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

1. Conflict serializable?



No

2. View Serializable

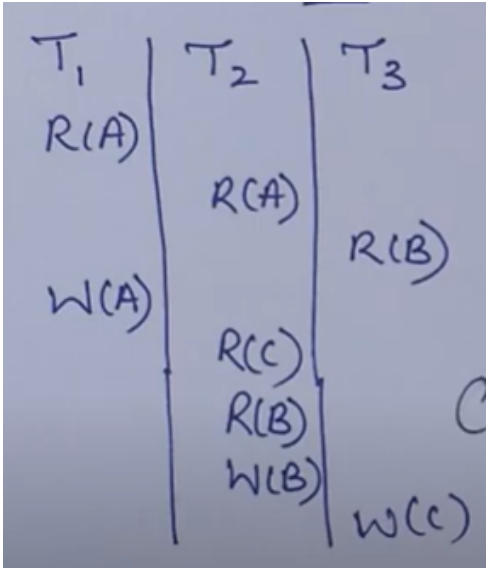
	A	B
Initial read	T_1 T_2	T_2 T_1
Final update write	T_1	T_1
	T_2 T_1	T_1

A: T_2 T_1

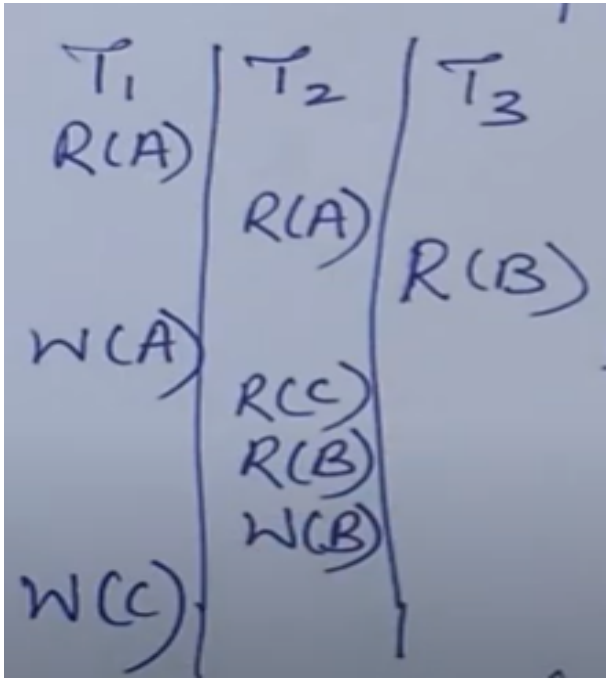
B: T_2 T_1

$\therefore T_2 \rightarrow T_1$

Ex4:no serializable schedule possible



Ex5: Serialzable schedule possible

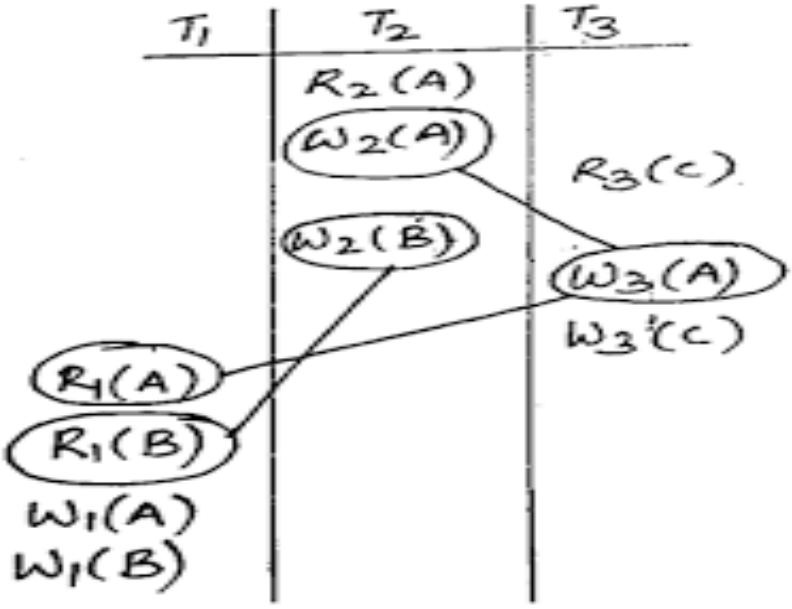


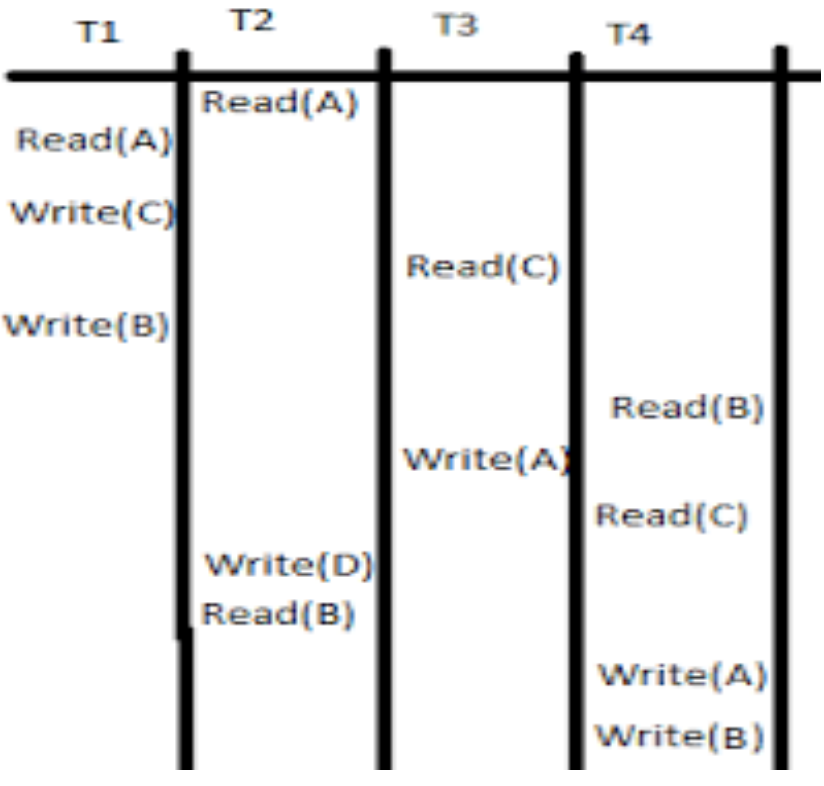
For student workout

- Example: Check for View equivalence

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

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


T1
T2
T3
R2(A)
W2(A)
R3(C)
W2(B)
W3(A)
W3'(C)
R1(A)
R1(B)
W1(A)
W1(B)


T1
T2
T3
T4
Read(A)
Read(A)
Write(C)
Read(C)
Write(B)
Read(B)
Write(A)
Read(C)
Write(D)
Read(B)
Write(A)
Write(B)

Recoverable Schedules

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

Schedule is not recoverable

Recoverable Schedules

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

- **Recoverable schedule** — if a transaction *Tj* reads a data item previously written by a transaction *Ti* , then the commit operation of *Ti* appears before the commit operation of *Tj.*
- The following schedule (Schedule 11) is not recoverable if *T9* 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 $T10$ fails, $T11$ and $T12$ must also be rolled back.
- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

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

- Every cascade-less schedule is also recoverable

- It is desirable to restrict the schedules to those that are cascade-less

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

Concurrency Control (Cont.)

- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonseralizable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

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 (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
  - E.g. Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g. in JDBC, connection.setAutoCommit(false);

End of Chapter 14

Database System Concepts, 6th Ed.

Figure 14.01

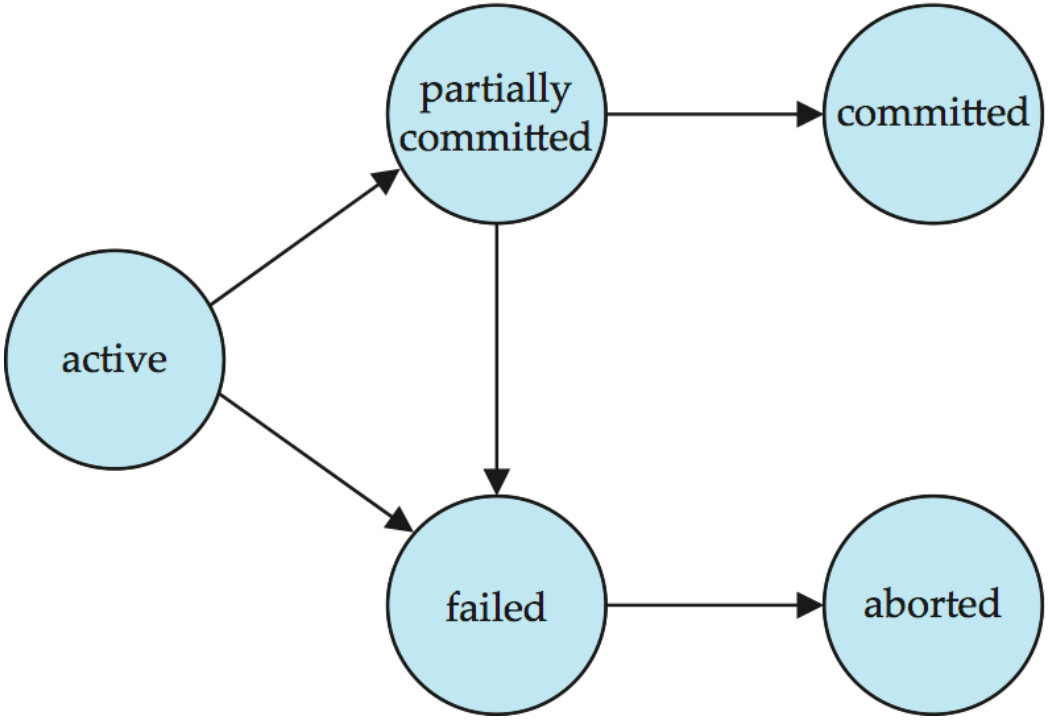


Figure 14.02

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

Figure 14.03

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	

Figure 14.04

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

Figure 14.05

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 14.06

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Figure 14.07

T_1	T_2
read (A)	
write (A)	
	read (A)
read (B)	
	write (A)
write (B)	
	read (B)
	write (B)

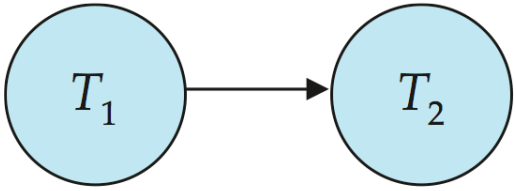
Figure 14.08

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

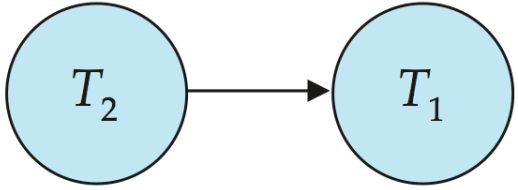
Figure 14.09

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

Figure 14.10



(a)



(b)

Figure 14.11

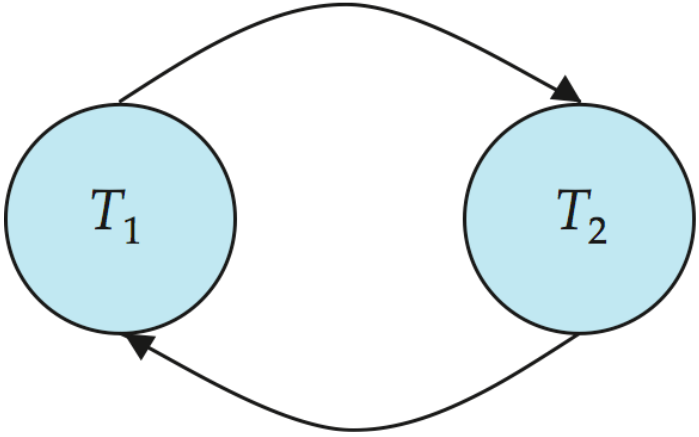
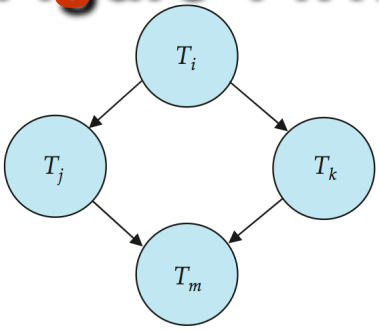
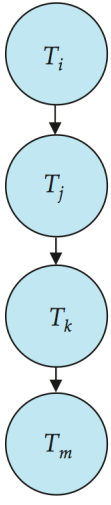


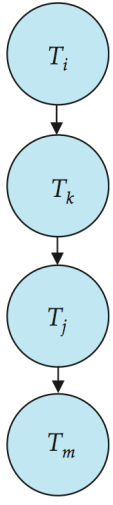
Figure 14.12



(a)



(b)



(c)

Figure 14.13

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)

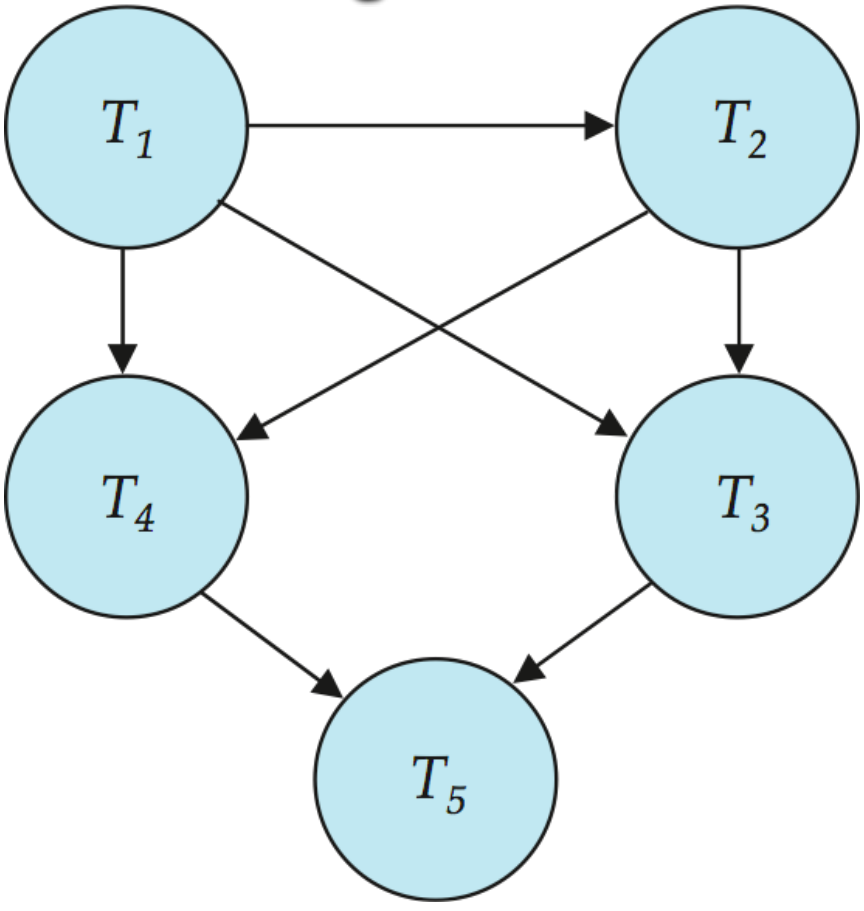
Figure 14.14

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

Figure 14.15

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

Figure 14.16



End of Chapter 14

Database System Concepts, 6th Ed.

Figure 14.01

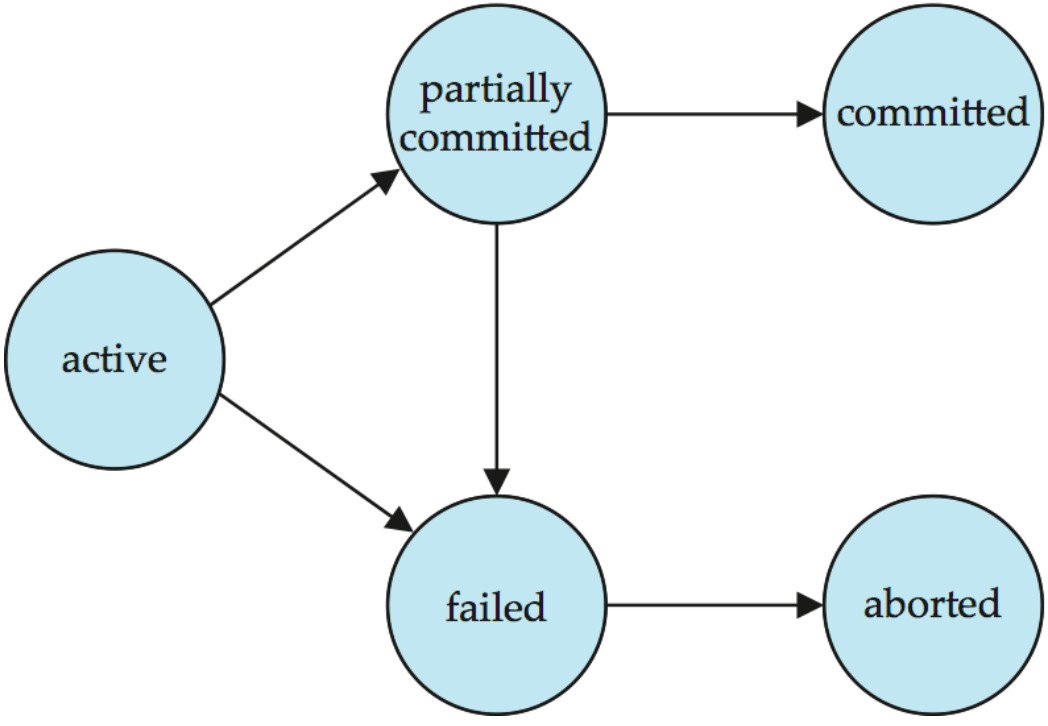


Figure 14.02

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

Figure 14.03

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	

Figure 14.04

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

Figure 14.05

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 14.06

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Figure 14.07

T_1	T_2
read (A)	
write (A)	
	read (A)
read (B)	
	write (A)
write (B)	
	read (B)
	write (B)

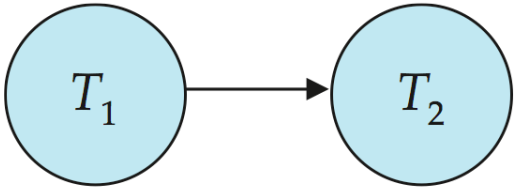
Figure 14.08

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

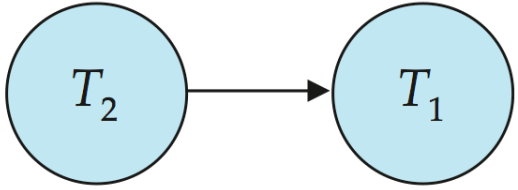
Figure 14.09

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

Figure 14.10



(a)



(b)

Figure 14.11

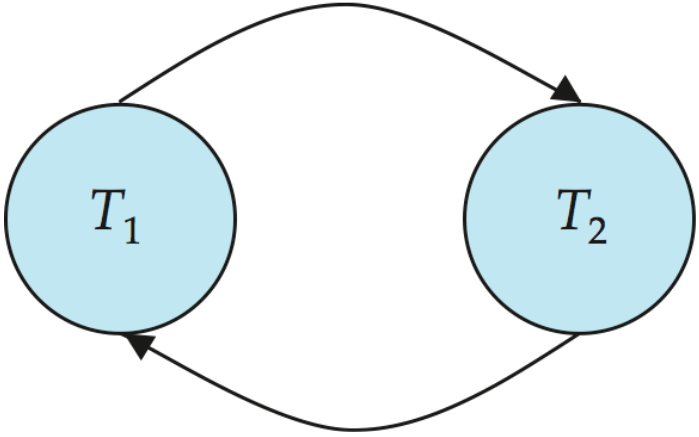
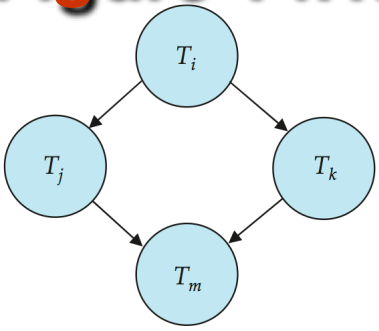
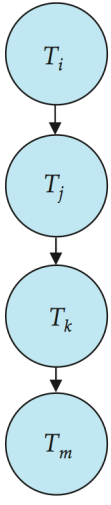


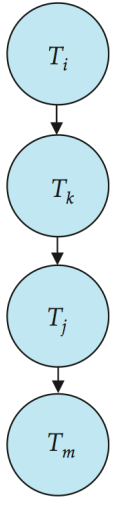
Figure 14.12



(a)



(b)



(c)

Figure 14.13

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)

Figure 14.14

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

Figure 14.15

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

Figure 14.16