Q1. How to ensure the atomicity using Recovery Methods? Explain the log based recovery method in detail.

Ans:- **How to Ensure Atomicity Using Recovery Methods?**

Atomicity means *a transaction must be executed completely or not executed at all*.
If a failure occurs (system crash, power failure, server crash, etc.), the DBMS must ensure that:

- **All completed transactions remain permanent**, and
- **All incomplete or failed transactions must be undone**.

To ensure this, **DBMS uses Recovery Methods**, mainly:

1. **Log-Based Recovery**
2. **Checkpointing**
3. **Shadow Paging**

Among these, **log-based recovery** is the most widely used method to ensure atomicity.

✅ **Log-Based Recovery Method (In Detail)**

What is a Log?

A **log** is a file stored on stable storage (like disk) that records all operations performed by transactions.

Every log entry contains:

<Transaction-ID, Data-item, Old-value, New-value>

This helps the DBMS **undo** or **redo** transactions during recovery.

🔶 **Types of Log Records**

Important log records are:

1. **<T_i start>**
   → Transaction T_i has started.
2. **<T_i, X, old_value, new_value>**
   → T_i updated data item X.
3. **<T_i commit>**
   → T_i has completed successfully (changes must be REDO).
4. **<T_i abort>**
   → T_i has failed (changes must be UNDONE).

🔶 **Two Main Operations in Log-Based Recovery**

1. Undo (Rollback)

Applied when a transaction **did not commit** before failure.
DBMS restores the *old values* using log entries.

Example:

<T2, A, 50, 80>

Undo means: set A = 50

---

2. Redo

Applied when a transaction **committed** before failure.
DBMS re-applies the *new values*.

Example:

<T1, B, 10, 20>

Redo means: set B = 20

🔶 **When to use Undo or Redo?**

Transaction Status	Action During Recovery
Committed before crash	**REDO**
Not committed before crash	**UNDO**

This rule ensures **atomicity**.

🎯 **Example of Log-Based Recovery**

Assume the log contains:

<T1 Start>

<T1, A, 100, 150>

<T1, B, 200, 300>

<T1 Commit>

<T2 Start>

<T2, C, 50, 80>

--- system crashes ---

After crash:

- **T1 committed → REDO T1**
- **T2 did not commit → UNDO T2**

This ensures atomicity:

- T1 is fully executed.
- T2 is completely removed.

🔶 **Types of Log-Based Recovery**

1. Deferred Database Modification

- Updates are **not written** to the database until **transaction commits**.
- Only **redo** is needed.
- Log contains only *new values*.

2. Immediate Database Modification

- Updates are written to the database **before commit**.
- Both **undo** and **redo** operations are required.
- Log contains both *old* and *new values*.

🔶 **Checkpoints (for faster recovery)**

A checkpoint is a point where the DBMS writes all in-memory data to disk and logs:

<checkpoint>

During recovery, system scans the log **from last checkpoint only**, not from beginning.

🎯 **Diagram: Log-Based Recovery**

(Textual form for exam)

```
+----------------------+
|      LOG FILE        |
+----------------------+
| <T1 start>           |
| <T1, X, 10, 20>      |
| <T1 commit>          |
| <T2 start>           |
| <T2, Y, 5, 15>       |
```

| <system crash> |

+---------------------+

Recovery Manager:

1. REDO T1 (committed)

2. UNDO T2 (not committed)

Q2. What is need of lock in DBMS? Explain shared lock and exclusive lock with the help of example.

Ans:- **What is the Need of Lock in DBMS?**

A **lock** is a concurrency control mechanism used in DBMS to manage simultaneous access to database data by multiple transactions.

🔹 **Why locks are needed?**

1. **To maintain data consistency**
   Prevents different transactions from accessing and modifying the same data simultaneously.
2. **To avoid anomalies** such as:
   - Dirty Read
   - Lost Update
   - Uncommitted Read
   - Inconsistent Read
3. **To ensure isolation property of ACID**
   Locks help ensure that each transaction runs as if it is alone in the system.
4. **To prevent conflicts**
   When two transactions try to read/write the same data item, locks ensure proper ordering.
5. **To allow controlled concurrency**
   Locks allow multiple transactions to run in parallel safely.

✅ **Types of Locks in DBMS**

DBMS uses mainly **two types** of locks:

1. **Shared Lock (S-Lock)**
2. **Exclusive Lock (X-Lock)**

These are used in **Two-Phase Locking (2PL)** and other concurrency control protocols.

🔶 **1. Shared Lock (S-Lock)**

✓ **Meaning:**

- A transaction uses a **shared lock** when it wants to **read** a data item.
- **Multiple transactions can hold shared locks on the same data item simultaneously**.

✓ **But:**

- A shared lock **prevents any transaction from writing** to the data item.

📘 **Example:**

Let the data item be **A**.

T1: wants to read A → Acquires Shared Lock (S-lock) on A

T2: also wants to read A → Allowed (shared lock compatible)

Both T1 and T2 can **read** A at the same time.

But:

T3: wants to write A → NOT allowed (exclusive lock needed)

So shared lock **allows multiple readers** but **blocks writers**.

🔶 **2. Exclusive Lock (X-Lock)**

✓ **Meaning:**

- A transaction uses an **exclusive lock** when it wants to **write/update** a data item.
- **Only one transaction can hold an exclusive lock on a data item.**
- It prevents **both read and write** access by other transactions.

📘 **Example:**

Let the data item be **A**.

T1: wants to write A → Acquires Exclusive Lock (X-lock) on A

Now:

T2: wants to read A → NOT allowed

T3: wants to write A → NOT allowed

No other transaction can access A until T1 releases the exclusive lock.

🔶 **Compatibility Table (Very important for exam)**

Lock Requested	Shared Lock Present	Exclusive Lock Present
Shared Lock	✓ Allowed	✗ Not Allowed

Lock Requested	Shared Lock Present	Exclusive Lock Present
Exclusive Lock	✗ Not Allowed	✗ Not Allowed

🎯 **Simple Real-Life Analogy**

- **Shared Lock** = Library book reading
  Many students can **read** a book at the same time.
- **Exclusive Lock** = Taking the book home
  If one student takes it, **no one else can read or borrow** it until returned.

Q3. When do deadlocks happen, how to prevent them, and how to recover if deadlock takes place?

Ans:- **1. When Do Deadlocks Happen?**

A **deadlock** occurs when two or more transactions are **waiting for each other** in a circular chain, and **none of them can proceed**.

Deadlock occurs when all 4 conditions hold:

1. **Mutual Exclusion:**
   Only one transaction can use a resource at a time.
2. **Hold and Wait:**
   A transaction holds one resource and waits for another.
3. **No Preemption:**
   Resources cannot be forcefully taken away; a transaction must release them voluntarily.
4. **Circular Wait:**
   A cycle of waiting exists.
   Example:
5. T1 waits for a resource held by T2,
6. T2 waits for a resource held by T3,
7. T3 waits for a resource held by T1.

Simple Example:

T1 holds lock on A, needs lock on B

T2 holds lock on B, needs lock on A

→ Both wait forever → Deadlock

✅ **2. How to Prevent Deadlocks?**

Deadlock prevention ensures the system **never enters** a deadlock state by **breaking at least one of the 4 conditions**.

🔹 **A. Resource Ordering Method**

- Order all data items globally (A < B < C ...)
- A transaction must lock items **in increasing order only**.
- Prevents **circular wait**.

Example:

Both T1 and T2 must lock A before B → No cycle

🔹 **B. Timeout Method**

- If a transaction waits too long, DBMS automatically **aborts it**.
- Simple but may cause unnecessary aborts.

🔹 **C. Deadlock Prevention Using Timestamp Protocol**

Two popular techniques:

1) Wait-Die Scheme (Non-Preemptive)

- If older transaction requests a lock held by younger → **It waits**
- If younger requests a lock held by older → **It dies (is aborted)**

2) Wound-Wait Scheme (Preemptive)

- If older transaction requests a lock held by younger → **Younger is rolled back** (wounded)
- If younger requests lock held by older → **Younger waits**

These avoid cycles → no deadlock.

✅ **3. How to Detect Deadlocks?**

Used when deadlock prevention is *not* used.
System allows deadlocks, but detects them using:

Wait-For Graph (WFG)

- Nodes = Transactions
- Edge Ti → Tj means **Ti is waiting for a resource held by Tj**.

If the graph contains a cycle → Deadlock exists.

✅ **4. How to Recover from a Deadlock?**

When deadlock is detected, DBMS must **break the cycle**.

🔶 **A. Abort (Rollback) a Transaction**

Kill one or more transactions involved in deadlock.

Criteria to decide which transaction to abort:

- Smallest amount of work done
- Youngest transaction
- Fewest resources held
- Lowest priority

🔶 **B. Resource Preemption**

Forcefully take a resource from some transactions and give it to others.

Steps:

1. Select a victim transaction
2. Rollback the victim
3. Release its locks
4. Restart it later

Q4. What is R-timestamp(Q) and W-timestamp(Q). Explain the necessary condition used by time stamp ordering protocol to execute for a read/write operation.

Ans:- **What is R-timestamp(Q) ?**

R-timestamp(Q) = The **largest (latest)** timestamp of any transaction **that successfully read data item Q**.

- It records **who read Q most recently**.
- Helps the system prevent *out-of-order* reads.

Example:

If

- T2 (TS = 20) reads Q
- T5 (TS = 50) reads Q later

Then:

R-timestamp(Q) = 50

✅ **What is W-timestamp(Q)?**

W-timestamp(Q) = The **largest (latest)** timestamp of any transaction **that successfully wrote data item Q**.

- It records **who last wrote Q**.
- Used to prevent *out-of-order* writes.

Example:

If

- T3 (TS = 30) writes Q
- T7 (TS = 70) writes Q later

Then:

W-timestamp(Q) = 70

⭐ **Timestamp Ordering Protocol (TO Protocol)**

This protocol ensures **conflict-serializability** by always executing operations **in the order of timestamps**.

If a transaction attempts to perform an operation that **violates timestamp order**, the transaction is **aborted and restarted** with a new timestamp.

✅ **Necessary Conditions for Read and Write Operations**

The timestamp of a transaction requesting read/write is denoted as **TS(T)**.

Let the data item be **Q**.

🔶 **1. Condition for READ(Q)**

Transaction **T** wants to read **Q**.

✓ **Allowed if:**

TS(T) ≥ W-timestamp(Q)

Meaning:
T is **not older** than the transaction that last wrote Q.

✗ **If TS(T) < W-timestamp(Q):**

- A **younger write** already happened.
- Reading old value → **violates timestamp order**.

➞ **Transaction T is aborted and restarted.**

After successful read:

R-timestamp(Q) = max(R-timestamp(Q), TS(T))

🔶 2. Condition for WRITE(Q)

Transaction **T** wants to write **Q**.

Two tests required:

Condition 1:

TS(T) ≥ R-timestamp(Q)

If TS(T) < R-timestamp(Q):

- A **younger transaction already read Q**
  ➞ T is too old → **abort T** (to prevent overwriting a new read)

Condition 2:

TS(T) ≥ W-timestamp(Q)

If TS(T) < W-timestamp(Q):

- A **younger write** already happened
  ➞ T is too late → **abort T**

✓ If both conditions hold:

- Write is allowed
- After write:

W-timestamp(Q) = TS(T)

⭐ Summary Table

Operation	Condition	Meaning	If Violated
Read(Q)	TS(T) ≥ W-timestamp(Q)	T is not older than last writer	Abort T
Write(Q) (1)	TS(T) ≥ R-timestamp(Q)	No younger reader	Abort T
Write(Q) (2)	TS(T) ≥ W-timestamp(Q)	No younger writer	Abort T

Q5. Explain the concept of conflict serializability with suitable example. Since every conflict - serializable schedule is view serializable, why do we emphasize conflict serializability rather than view serializability?

Ans:- **Concept of Conflict Serializability**

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

✓ **Two operations conflict when:**

1. They belong to **different transactions**,
2. They operate on the **same data item**, and
3. **At least one is a write**.

✓ **Types of conflicting pairs:**

- **Read–Write (RW)**
- **Write–Read (WR)**
- **Write–Write (WW)**

If we can swap non-conflicting operations step-by-step to transform a schedule into a serial schedule, then the schedule is **conflict serializable**.

⭐ **Example of Conflict Serializability**

Consider schedule **S**:

T1: R(A) W(A)

T2: R(A) W(A)

Written as a sequence:

1. T1: R(A)

2. T2: R(A)

3. T1: W(A)

4. T2: W(A)

✓ **Find conflicts:**

- (T1:R(A), T2:W(A)) → conflict
- (T1:W(A), T2:R(A)) → conflict
- (T1:W(A), T2:W(A)) → conflict

Construct a **precedence graph**:

- T1 → T2 (because T1:W(A) happens before T2:R(A) / T2:W(A))
- T2 → T1 (because T2:R(A) happens before T1:W(A))

We have a **cycle** → **Not conflict serializable**.

✓ **Another Example (Conflict Serializable)**

Schedule S2:

1. T1: R(A)

2. T1: W(A)

3. T2: R(A)

4. T2: W(A)

Conflicts: only T1 → T2
No cycle in precedence graph → **Conflict Serializable**

Equivalent serial schedule:

T1 → T2

⭐ **Why Do We Emphasize Conflict Serializability Instead of View Serializability?**

Even though:

👉 **Every conflict-serializable schedule is view-serializable**,
but the **reverse is not true**.

So, why is conflict serializability preferred?

🔶 **1. Conflict Serializability is Easy to Test**

- We can check conflict serializability using a **precedence graph**.
- Graph **acyclic → conflict serializable**.
- The test is **efficient (O(n))**.

View serializability test:

- Extremely **complex**
- Requires checking many possible equivalent serial schedules
- Computationally **NP-complete** (very slow)

Hence, for practical concurrency control, conflict serializability is used.

🔶 **2. Conflict Serializability is Enforced by Locking Protocols**

Most DBMS concurrency control systems (like **Two-Phase Locking – 2PL**) automatically guarantee **conflict serializability**.

Lock-based protocols are **not designed** to ensure view serializability.

🔶 3. Conflict Serializable Schedules Avoid "Dirty Reads" More Naturally

View-serializable schedules may sometimes include:

- Reads on uncommitted writes (dirty reads)
- Blind writes (writing without reading first)

These can be unsafe in DBMS.

Conflict serializability disallows such unsafe operations.

🔶 4. View Serializability Allows More Schedules but Is Harder to Implement

Though view serializability is more flexible:

- DBMS systems avoid it because **checking safety is expensive**
- Enforcement is complicated
- Not required for real-world systems where speed matters

Thus, **conflict serializability** gives a **good balance between safety and practicality**.

⭐ Final Summary

✓ Conflict Serializability:

- Based on **swapping non-conflicting operations**
- Easy to check using **precedence graph**
- Guaranteed by standard protocols like 2PL

✓ View Serializability:

- More general but **much harder to test**
- Computationally costly
- Rarely used in practical DBMS

✓ Why emphasize conflict serializability?

- **Easier to test**
- **Efficient and practical**
- **Supported naturally by lock-based protocols**
- Ensures correctness without expensive computation

Q6. Explain the two-phase lock protocol for concurrency control. Also explain its two versions: strict two-phase lock protocol and rigorous two-phase lock protocol.

Ans:- **Two-Phase Locking (2PL) Protocol**

Two-Phase Locking is a concurrency control protocol used in DBMS to ensure **conflict serializability**.

It divides the execution of each transaction into **two phases**:

⭐ **1. Growing Phase**

- A transaction **may obtain locks** (shared or exclusive).
- **Cannot release any lock** in this phase.

Acquire locks → Acquire more locks → (But no release)

⭐ **2. Shrinking Phase**

- A transaction **may release locks**.
- **Cannot obtain any new lock** after it starts releasing.

Release locks → Release more locks → (But no new locks)

✓ **Key Idea:**

The point at which the transaction **acquires its final lock** is called the **lock point**.

After the lock point → only releasing of locks occurs.

✓ **Why use 2PL?**

- It ensures the schedule is **conflict serializable**.
- Prevents conflicts by controlling when locks can be acquired or released.

⭐ **Example of 2PL**

Consider transaction T:

Growing Phase:

lock-X(A)

lock-S(B)

Shrinking Phase:

unlock(A)

unlock(B)

Transaction cannot do:

unlock(A) → then lock(C) ❌ violates 2PL

⏳ **Limitations of Basic 2PL**

Although 2PL ensures serializability, it does NOT guarantee:

1. **Freedom from deadlocks**
2. **Cascading aborts**

To solve these problems, two stronger versions are used:

- **Strict 2PL**
- **Rigorous 2PL**

✅ **Strict Two-Phase Locking Protocol**

✓ **Rules:**

1. **All exclusive (write) locks are held until the transaction commits or aborts.**
2. Shared (read) locks may be released earlier, BUT **write locks cannot be released early**.

✓ **Prevents:**

- **Cascading aborts**
- **Dirty reads**

✓ **Still allows:**

- Shared locks to be released during execution
- Thus, concurrency is higher than rigorous 2PL

⭐ **Example (Strict 2PL)**

Transaction T1:

lock-S(A) → may unlock early

lock-X(B) → must hold till commit/abort

So:

unlock(A) ✓ allowed

unlock(B) ❌ not allowed before commit

⭐ **Why Strict 2PL is Useful?**

- If T1 writes B and T2 reads B before T1 commits → T2 might read incorrect (dirty) data.
- Strict 2PL prevents this by holding **write-locks until commit**.

This is the most commonly used protocol in real DBMS.

✅ **Rigorous Two-Phase Locking Protocol**

✓ **Rules:**

1. **Both shared and exclusive locks are held until the transaction commits or aborts.**
2. No locks are released during the execution of the transaction.

✓ **Strongest version of 2PL.**

⭐ **Example (Rigorous 2PL)**

Transaction T:

lock-S(A)

lock-X(B)

...

commit

unlock(A)

unlock(B)

Here:

- Both S and X locks are released **only after commit**.

⭐ **Advantages of Rigorous 2PL**

- Ensures **strict serializability** (the strongest correctness guarantee).
- Ensures **no cascading aborts**.
- Produces schedules equivalent to **serial schedules in commit order**.

⭐ **Comparison Table (Perfect for Exams)**

Feature	**Basic 2PL**	**Strict 2PL**	**Rigorous 2PL**
Ensures Serializability	✓ Yes	✓ Yes	✓ Yes

Feature	Basic 2PL	Strict 2PL	Rigorous 2PL
Cascading Aborts Prevented	✗ No	✓ Yes	✓ Yes
Dirty Reads Prevented	✗ No	✓ Yes	✓ Yes
When Locks Released	Anytime in shrinking phase	All X-locks at commit	All locks at commit
Strength	Weakest	Strong	Strongest
Real DBMS Usage	Rare	Very common	Sometimes used for high isolation

---

⭐ **Final Summary**

Two-Phase Locking Protocol

- Growing phase → only acquire locks
- Shrinking phase → only release locks
- Ensures **conflict serializability**

Strict 2PL

- Write locks held until commit
- Prevents cascading aborts
- Widely used in real DBMS

Rigorous 2PL

- **All locks (read + write)** held until commit
- Strongest isolation level
- Guarantees strict serializability

Q7. To ensure atomicity despite failures we use Recovery Methods . Explain in detail following Log-Based Recovery methods with example . 1)Deferred Database Modifications 2)Immediate Database Modifications

Ans:- **Quick conceptual difference (one-line)**

- **Deferred database modification (deferred-update):** a transaction's *actual data-page writes are postponed until the transaction commits.* → **Only REDO** is needed during recovery.
- **Immediate database modification (immediate-update):** a transaction's updates **may be written to the database before commit.** → **UNDO and REDO** may both be needed.

Shared requirements for both methods

- A **stable log** (on disk) that survives crashes.
- **Log records** record enough info to redo or undo updates.
- **Write-Ahead Logging (WAL)** principle applies: necessary log records must be forced to stable storage before the corresponding database pages are written (details vary slightly by scheme — I call this out below).

---

1) Deferred Database Modification (Deferred-Update)

Key idea

All updates made by a transaction are *kept in volatile memory (buffers)* and **not written to the disk** until the transaction reaches the COMMIT point. The log records of the updates are written, but the DB pages are not changed on disk until commit time.

Typical log records

- <T, START>
- <T, Q, old_value, new_value> — often the log stores just the *new values* or both (we'll show both for clarity).
- <T, COMMIT>

Important: data pages are not written to disk until the <T, COMMIT> record has been produced and the corresponding pages are flushed.

Write-Ahead Logging rule (deferred)

- Log entries for the updates and the <T, COMMIT> must be on stable storage before any dirty pages for that transaction are written to disk at commit time. (We force pages at commit, but the ordering still matters.)

What recovery must do

- Since no transaction has written its updates to disk before commit, **there is no need to UNDO** — uncommitted transactions left nothing on disk.
- On recovery, **REDO** all updates of transactions that had committed but whose changes may not yet be on disk.

Example schedule & log

Assume initial DB: A=100, B=200

Transaction T1 updates A from 100→150. Transaction T2 updates B from 200→250.

Log (chronological):

1. <T1, START>

2. <T1, A, 100, 150> -- update recorded in log, not written to DB yet

3. <T2, START>

4. <T2, B, 200, 250> -- recorded only in log, not written to DB

5. <T1, COMMIT> -- T1 commits; now its pages will be forced to disk

6. (DB pages for A are written to disk: A=150)

7. [system crash occurs here]

Recovery actions after crash

- Scan log to find committed transactions (T1 is committed; T2 is not).
- **Redo T1**: ensure A=150 on disk — if the page write at step 6 had happened, REDO is idempotent; if it had not finished, redo writes A=150.
- **No UNDO for T2** because T2 did not commit and never wrote to disk.

Advantages & disadvantages

-
    - Simpler recovery (only REDO).
-
    - No cascading aborts (uncommitted changes are not visible on disk).
- − Lower concurrency/performance during commit: forced writes of all modified pages at commit time (burst I/O).
- − Not widely used for high-performance systems because holding updates until commit can hurt concurrency and response time.

---

2) Immediate Database Modification (Immediate-Update)

Key idea

Transactions may **write their updated values to database pages on disk before the transaction commits**. Because uncommitted values *may be on disk*, the recovery mechanism must be able to **undo** those changes if the transaction aborts or if a crash occurs before commit. Also, committed transactions whose writes may not have reached disk must be **redone**.

Typical log records

- <T, START>
- <T, Q, old_value, new_value> — we must log the **old** value (for undo) and the **new** value (for redo).
- <T, COMMIT> or <T, ABORT>

Write-Ahead Logging (WAL) rule (immediate)

- **Before any page containing the updated value is written to disk, the corresponding log record (including the old value and/or new value) must be flushed to stable storage.** This guarantees the system can UNDO if needed even if the page on disk contains uncommitted data.

What recovery must do

Recovery typically has two phases:

(a) REDO phase — reapply all actions of transactions that committed before the crash (to ensure their effects are on disk).
(b) UNDO phase — undo actions of transactions that did not commit (to remove their possibly-on-disk changes).

A common approach: scan forward to find transactions that committed; then redo from the earliest necessary point (often beginning of log or last checkpoint); then undo uncommitted transactions in reverse log order (to restore old values).

Example schedule & log

Initial DB: A=100, B=200

Transactions:

- T1: read A, write A=150 (writes page to disk before commit)
- T2: read B, write B=250

Log (chronological) and actions:

1. <T1, START>

2. <T1, A, 100, 150> -- log flushed to stable storage

3. (Page A is written to disk with A=150) -- allowed in immediate-update

4. <T2, START>

5. <T2, B, 200, 250> -- log flushed

6. (Before T2 commits, system crashes)

Note: T1 may or may not have committed; assume T1 did NOT commit (no <T1, COMMIT>), and T2 did not commit either.

Recovery actions after crash

- Identify committed transactions: none in this example.
- **REDO phase**: none, because no transaction committed.
- **UNDO phase**: find all transactions that were active (T1, T2) and undo their updates using the old_value in the log:
    - Undo T2: set B := 200
    - Undo T1: set A := 100

This restores the disk to a consistent state (no uncommitted updates remain).

If T1 had committed before crash

If T1 had a <T1, COMMIT> in log before crash, then on recovery:

- **REDO** T1 (ensure A=150 on disk).
- **UNDO** T2 (set B back to 200).

Advantages & disadvantages

-
    - Higher concurrency: changes can be made visible earlier (less delay).
-
    - Better performance for long transactions (no need to force all pages at commit).
- − Recovery is more complex (both UNDO and REDO).
- − Must strictly enforce WAL (log must be flushed before data pages) — otherwise you can't undo.

Detailed algorithms (high-level)

Deferred-update recovery algorithm

1. During normal execution, record update log entries; do not write DB pages.
2. At <T, COMMIT>:
    - Force log entries for T and <T, COMMIT> to stable storage.
    - Force all modified DB pages for T to disk.
3. On crash:
    - Scan log forward from last checkpoint to identify transactions with <COMMIT>.
    - **Redo** all updates (in log order) for transactions that committed but whose page writes might be missing.

Immediate-update recovery algorithm (typical ARIES-like pattern simplified)

1. For each data update, write <T, Q, old, new> to log and **force log to stable storage** before the page write (WAL).
2. DB pages may be written anytime after log flush.
3. On crash:
    - Identify committed transactions by scanning log.
    - **Redo** phase: redo all actions starting from a safe point (e.g., last checkpoint) for transactions that committed (and possibly for some uncommitted depending on algorithm).

- **Undo** phase: for transactions that did not commit, traverse their log entries backward and apply old values to disk (undo), writing compensation log records if needed.

(Modern systems like ARIES refine this with repeating history, CLRs, and partial REDO/UNDO, but the high-level UNDO+REDO idea remains.)

Checkpoints — how they help

- Both methods use **checkpoints** to avoid scanning the whole log at recovery.
- At checkpoint, DBMS notes which transactions are active; recovery needs only start scan from the last checkpoint's log position (and then follow from there).

Summary comparison table

Aspect	**Deferred-Update**	**Immediate-Update**
When pages written to DB	At **commit** only	**Anytime** (before or after commit)
Log contents required	Usually new values (for redo)	**Both** old and new (for undo & redo)
Recovery actions needed	**Only REDO**	**Both UNDO & REDO**
WAL requirement	Log before page write at commit	**Strict WAL needed** before any page write
Concurrency	Lower (flush many pages at commit)	Higher (pages can be written earlier)
Cascading aborts	Avoided (uncommitted changes not on disk)	Possible unless locking/other measures used
Practical usage	Simpler but less common in high-performance DBMS	Widely used (with WAL + compensation log records)

Q8. Suppose a transaction T1 issues a read command on data item Q . How time-stamp based protocol decides whether to allow the operation to be executed or not using time-stamp based protocol of concurrency control. Explain in detail time-stamp based protocol.

Ans:- **Time-Stamp Based Concurrency Control Protocol (TSO Protocol)**

The **Time-Stamp Ordering (TSO)** protocol is a concurrency control method that ensures **serializability** by ordering all transactions in the order of their **timestamps**.

✅ 1. What is a Timestamp?

Each transaction **T** is assigned a unique timestamp:

TS(T) = Timestamp of the transaction T

It decides the **serial order** in which transactions should logically appear.

- If **TS(T1) < TS(T2)**
  → T1 should appear before T2 in the serial schedule.

⭐ 2. Timestamps for Data Items

Each data item **Q** is associated with two timestamps:

✓ R-timestamp(Q)

The **largest timestamp** of any transaction that **successfully read** Q.

✓ W-timestamp(Q)

The **largest timestamp** of any transaction that **successfully wrote** Q.

These help decide if a new read/write request violates timestamp order.

⭐ 3. Rules Used to Check Read/Write Operations

The protocol uses **three rules**:

Rule for Read(Q) by T → ReadTS(Q)

When **T issues Read(Q)**:

- If
- TS(T) < W-timestamp(Q)

Then T wants to read an **old obsolete value**.
→ **Reject read** and **abort T**.

- Else
  → Allow the read.
  → Update
- R-timestamp(Q) = max(R-timestamp(Q), TS(T))

Rule for Write(Q) by T → WriteTS(Q)

When **T issues Write(Q)**:

- If

- TS(T) < R-timestamp(Q)

T is trying to write an outdated value.
→ **Abort T** (write too late).

- If
- TS(T) < W-timestamp(Q)

T is also trying to write a value that is too old.
→ **Abort T** (write obsolete).

- Else
  → Allow the write.
  → Update
- W-timestamp(Q) = TS(T)

⭐ **No waiting is allowed**

If a conflict occurs → the transaction **must be rolled back and restarted** with a new timestamp.

📌 **Summary Table (For Exams)**

Operation	Allow?	Condition	Action if Violated
Read(Q)	Allow	TS(T) ≥ W-TS(Q)	Abort T
Write(Q)	Allow	TS(T) ≥ R-TS(Q) and TS(T) ≥ W-TS(Q)	Abort T

⭐ ⭐ **EXAMPLE 1: Read(Q) decision**

Let timestamps be:

For Q:

R-timestamp(Q) = 20

W-timestamp(Q) = 30

Transaction:

TS(T1) = 25

T1 issues Read(Q):

Check rule:

Is TS(T1) < W-timestamp(Q)?

Is 25 < 30 ? → YES

Meaning:
A **younger transaction (TS=30)** already wrote Q.
T1 (older transaction) wants to read a value **older** than what is currently written → **invalid read**.

✓ **Result: Read is rejected, T1 is aborted.**

⭐ ⭐ **EXAMPLE 2: Read(Q) allowed**

Let:

R-timestamp(Q) = 40

W-timestamp(Q) = 50

TS(T1) = 60

TS(T1) > W-TS(Q) → 60 > 50

So T1 is **newer**, reading the latest written value.

✓ **Read Allowed**

Update:

R-timestamp(Q) = max(40, 60) = 60

⭐ **Complete Explanation of TSO Protocol**

1️⃣ **Assign a timestamp to every transaction**

- Based on system clock
- Or logical counter

2️⃣ **All operations must follow timestamp order**

Older transaction → must appear before newer one.

3️⃣ **Operations violating order are aborted**

Because they cause non-serializable schedules.

4️⃣ **No deadlocks occur**

Since **waiting is not allowed**, deadlocks are impossible.

5️⃣ **Guarantees Conflict Serializability**

All approved operations follow timestamp order.

⭐ **Advantages**

- No deadlocks

- Simple validation based on timestamps
- Ensures serializability

⭐ **Disadvantages**

- Many rollbacks if conflicts are frequent
- Long transactions often restart repeatedly

⭐ **Final Exam-Ready Answer (Short Form)**

Time-stamp ordering protocol ensures serializability by ordering all transactions according to their timestamps. Each data item Q has two timestamps: R-timestamp(Q) and W-timestamp(Q). When a transaction T issues Read(Q) or Write(Q), the protocol checks these timestamps to decide whether the operation obeys the timestamp order.

Q9. State and explain the ACID properties .During its execution a transaction passes through several states, until it finally commits or aborts . List all possible sequences of states through which a transaction may pass. explain the situations when each state transition occurs.

Ans:- **ACID Properties of a Transaction**

A transaction in DBMS must satisfy **ACID** properties to ensure correctness, consistency, and reliability.

🟩 **1. A – Atomicity**

- A transaction must be treated as a **single, indivisible unit**.
- Either **all operations** of the transaction are executed, or **none** of them are.
- If a failure occurs → the transaction is **rolled back**.

Example:
In a bank transfer:

T: Withdraw 1000 from A

Deposit 1000 to B

If the deposit fails → the withdrawal must be undone.

🟩 **2. C – Consistency**

- A transaction must take the database from **one consistent state to another**.
- No integrity constraints should be violated.

Example:
Account balances must not become negative (if constraint exists).

🟩 3. I – Isolation

- Concurrent transactions must not interfere with each other.
- The result should be **equivalent to some serial execution**.

Example:
If T1 and T2 run simultaneously, they should behave as if they were run in order (T1 then T2 or T2 then T1).

🟩 4. D – Durability

- Once a transaction **commits**, the changes must be **permanent**.
- Even if the system crashes after commit, changes should survive.

Example:
After successful money transfer, the new balances must survive any crash.

🟦 ACID Properties Summary Table (Exam Perfect)

Property	Meaning	Ensures
Atomicity	All or nothing	Transactions are indivisible
Consistency	Constraint preserved	Database integrity
Isolation	No interference	Correctness under concurrency
Durability	Permanent results	Recovery from failures

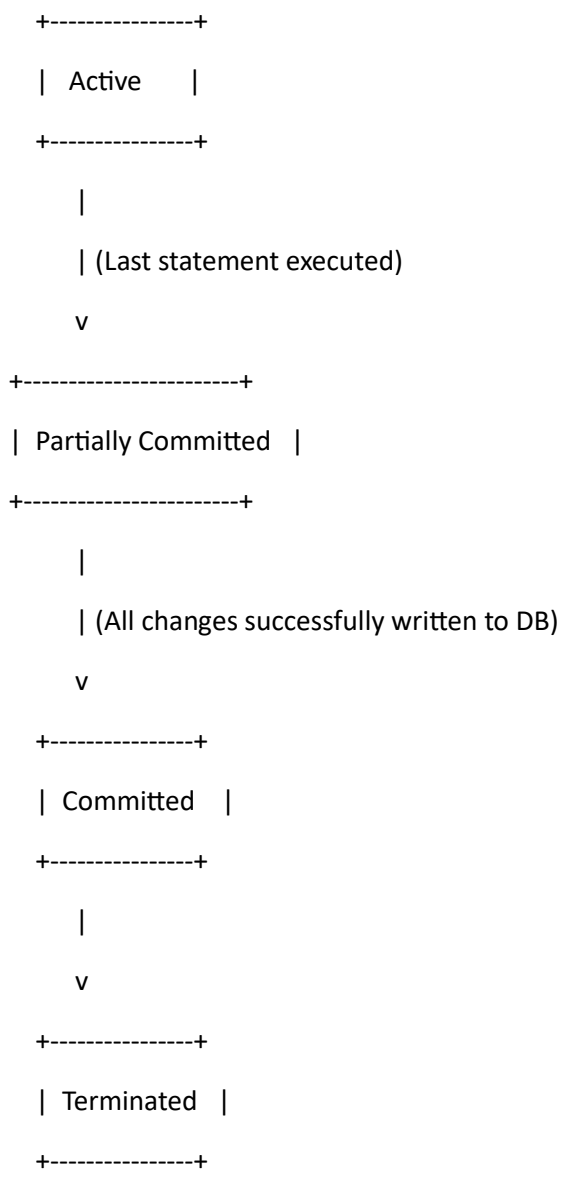
⭐ Transaction States in DBMS

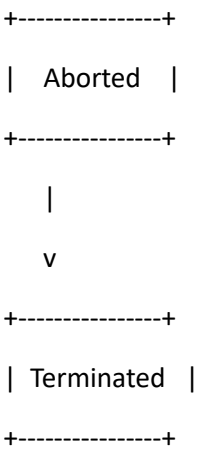
A transaction passes through several states during its execution.

The major states are:

1. **Active**
2. **Partially Committed**
3. **Committed**
4. **Failed**
5. **Aborted**
6. **Terminated**

⭐ State-Transition Diagram (Text Form)


Active
(Last statement executed)
Partially Committed
(All changes successfully written to DB)
Committed
Terminated
(Error/Failure)
Failed
(Rollback completes)



⭐ **State Explanation + When Transitions Occur**

🟢 **1. ACTIVE State**

A transaction enters active state when it **starts execution**.

✓ It performs read/write operations.
✓ Remains active until:

- It finishes its final instruction, or
- It encounters a failure.

Transition:

- On successful completion → **Partially Committed**
- On error → **Failed**

🟡 **2. PARTIALLY COMMITTED State**

The transaction has executed **its final instruction**.

✓ Only final checks (writing to database, logging) remain.

Transition:

- If updates are successfully written → **Committed**
- If failure occurs during final write → **Failed**

🟢 **3. COMMITTED State**

The transaction has **successfully completed** and changes are made permanent.

✓ Effects cannot be undone (except by another compensating transaction).

Transition:

- After completing commit → **Terminated**

🔴 4. FAILED State

The transaction cannot continue due to:

- Logical error
- System crash
- Deadlock
- Concurrency conflict
- Invalid operation

Transition:

- System starts rollback → **Aborted**

🟣 5. ABORTED State

Rollback is performed and all changes are **undone**.

After rollback, two possibilities:

1. Transaction may be **restarted**
2. Or may be **killed** and ended

Transition:

- Once cleanup is done → **Terminated**

⚪ 6. TERMINATED State

The transaction ends completely.

No further operation possible.

⭐ All Possible State Sequences (Very Important for Exams)

1. **Normal successful execution:**

Active → Partially Committed → Committed → Terminated

2. **Failure before partial commit:**

Active → Failed → Aborted → Terminated

3. **Failure during partial commit:**

Active → Partially Committed → Failed → Aborted → Terminated

4. **Restart after abort (optional):**

Active → Failed → Aborted → Active → …

⭐ **Final Summary**

ACID Properties:

- **Atomicity** → all or nothing
- **Consistency** → valid state → valid state
- **Isolation** → no interference
- **Durability** → permanent after commit

States & Transitions:

Transaction passes through:
Active → Partially Committed → Committed → Terminated,
or
Active → Failed → Aborted → Terminated

Failures at any stage trigger rollback.

Q10. A transaction may be waiting for more time for an Exclusive(X) lock on an item, while a sequence of other transactions request and are granted as Shared(S) lock on the same item. What is this problem? How is it solved by two phase lock protocol?

Ans:- **What is the Problem?**

When a transaction **T1** requests an **Exclusive (X) lock** on a data item **Q**, it must wait until:

- no other transaction holds a Shared (S) lock
- and no new transactions acquire S locks during the wait.

But the issue is:

✓ **While T1 is waiting for an X-lock,**

other transactions keep requesting and acquiring S-locks on Q, because S-locks are compatible with each other.

This prevents the X-lock from ever being granted.

🔴 **This Problem is called: "Starvation" (or X-lock starvation)**

Why starvation occurs?

- The X-lock request is **waiting**.

- Meanwhile, many incoming S-lock requests are immediately granted (since they do not conflict with existing S-locks).
- As long as S-locks keep coming, the X-lock is **never granted**.

This means **T1 starves** because it waits indefinitely.

⭐ **Example of X-Lock Starvation**

1. T1 requests **lock-X(Q)** → must wait
2. T2 requests **lock-S(Q)** → granted
3. T3 requests **lock-S(Q)** → granted
4. T4 requests **lock-S(Q)** → granted

…and so on.

➞ **T1 keeps waiting forever**.

⭐ **How Does Two-Phase Locking (2PL) Solve This Problem?****

Two-Phase Locking introduces **a strict locking discipline** which indirectly prevents starvation:

🟩 **1. No new locks can be acquired after unlock begins**

In 2PL, a transaction:

✓ **In Growing phase → acquires locks**

✓ **In Shrinking phase → releases locks but cannot acquire new locks**

This ensures:

- Older transactions (with earlier timestamps or earlier lock requests) get locks **before** newer ones.

🟩 **2. Lock Requests Are Ordered in a Queue**

In most 2PL systems:

- Every lock request is placed in a **first-come, first-served (FCFS) queue**.

Thus:

✓ **Once T1 requests X-lock,**

all further S-lock requests are placed **behind T1** in the queue.

➞ S-locks **cannot jump ahead** of the waiting X-lock request.

⭐ **Illustration: 2PL Queue Prevents Starvation**

Queue before 2PL:

S (T2) → S (T3) → S (T4) → X (T1) ❌ T1 waits forever

Queue under 2PL:

X (T1) → S (T2) → S (T3) → S (T4) ✓ T1 gets X-lock next

Thus, no new shared locks are granted until the **waiting exclusive lock is handled**.

🟩 **3. Strict 2PL and Rigorous 2PL Improve Further**

- Strict 2PL: all X-locks held until commit
- Rigorous 2PL: all S and X locks held until commit

This ensures transactions release locks only at commit, reducing chances of late-arriving S-lock dominance.

⭐ **Summary (Perfect for Exams)**

⭐ **The Problem:**

When a transaction waits for an Exclusive lock, multiple other transactions continuously obtain Shared locks on the same item, **causing X-lock starvation**.

⭐ **How 2PL Solves It:**

1. 2PL enforces a strict **locking order** (growing → shrinking).
2. Lock manager maintains a **FIFO queue**.
3. Once an X-lock is requested, **no new S-locks are granted** before the X-lock is processed.
4. Strict and rigorous 2PL prevent releasing locks early, reducing conflicts.

➞ Thus, **starvation is prevented**, and X-lock eventually gets granted.

Q11. What is conflict serializability? How to check schedule is conflict serializable schedule? Explain with example.

Ans:- **What is Conflict Serializability?**

A schedule is **conflict serializable** if it can be **converted into a serial schedule** (i.e., transactions executed one after another) **by swapping non-conflicting operations**.

Operations conflict if they:

1. Belong to **different transactions**

2. Access the **same data item**
3. And **at least one** of them is a **write**

✓ **Types of Conflicting Pairs:**

- **Read–Write (R–W)**
- **Write–Read (W–R)**
- **Write–Write (W–W)**

If you can reorder the schedule by swapping non-conflicting operations and get a serial schedule → the schedule is **conflict serializable**.

⭐ **How to Check Conflict Serializability?**

We use a **Precedence Graph (Serialization Graph)**.

✓ **Steps:**

Step 1: Create a node for each transaction.

Step 2: For every conflicting operation pair:

Add an **edge Ti → Tj**
if an operation of **Ti** occurs before and conflicts with an operation of **Tj**.

Step 3: Check for cycles

- **No cycle → conflict serializable**
- **Cycle → NOT conflict serializable**

⭐ **Example 1: Conflict Serializable Schedule**

Schedule S_1:

T1: R(A) W(A)

T2: R(A) W(A)

Let's rewrite as a sequence:

R1(A), W1(A), R2(A), W2(A)

Step 1: Identify conflicts:

Pair	Reason	Direction
W1(A) → R2(A)	W–R conflict	T1 → T2
R1(A) → W2(A)	R–W conflict	T1 → T2

Pair	Reason	Direction
W1(A) → W2(A)	W–W conflict	T1 → T2

Step 2: Draw graph:

T1 → T2

No reverse edge (T2 → T1), so **no cycle**.

✓ Conclusion: Serial order is T1 → T2 → Schedule is Conflict Serializable.

⭐ **Example 2: Not Conflict Serializable (Cycle Present)**

Consider schedule S_2:

R1(A)

W2(A)

R2(B)

W1(B)

Step 1: List the conflicts:

Operations	Conflict	Edge
R1(A) before W2(A)	R–W	T1 → T2
R2(B) before W1(B)	R–W	T2 → T1

Step 2: Precedence graph:

T1 → T2

T2 → T1

Cycle is present.

❌ **Conclusion:**

Not conflict serializable
because no serial order can satisfy both directions.

⭐ **Example 3: Step-by-Step Graph Construction (Perfect for 7-Mark Answer)**

Schedule S_3:

R1(X)

R2(X)

W1(X)

R3(X)

W2(X)

W3(X)

Step 1: Find conflicts:

Conflicts on X:

- R1–W1 → no (same transaction)
- R1–W2 → T1 → T2
- R1–W3 → T1 → T3
- R2–W1 → T2 → T1
- R2–W2 → no (same)
- R2–W3 → T2 → T3
- W1–R3 → T1 → T3
- W1–W2 → T1 → T2
- W1–W3 → T1 → T3
- R3–W2 → T3 → T2
- R3–W3 → no
- W2–W3 → T2 → T3

Step 2: Edges:

- T1 → T2
- T1 → T3
- T2 → T1 (because R2 → W1)
- T2 → T3
- T3 → T2 (because R3 → W2)

Graph contains cycles:

T1 ↔ T2

T2 ↔ T3

❌ **Not conflict serializable.**

⭐ **Why Use Conflict Serializability?**

- Easy to check using a graph
- Used in theory & practice in concurrency control

- Ensures schedules behave like **serial schedules** → guarantee correctness

---

⭐ **Final Short Exam Answer**

Conflict Serializability is a type of serializability where a schedule can be transformed into a serial schedule by swapping non-conflicting instructions. Operations conflict when they belong to different transactions, operate on the same data item, and at least one is a write. A schedule is conflict serializable if its precedence graph has no cycles.

To check:

1. Create nodes for each transaction.
2. For every conflicting operation Ti → Tj, add an edge.
3. If the graph has **no cycles**, the schedule is conflict serializable; if there is a **cycle**, it is not.

Q12. Explain two phase locking protocol. Consider the following two transactions.

T1:read(A); read(B);

if A=0 then B=B+1; Write(B).

T2:read(B); read(A);

if B=0 then A=A+1; Write(A).

Add lock and unlock instructions to transactions T1 and t2, so that they observe the two phase locking protocol. can the execution of these transactions result in a deadlock?

Ans:- **1) Two-Phase Locking (brief)**

- Growing phase: a transaction may **acquire** any number of locks (S or X).
- Shrinking phase: once it **releases** any lock, it cannot acquire any new lock.
- **Strict 2PL**: all X locks (and often all locks) are held until commit — prevents cascading aborts.

Locks:

- lock-S(X) — shared lock for reading X (many transactions may hold S simultaneously).
- lock-X(X) — exclusive lock for writing X (only one transaction; blocks reads/writes by others).
- unlock(X) — releases the lock on X.
- COMMIT / ABORT

---

2) Given transactions (original)

T1:

read(A);

read(B);

if A = 0 then B = B + 1;

write(B);

T2:

read(B);

read(A);

if B = 0 then A = A + 1;

write(A);

We must add lock/unlock so they obey 2PL. Two common safe ways:

A — Standard 2PL with lock upgrade (strict 2PL: hold locks until commit)

Use S locks for reads, upgrade to X before write, and release all locks at commit.

T1 (standard 2PL, strict)

lock-S(A);

read(A);

lock-S(B);

read(B);

/* need to write B -> upgrade S(B) to X(B) (or acquire X(B) if supported) */

if (A = 0) then {

  upgrade lock-S(B) to lock-X(B); /* or lock-X(B) after releasing S(B) if allowed — but releasing then re-acquiring breaks 2PL */

  B = B + 1;

  write(B);

}

/* hold locks until commit in strict 2PL */

COMMIT;

unlock(A);

unlock(B);

T2 (standard 2PL, strict)

```
lock-S(B);
read(B);

lock-S(A);
read(A);

if (B = 0) then {
   upgrade lock-S(A) to lock-X(A);
   A = A + 1;
   write(A);
}

COMMIT;
unlock(B);
unlock(A);
```

Note: upgrade/convert means you request an X lock while holding S. Many lock managers support an atomic upgrade request; if two transactions try to upgrade different locks in opposite order deadlock is still possible.

B — Conservative 2PL (obtain all locks before doing any operation)

Acquire all needed locks in a fixed order before any read/write; this avoids upgrade-related deadlocks.

T1 (conservative, acquire in A then B order)

```
lock-S(A);
lock-X(B);      /* if you know you might write B, you take X(B) up front */
read(A);
read(B);
if (A = 0) {
   B = B + 1;
   write(B);
}
COMMIT;
```

unlock(A);

unlock(B);

T2 (conservative, same global order: A then B)

lock-S(B); /* but to respect same global order, we should lock A then B -> see recommended ordering below */

...

(This version reveals why we prefer a consistent global order — see below for corrected ordering.)

---

3) Can execution lead to a deadlock? (Yes — demonstration)

Using the **standard 2PL** version above (T1: S(A) then S(B) then upgrade S(B)→X(B); T2: S(B) then S(A) then upgrade S(A)→X(A)), consider this interleaving:

1. T1: lock-S(A) — granted.
2. T1: read(A).
3. T2: lock-S(B) — granted.
4. T2: read(B).
5. T1: lock-S(B) — granted (S locks are compatible), read(B).
6. T2: lock-S(A) — request S(A) but T1 already holds S(A). S is compatible so T2 may get S(A) *only if lock manager allows multiple S on same item* — in many systems S is allowed, so T2 gets S(A) too. (Alternatively T2 waited earlier if manager queued; but typical pattern is both have S on both items.)
7. Now both transactions try to **upgrade**:
    - T1 requests upgrade S(B → X(B)). This blocks because T2 holds S(B) (and T2 hasn't released it).
    - T2 requests upgrade S(A → X(A)). This blocks because T1 holds S(A).

Now T1 waits for T2 and T2 waits for T1: **circular wait → deadlock**.

So yes — with opposing lock acquisition/upgrades a circular wait arises and deadlock occurs.

---

4) Why did deadlock happen (concise)

- T1 requested locks in order A then B; T2 requested in order B then A.
- Each acquired S locks, then tried to upgrade to X on the other item → cycle.
- Two-phase locking by itself **does not prevent deadlock**; 2PL guarantees serializability but not freedom from deadlocks.

---

5) How to avoid this deadlock (practical solutions)

(a) Enforce a global lock ordering (recommended)

Pick a global order of data items (e.g., alphabetical: A < B). **Always acquire locks in that order**.

So both transactions must lock A then B (or request X for both in that order). Example safe versions:

T1 (locks in A then B)

lock-S(A);

lock-X(B); /* or lock-S(B) then upgrade to X(B) but keep order A then B */

read(A); read(B);

...

COMMIT; unlock(A); unlock(B);

T2 (also locks A then B — note it must request A first even though original read order was B then A)

lock-S(A); /* must be acquired first to maintain ordering */

lock-S(B);

read(B); read(A);

...

COMMIT; unlock(A); unlock(B);

If both transactions follow same global order, the cycle cannot form.

(b) Use conservative locking (acquire all locks up front)

Before executing, transaction obtains all locks it may need (in global order). If cannot get them all, it releases and retries. This prevents deadlock.

(c) Use deadlock detection + victim selection

Allow deadlocks to occur, detect cycle in Wait-For Graph periodically, abort one victim (e.g., the one with least cost or youngest), release its locks, continue.

(d) Use deadlock avoidance schemes (wait-die / wound-wait)

Use timestamp-based policies to decide whether a waiting transaction waits or is rolled back, preventing cycles.

(e) Timeouts

If a transaction waits too long, abort it — simple but can cause unnecessary aborts.

---

6) Recommended exam answer (concise)

1. **Add S/X locks and COMMIT/UNLOCK** to both transactions so they follow 2PL (example code provided above).

2. **Yes**, the execution can deadlock: T1 acquires S(A) then tries to upgrade to X(B) while T2 acquires S(B) then tries to upgrade to X(A) — circular wait.
3. **Solution:** enforce a global lock acquisition order (A then B) or use deadlock detection/avoidance (wait-die/wound-wait), or conservative locking to prevent the deadlock.

Q13. What is recoverable schedule ? Why is recoverability of schedule desirable ? Are there any circumstance under which it could be desirable to allow non recoverable scheduler? Explain your answer.

Ans:- **1. What is a Recoverable Schedule?**

A **schedule is recoverable** if **a transaction commits only after all transactions from which it has read have committed**.

In simple words:

✓ **If T2 reads a value written by T1,**

T2 must commit **after** T1 commits.

This prevents T2 from **committing based on uncommitted (dirty) data**.

🔹 **Example of a Recoverable Schedule**

T1: W(X)

T2: R(X)

T1: COMMIT

T2: COMMIT

Here:

- T2 reads X written by T1
- T2 commits **after** T1 commits → ✓ Schedule is **recoverable**

⭐ **2. Why is Recoverability Important?**

Recoverability is crucial to prevent **cascading aborts** and **inconsistent database states**.

If the schedule is NOT recoverable:

T2 might commit using T1's uncommitted (dirty) data.

If T1 aborts later →
✓ DB must rollback T2
✓ But T2 has already committed → **inconsistent database**

Thus:

🔸 **Recoverable schedules ensure:**

- No committed transaction depends on aborted data
- Database maintains **consistency**
- System avoids **irreversible errors**

⭐ 3. When is a Schedule Non-Recoverable?

A schedule is **non-recoverable** if **a transaction commits before the transaction from which it read commits**.

Example:

T1: W(X)

T2: R(X)

T2: COMMIT ❌ (Committed based on uncommitted data)

T1: ABORT

- T2 has committed using dirty data from T1 → **Non-recoverable**
- If T1 aborts → DB becomes **inconsistent**

⭐ 4. Are There Any Situations Where Allowing a Non-Recoverable Schedule is Desirable?

In general: No.
Non-recoverable schedules can cause **permanent inconsistencies**, so database systems always avoid them.

But theoretically, there are **two special circumstances** where allowing a non-recoverable schedule may be acceptable:

⭐ Case 1: Read-Only Transactions

If T2 is **read-only**:

- It does not modify the database
- Even if it reads dirty data, it does not commit anything harmful

Thus, some systems allow non-recoverable schedules for **read-only queries** to improve performance.

Example:

- Analytics queries
- Reporting
- Monitoring dashboards

⭐ Case 2: Applications Where Performance is More Important Than Consistency

In rare cases, such as:

- Real-time systems
- Approximate computing
- Sensor data systems
- Streaming analytics

Temporary inconsistency may be acceptable, and performance may be prioritized over strict recoverability.

But **this is NOT done in classical DBMS**.

⭐ Summary (Perfect for Exam)**

✓ A recoverable schedule ensures:

If T2 reads from T1 → T2 must commit **after** T1 commits.

✓ Recoverability is desirable because:

- It avoids **inconsistencies**
- Prevents **committing transactions that depend on aborted ones**
- Ensures safe recovery from failures

✓ Allowing non-recoverable schedules is generally not desirable, but may be allowed:

1. For **read-only transactions**
2. In special systems where **performance > strict consistency**

Q14.

b) What is conflict serializability? Check following schedule is conflict serializable or not? Also, explain the concept of conflict equivalent schedule.

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

R(X) denotes read operation on data item X by transaction Ti.
W(X) denotes read operation on data item X by transaction Ti.

Ans:- **Definition (short):**
Two schedules are **conflict-equivalent** if for every pair of conflicting operations (same data item, different transactions, and at least one is a write) the two schedules order that pair the same way. A schedule is **conflict-serializable** if it is conflict-equivalent to some serial schedule.

---

1) Transcribe the given schedule (time order)

Read the table row-by-row (top → down). The operations occur in this order:

1. T1: R(X)
2. T1: R(Z)
3. T2: W(X)
4. T3: R(Y)
5. T3: W(Y)
6. T4: W(X)
7. T4: W(Y)
8. T4: W(Z)

(There are no other operations.)

---

2) Find all conflicting pairs and add precedence edges

Conflicts are on the same data item across different transactions, where at least one operation is a write.

- **On X**
    - R1(X) before W2(X) → conflict R–W ⇒ edge **T1 → T2**
    - W2(X) before W4(X) → conflict W–W ⇒ edge **T2 → T4**
    - R1(X) before W4(X) → conflict R–W ⇒ edge **T1 → T4**
- **On Y**
    - W3(Y) before W4(Y) → conflict W–W ⇒ edge **T3 → T4**
    - (R3(Y) is same transaction as W3(Y), so no extra cross-transaction conflict there)
- **On Z**
    - R1(Z) before W4(Z) → conflict R–W ⇒ edge **T1 → T4**

No other data item conflicts exist.

So the precedence (serialization) graph has nodes {T1,T2,T3,T4} and edges:

- T1 → T2
- T2 → T4
- T1 → T4
- T3 → T4

---

3) Check for cycles

The graph edges imply the partial order:

- T1 must come before T2 and before T4.
- T2 must come before T4.
- T3 must come before T4.

There is **no edge that leads back** to T1 or T2 or T3 from T4, so **no cycle** exists.

Therefore the schedule is conflict-serializable.

---

4) One possible equivalent serial schedule

Any serial order that respects the edges is acceptable. For example:

T1 → T2 → T3 → T4

This respects T1 before T2, T1 before T4, T2 before T4, and T3 before T4. (T1 and T3 may be ordered either way; many valid serial orders exist as long as T4 is last and T2 after T1.)

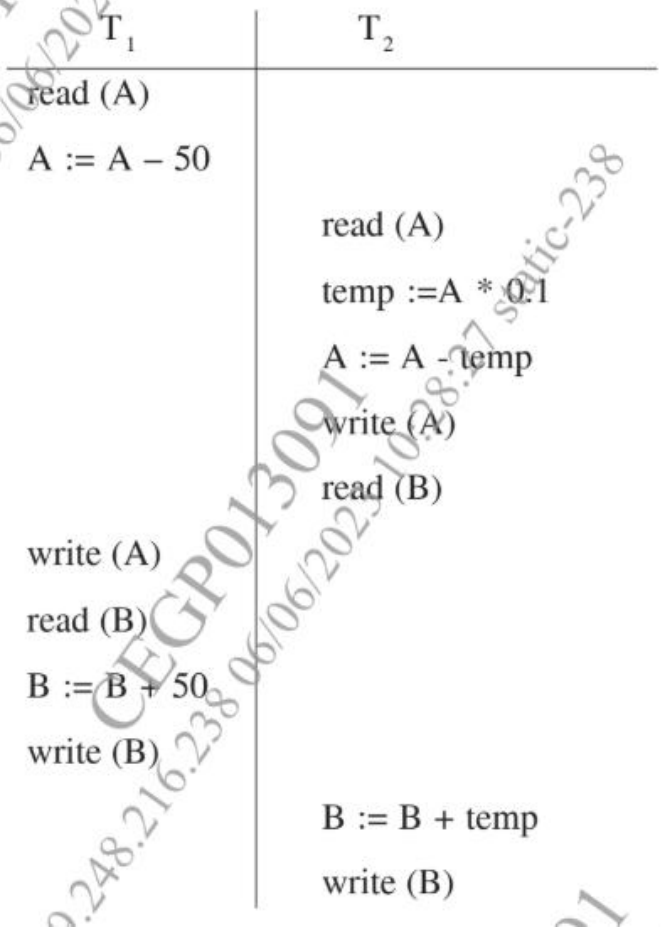
---

5) Concept of conflict-equivalent schedule (brief)

Two schedules S and S' are **conflict-equivalent** if for every pair of conflicting operations (o_i, o_j) that appear in S, they appear in the same relative order in S'. If a schedule is conflict-equivalent to a serial schedule, it is conflict-serializable.

Q15.

b) Check whether following schedule is view serializable or not. Justify your answer. (Note : T_1 & T_2 are transactions). Also explain the concept of view equivalent schedules and conflict equivalent schedule considering the example schedule given below : **[8]**



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)

Ans:- **Short answer up front:**
The schedule is **not view-serializable** (and therefore not conflict-serializable). The reason is that the final-writer condition for view equivalence produces a contradiction: A's final write is by **T1** while B's final write is by **T2**, which forces inconsistent ordering (T2 before T1 for A, and T1 before T2 for B). Hence no serial order can produce the same *view*.

---

1. Transcribe the schedule (operation order)

From the diagram the operations occur (interleaved) in this order:

1. T1: R(A)
2. T1: A := A - 50 (local computation, not yet written)
3. T2: R(A)
4. T2: temp := A * 0.01
5. T2: A := A - temp
6. T2: W(A)
7. T2: R(B)
8. T1: W(A)
9. T1: R(B)
10. T1: B := B + 50
11. T1: W(B)
12. T2: B := B + temp
13. T2: W(B)

Important points visible in this order:

- Both transactions read A initially (T1 reads first, then T2 reads the old value).
- T2 writes A (step 6) **before** T1 writes A (step 8). So the **last writer of A** in the schedule is **T1**.
- T1 writes B (step 11) **before** T2 writes B (step 13). So the **last writer of B** is **T2**.

---

2. Recall the view-equivalence (view-serializability) rules

Two schedules S and S' are **view-equivalent** iff for every data item:

1. The transaction that **reads the initial value** is the same in both schedules.
2. For each read of a value produced by some write, the **read-from relation** (which write provided the value read) is the same.
3. The **final writer** of each data item is the same in both schedules.

A schedule is **view-serializable** if it is view-equivalent to some serial schedule.

A convenient quick test for impossibility: if two transactions each must appear after the other to satisfy final-writer conditions on different items, there's no serial order → not view-serializable.

---

3. Apply the final-writer test to this schedule

- Writers of **A**: both T2 and T1 write A; the *final writer* of A (last W(A)) in the given schedule is **T1** (step 8).
  → Any serial schedule view-equivalent to this one must have **T1 after T2** (so that T1 can be the final writer of A).

- Writers of **B**: both T1 and T2 write B; the *final writer* of B in the schedule is **T2** (step 13).
  → Any view-equivalent serial schedule must have **T2 after T1** (so that T2 can be the final writer of B).

So we must have simultaneously T1 after T2 (from A) and T2 after T1 (from B). This is a contradiction — no serial order can satisfy both. Therefore **no serial schedule is view-equivalent** to the given schedule.

Hence the schedule is **not view-serializable**.

---

4. Connection with conflict-equivalence

(Briefly) Conflict-equivalence is a stronger notion: if a schedule is conflict-serializable then it is view-serializable, but not vice versa. For conflict-serializability we build a precedence graph from conflicting pairs (R/W, W/R, W/W). For this schedule:

- On A: W2(A) occurs before W1(A) → edge T2 → T1.
- On B: W1(B) occurs before W2(B) → edge T1 → T2.

That gives a cycle T1 ↔ T2 so the schedule is **not conflict-serializable** either (consistent with the view-serializability failure).

---

5. Final conclusion (exam-ready)

- **Definition:** Two schedules are *view-equivalent* if they have the same initial reads, the same read-from relations, and the same final writers for every data item. A schedule is *view-serializable* if it is view-equivalent to some serial schedule.
- **Answer for this example:** The schedule is **not view-serializable** (and not conflict-serializable). Reason: the final-write constraints force contradictory transaction orders (T1 after T2 for A and T2 after T1 for B), so no serial ordering can be view-equivalent.

Q16. Write a short note on: 1) Log Based Recovery 2) Shadow Paging

Ans:- **1) Log–Based Recovery (Short Note)**

Log-based recovery is a recovery technique in DBMS that uses a **log file** to record all changes made by transactions. The log ensures that the database can be **restored to a consistent state** after a crash.

Key Points:

- Every transaction writes log records **before** performing actual database updates (Write-Ahead Logging – WAL).
- Log is stored on stable storage (disk), so it survives system failures.
- Log contains records such as:
  - <Ti start>

- <Ti, X, old_value, new_value>
- <Ti commit> / <Ti abort>

Types of Log-Based Recovery:

A) Deferred Database Modification

- Updates are **not** applied to the database until the transaction commits.
- Only **after commit**, actual writes are performed.
- Before commit, logs contain only **new-value** entries.
- Easy recovery: If crash occurs before commit → **no changes to undo**.

B) Immediate Database Modification

- Updates are applied to the database **as soon as they occur**.
- Log contains both **old values (undo)** and **new values (redo)**.
- If crash occurs:
  - Undo all operations of **uncommitted** transactions.
  - Redo all operations of **committed** transactions.

Recovery Steps:

1. **Undo**: Use old values in log of uncommitted transactions.
2. **Redo**: Use new values in log of committed transactions.

Advantages:

- Ensures atomicity and durability.
- Supports efficient crash recovery.

---

2) Shadow Paging (Short Note)

Shadow paging is an **alternative recovery technique** that eliminates the need for log-based recovery by maintaining two page tables during execution.

Key Idea:

- Maintain a **shadow page table** (original table) and a **current page table** (transaction's working version).
- The shadow table **never changes** during the transaction.
- All updates are made only in the current page table.

How it Works:

1. When a transaction begins:
   - A **shadow page table** is created as a copy of current page table.

2. When a page is updated:
    - A **new copy** of that page is created (copy-on-write).
    - The current page table points to this new updated page.
3. On **commit**:
    - The shadow page table is replaced with the current page table (atomic switch).
4. On **abort** or crash:
    - Simply discard the current page table.
    - Shadow page table becomes the new consistent database.

Advantages:

- No need for logs → recovery is very fast.
- Guaranteed consistency (shadow table remains safe).

Disadvantages:

- High overhead of copying pages.
- Inefficient for large databases.
- Not suitable for multi-user environments.