Unit - IV

Transaction Management

1. Transaction Concept

- **Transaction:** A **transaction** is a **unit** of program execution that accesses and possibly updates various data items. As an example consider the following transaction that transfers $50 from account A to account B. It can be written as:

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

  This transaction consists of 6 actions that need to takes place in order to transfer $50 from account A to account B.

- **Comparing Transaction and Program:** A transaction is the result from the execution of user program written in high-level data manipulation language (DML) or programming language and it is started and ended between the statements **begin transaction** and **end transaction**.

- **Transaction operations**: Every transaction access data using two operations:
  - **read(X)** : which transfers the data item *X* from the database to a local buffer belonging to the transaction that executed the read operation.
  - **write(X)**: which transfers the data item *X* from the local buffer of the transaction that executed the write back to the database.

2. Properties of the Transaction

- To ensure integrity of the data, we require that the database system maintain the following properties of the transactions:
  - **Atomicity**. Either all operations of the transaction are reflected properly in the database, or none are.
  - **Consistency**. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) 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. Thus, each transaction is unaware of other transactions executing concurrently in the system.
  - **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
- These properties are often called the **ACID properties**.
- To gain a better understanding of ACID properties and the need for them, consider a simplified banking system consisting of several accounts and a set of transactions that access and update those accounts.
- Let *Ti* be a transaction that transfers $50 from account *A* to account *B*. This transaction can be defined as;

T_i : read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Let us now consider each of the ACID requirements.

- **Consistency:** The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction.
- It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

- **Atomicity:** Suppose that, just before the execution of transaction Ti the values of accounts A and B are $1000 and $2000, respectively.
- Now suppose that, during the execution of transaction Ti, a failure occurs (power failures, hardware failures, and software errors) that prevent Ti from completing its execution successfully.
- Further, suppose that the failure happened after the write(A) operation but before the write(B) operation. In this case, the values of accounts A and B reflected in the database are $950 and $2000. The system destroyed $50 as a result of this failure. In particular, we note that the sum $A + B$ is no longer preserved.
- We call such a state as an **inconsistent state**. We must ensure that such inconsistencies are not visible in a database system.
- This state, however, is eventually replaced by the consistent state where the value of account A is $950, and the value of account B is $2050.

- This is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.
- The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write, and, if the transaction does not complete its execution, the database system restores the old values to make it appear as though the transaction never executed.

- **Isolation:** If several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.
- For example consider the following; 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*)	

- A way to avoid the problem of concurrently executing transactions is to execute transactions **serially**—that is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

- **Durability:** The durability property guarantees that, once a transaction completes successfully (i.e., the transfer of the $50 has taken place), all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

3. 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 has been restored to its state prior to the start of the transaction
- **Committed**, after successful completion

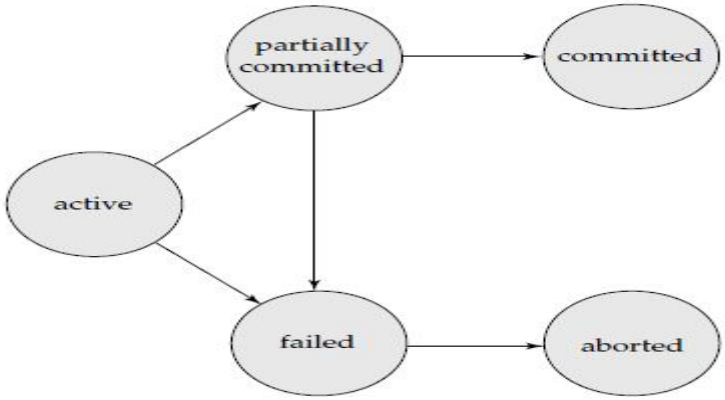


Figure State diagram of a transaction.

- ✓ In the absence of failures, all transactions complete successfully.
- ✓ A transaction may not always complete its execution successfully. Such a transaction is termed **aborted**. An **aborted** transaction must have no effect on the state of the database. Thus, any changes that the aborted transaction made to the database must be undone.
- ✓ Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**.

- ✓ A transaction that completes its execution successfully is said to be **committed**.
- ✓ A transaction enters the **failed state** after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be **rolled back**. Then, it enters the **aborted state**. At this point, the system can either **restart** the transaction or **kill** the transaction.

4. Schedules

- A schedule is a sequences of instructions that specify the chronological order in which instructions of transactions are executed.
- **Serial Schedule:** If transactions are executed from start to finish, one after another then the schedule is called as a serial schedule.
- **Concurrent schedule:** If the instructions of different transactions are interleaved then the schedule is called as a concurrent schedule.
- Let *T*1 and *T*2 be two transactions that transfer funds from one account to another. Transaction *T*1 transfers $50 from account *A* to account *B*. It is defined as;

T1:
read(*A*);
A := *A* − 50;
write(*A*);
read(*B*);
B := *B* + 50;
write(*B*).

- Transaction *T*2 transfers 10 percent of the balance from account *A* to account *B*. It is defined as;

T2:
read(*A*);
temp := *A* * 0.1;
A := *A* − *temp*;

write(A);

read(B);

$B := B + temp$;

write(B).

- The following figure shows a serial schedule in which T1 executes before 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)

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

- Similarly, the following figure shows a serial schedule in which T2 executes before 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)	

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

- Thus, for a set of *n* transactions, there **exist *n*!** different valid serial schedules.
- If two transactions are running **concurrently**, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on. With multiple transactions, the CPU time is shared among all the transactions.
- Concurrent execution of transactions improves the performance of the system. Following figure shows a concurrent 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)

Schedule 3—a concurrent schedule equivalent to schedule 1.

- In case of concurrent execution, several execution sequences are possible, since various instructions from different transactions may now be interleaved. Thus, the number of possible schedules for a set of *n* transactions is much **larger than *n*!.**

- Not all concurrent executions result in a correct state. To illustrate, consider the schedule shown in following Figure. Suppose the current values of accounts A and B are \$1000 and \$2000, respectively. After the execution of this schedule, we arrive at a state where the final values of accounts A and B are \$950 and \$2100, respectively. This final state is an *inconsistent state*, since we have gained \$50 in the process of the concurrent execution and the sum $A + B$ is not preserved by the execution of the two transactions.

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)

Schedule 4—a concurrent schedule.

- We can ensure consistency of the database under concurrent execution by making sure that any schedule that executed has the same effect as a schedule that could have occurred without any concurrent execution. The **concurrency-control component** of the database system carries out this task.
- A transaction that successfully completes its execution will have a commit instruction 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

5. Serializability

- The database system must control concurrent execution of transactions, to ensure that the database state remains consistent.
- For this, we must first understand which schedules will ensure consistency, and which schedules will not. For this, we need to consider simplified view of transactions.
- **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
- The following schedule consist only read and write instructions.

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

Schedule 3—showing only the read and write instructions.

- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
    1. Conflict serializability
    2. View serializability

5.1 Conflict Serializability

- Let us consider a schedule *S* in which there are two consecutive instructions *Ii* and *Ij*, of transactions *Ti* and *Tj* , respectively ($i \neq j$). If *Ii* and *Ij* refer to different data items, then we can swap *Ii* and *Ij* without affecting the results of any instruction in the schedule. If *Ii* and *Ij* refer to the same data item *Q*, then the order of the two steps may matter.
- Instructions *li* and *lj* of transactions *Ti* and *Tj* respectively, **conflict** if and only if there exists some item *Q* accessed by both *li* and *lj*, and at least one of these instructions wrote *Q*.
    1. *li* = **read**(*Q*), *lj* = **read**(*Q*). *li* and *lj* don't conflict.
    2. *li* = **read**(*Q*), *lj* = **write**(*Q*). They conflict.
    3. *li* = **write**(*Q*), *lj* = **read**(*Q*). They conflict
    4. *li* = **write**(*Q*), *lj* = **write**(*Q*). They conflict
- We say that *Ii* and *Ij* **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.
- If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of nonconflicting 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.
- To illustrate the concept of conflicting instructions, consider schedule 3, shown in the above figure. By performing a series of swaps of

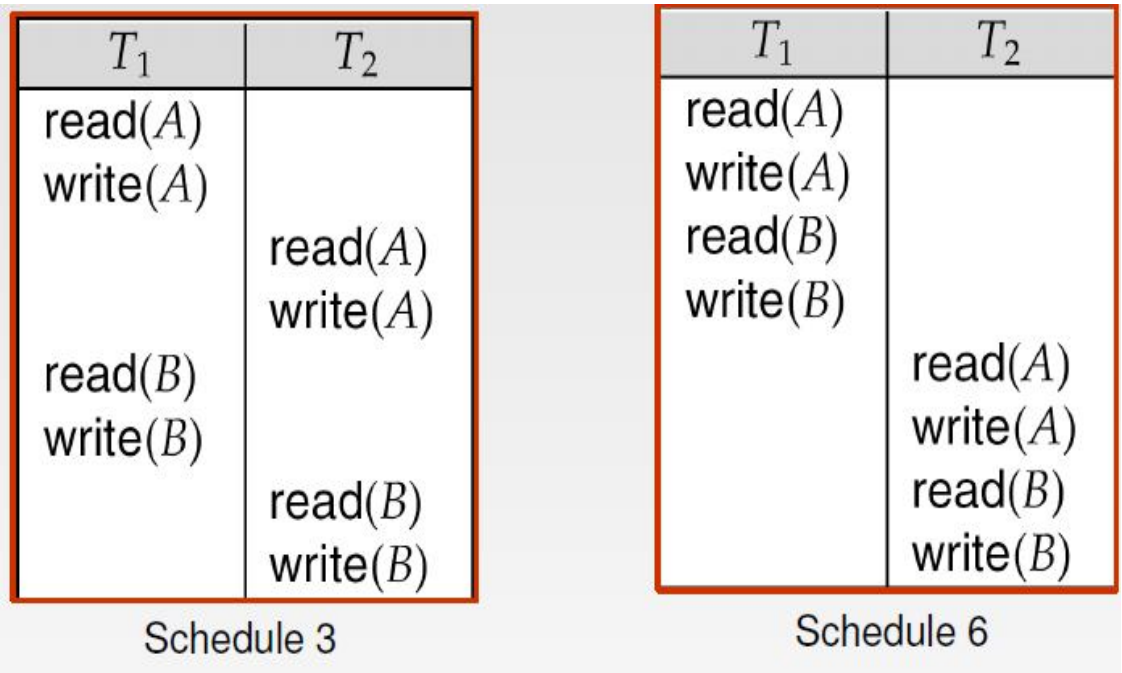
nonconflicting instructions, schedue 3 can be transformed to a serial schedule as follows;

- Swap read(B) of T1 with write(A) of T2. The result of this is shown beow;

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

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

- Similarly, by performing the following series of swaps of nonconflicting instructions, schedule 3 is transformed to a serial schedule as shown below;
    - Swap read(*B*) of *T*1 with read(*A*) of *T*2.
    - Swap write(*B*) of *T*1 with write(*A*) of *T*2.
    - Swap write(*B*) of *T*1 with read(*A*) of *T*2.



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

Schedule 3

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

Schedule 6

- The final result of these swaps, schedule 6 shown above, is a serial schedule. Thus, we have shown that schedule 3 is equivalent to a serial schedule and therefore schedule 3 is a conflict serializable schedule..
- Consider the schedule 7 of the following figure; it consists of only the read and write operations of transactions $T3$ and $T4$. This schedule is not conflict serializable, since it is not equivalent to either the serial schedule <$T3$,$T4$> or the serial schedule <$T4$,$T3$>.

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

Schedule 7

5.2 View Serializability

- Let *S* and *S*[1] be two schedules with the same set of transactions. The schedules *S* and *S*[1] are said to be **view equivalent** if three conditions are met, for each data item Q:
  1. If in schedule S, transaction *Ti* reads the initial value of *Q*, then in schedule *S'* also transaction *Ti* must read the initial value of *Q*.
  2. If in schedule S transaction *Ti* executes **read**(*Q*), and that value was produced by transaction *Tj* (if any), then in schedule *S'* also transaction *Ti* must read the value of *Q* that was produced by the same **write**(Q) operation of transaction *Tj* .
  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'*.

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

- 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 viewserializable but *not* conflict serializable.

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

Schedule 9 – A view serializable schedule

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

Testing for Serializability

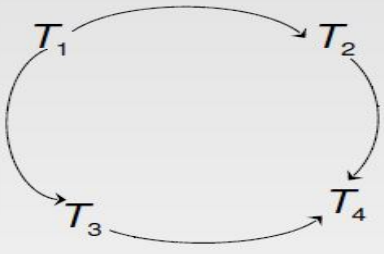
- Consider some schedule of a set of transactions *T*1, *T*2, ..., *Tn*
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from *Ti* to *Tj* if the two transaction conflict.
- A schedule is conflict serializable iff its precedence graph is acyclic.
- Examples;

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)

(a) (b)

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)	

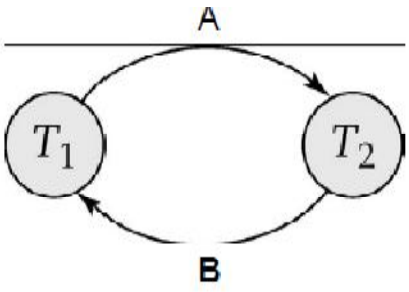
T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Schedule (c)

- The schedules shown above are conflict serializable schedules since, their precedence graphs are acyclic.
- The following schedule is not conflict serializable as its precedence graph consists a cycle.

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)



The schedule cannot be conflict equivalent to a serial schedule

6. Recoverability

- If a transaction *Ti* fails, we need to undo the effect of this transaction to ensure the atomicity property of the transaction.
- In the following sections, we address the issue of what schedules are acceptable from the viewpoint of recovery from transaction failure.

6.1 Recoverable Schedules

- A **recoverable schedule** is one where, 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 commit operation of *Tj* .

- The following schedule (schedule 11) is not a recoverable schedule if T9 commits immediately after the read(A) operation.

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

Schedule 11 – non-recoverable schedule

- 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.

6.2 Cascadeless Schedules

- **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)

Schedule 12

If *T*10 fails, *T*11 and *T*12 must also be rolled back.

- Cascading rollback lead to the undoing of a significant amount of work.
- A **cascadeless schedule** is one where, 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 cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless.

UNIT-IV

Assignment-Cum-Tutorial Questions

SECTION-A

Objective Questions

1. Identify the characteristics of transactions [ ]

A) Atomicity B) Durability C) Isolation D) All of the mentioned

2. Which of the following has "all-or-none" property? [ ]

A) Atomicity B)Durability C) Isolation D) All of the mentioned

3. Which one of the following is NOT a part of the ACID properties of database transactions? [ ]

A. Atomicity B) Isolation C) Consistency D) Deadlock-freedom

4. 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: [ ]

A) Atomicity B) Durability C) Isolation D) All of the mentioned

5. The property of transaction that persists all the crashes is [ ]

A) Atomicity B)Durability C) Isolation D) All of the mentioned

6. Consider the following transaction involving two bank accounts x and y. read(x); x := x – 50; write(x); read(y); y := y + 50; write(y); The constraint that the sum of the accounts x and y should remain constant is known as:

A. Atomicity B) Isolation C) Consistency D) Durability[ ]

7. Precedence graphs help to find a [ ]

A) Serializable schedule C) Recoverable schedule

B) Deadlock free schedule D) Cascadeless schedule

8. Consider the following four schedules due to three transactions(indicated by the subscript) using read and write on a data item x, denoted by r(x) and w(x) respectively. Which one of them is conflict serializable? [ ]

(A) $r_1(x)$; $r_2(x)$; $w_1(x)$; $r_3(x)$; $w_2(x)$

(B) $r_2(x)$; $r_1(x)$; $w_2(x)$; $r_3(x)$; $w_1(x)$

(C) $r_3(x)$; $r_2(x)$; $r_1(x)$; $w_2(x)$; $w_1(x)$

(D) $r_2(x)$; $w_2(x)$; $r_3(x)$; $r_1(x)$; $w_1(x)$

9. Consider the transactions T1, T2, and T3 and the schedules S1 and S2 given below. [ ]

T1: r1(X); r1(Z); w1(X); w1(Z)

T2: r2(Y); r2(Z); w2(Z)

T3: r3(Y); r3(X); w3(Y)

S1: r1(X); r3(Y); r3(X); r2(Y); r2(Z); w3(Y); w2(Z); r1(Z); w1(X); w1(Z)

S2: r1(X); r3(Y); r2(Y); r3(X); r1(Z); r2(Z); w3(Y); w1(X); w2(Z); w1(Z)

Which one of the following statements about the schedules is TRUE?

A. Only S1 is conflict-serializable.

B. Only S2 is conflict-serializable.

C. Both S1 and S2 are conflict-serializable.

D. Neither S1 nor S2 is conflict-serializable.

10. Consider the following two phase locking protocol. Suppose a transaction T accesses (for read or write operations), a certain set of objects {O1,...,Ok}. This is done in the following manner:

Step 1. T acquires exclusive locks to O1, . . . , Ok in increasing order of their addresses. Step 2. The required operations are performed. Step 3. All locks are released. This protocol will; [ ]

A. guarantee serializability and deadlock-freedom

B. guarantee neither serializability nor deadlock-freedom

C. guarantee serializability but not deadlock-freedom

D. guarantee deadlock-freedom but not serializability

11. Which of the following property state that the data used during the execution of a transaction cannot be used by a second transaction until the first one is completed. [ ]

A) Consistency B) Atomicity C) Durability D) Isolation

12. Which property states that only valid data will be written to the database?

A. Consistency B) Durability C) Atomicity D) Isolation

SECTION-B

SUBJECTIVE QUESTIONS

1) Define Transaction and briefly explain ACID properties.
2) Draw transaction state diagram and describe each state that a transaction goes through during its execution.
3) What is schedule? Explain different types of schedules.
4) How can you test whether a given schedule is conflict-serializable? Is every conflict-serializable schedule is serializable? Justify.
5) Construct a precedence graph for serial schedule and non serial schedule.
6) Create a concurrent schedule for executing the following transactions;

   T1: transfer funds $1000 from account A to account B

   T2: Increase the balance amount of account A to 10%
7) Consider the following two transactions:

   *T*1: read(*A*);

   read(*B*);

   if *A* = 0**then** *B* := *B* + 1;

   write(*B*).

   *T*2: read(*B*);

   read(*A*);

   if *B* = 0**then** *A* := *A* + 1;

   write(*A*).

   Let the consistency requirement be $A = 0 \vee B = 0$, with $A = B = 0$ the initial values.

a. Show that every serial execution involving these two transactions preserves the consistency of the database.
b. Show a concurrent execution of *T*1 and *T*2 (shown in the above problem) that produces a nonserializable schedule.

8) Is there a concurrent execution of *T*1 and *T*2 (shown in the above problem) that produces a serializable schedule?

9) Test whether the following schedule is conflict serializable (Subscripts denote transactions)?

S1: R1(X); R2(X); W1(X); R3(X); W2(X);

10) Consider the following three schedules due to three transactions (indicated by the subscript) using read and write on a data item X, denoted by R(X) and W(X) respectively. Construct the precedence graph for each schedule and determine which of them is conflict serializable?

S1: R2(X); R1(X); W1(X); R3(X); W2(X);

S2: R3(X); R2(X); R1(X); W2(X); W1(X);

S3: R2(X); W2(X); R3(X); R1(X); W1(X);

11) Consider three transactions: T1, T2 and T3. Draw the precedence graph for the following schedule consisting of these three transactions and determine whether it is serializable. If so, give its serial order(s).

T1	T2	T3
		read(Y)
		read(Z)
read(X)		
write(X)		
		write(Y)
		write(Z)
	read(Z)	
read(Y)		
write(Y)		

	read(Y) write(Y) read(X) write(X)	

SECTION-C
QUESTIONS AT THE LEVEL OF GATE

1. Consider the following schedules involving two transactions. Which one of the following statements is TRUE? (GATE 2007)

   S_1: $r_1(X)$; $r_1(Y)$; $r_2(X)$; $r_2(Y)$; $w_2(Y)$; $w_1(X)$

   S_2: $r_1(X)$; $r_2(X)$; $r_2(Y)$; $w_2(Y)$; $r_1(Y)$; $w_1(X)$

   A) Both S_1 and S_2 are conflict serializable

   B) S_1 is conflict serializable and S_2 is not conflict serializable

   C) S_1 is not conflict serializable and S_2 is conflict serializable

   D) Both S_1 and S_2 are not conflict serializable

2. Consider the following schedule **S** of transactions T1, T2, T3, T4:

T1	T2	T3	T4
	Reads(X)		
		Writes(X)	
		Commit	
Writes(X)			
Commit			
	Writes(Y)		
	Reads(Z)		
	Commit		

			Reads(X) Reads(Y) Commit

Which one of the following statements is CORRECT? (GATE 2014)

A) **S** is conflict-serializable but not recoverable

B) **S** is not conflict-serializable but is recoverable

C) **S** is both conflict-serializable and recoverable

D) **S** is neither conflict-serializable nor is it recoverable

3. Consider the following transactions with data items P and Q initialized to zero: (GATE 2012)

T1: read (P) ;
read (Q) ;
if P = 0 then Q : = Q + 1 ;
write (Q).

T2: read (Q) ;
read (P)
if Q = 0 then P : = P + 1 ;
write (P).

Any non-serial interleaving of T1 and T2 for concurrent execution leads to

A) a serializable schedule

B) a schedule that is not conflict serializable

C) a conflict serializable schedule

D) a schedule for which precedence graph cannot be drawn

4. Which of the following scenarios may lead to an irrecoverable error in a database system? (GATE 2003)
   A) A transaction writes a data item after it is read by an uncommitted transaction
   B) A transaction reads a data item after it is read by an uncommitted transaction
   C) A transaction reads a data item after it is written by a committed transaction
   D) A transaction reads a data item after it is written by an uncommitted transaction

4. Consider the data items D1, D2 and D3, and the following execution schedule of transactions T1, T2, and T3. In the diagram, R(D) and W(D) denote the actions reading and writing the data item D respectively.

T1	T2	T3
	R(D3)	
	R(D2)	
	W(D2)	
		R(D2)
		R(D3)
R(D1)		
W(D1)		
		W(D2)
		W(D3)
	R(D1)	
R(D2)		

W(D2)	W(D1)	

Which of the following statements is correct?

A) The schedule is searializable as T2, T3, T1

B) The schedule is searializable as T2, T1, T3

C) The schedule is searializable as T3, T2, T1

D) The schedule is not searializable

****