Explain :
[8]
i)
ii)
ACID properties
Explain Timestamp Based Concurrency Control

Here's your answer in **8-mark** style — clear, structured, and complete:

---

i) ACID Properties:

In database systems, **ACID** properties ensure that transactions are processed reliably. ACID stands for:

1. **Atomicity:**
   - A transaction is an indivisible unit of work.
   - Either all operations within the transaction are completed successfully, or none are.
   - If any part of the transaction fails, the entire transaction is rolled back.

2. **Consistency:**
   - A transaction must bring the database from one valid state to another valid state.
   - It ensures that database integrity constraints are not violated.

3. **Isolation:**
   - Concurrent transactions should not interfere with each other.
   - The result should be the same as if transactions were executed sequentially.

4. **Durability:**
   - Once a transaction has been committed, its changes are permanent.
   - Even in case of a system failure, committed changes must persist.

Diagram:

```pgsql
`Transaction --> Commit --> Changes Permanent (Durability)
Transaction --> Error --> Rollback (Atomicity)
Concurrent Transactions --> No Interference (Isolation)
Before Transaction --> After Transaction --> Valid State (Consistency)
`
```

---

ii) Timestamp Based Concurrency Control:

Definition:
Timestamp-Based Concurrency Control is a method where each transaction is assigned a unique **timestamp** when it starts. The system uses these timestamps to control the order of transaction execution, ensuring serializability.

Working:

- Each data item has two timestamps:
  - **W-timestamp(X):** Latest time X was written.
  - **R-timestamp(X):** Latest time X was read.
- When a transaction wants to **read** or **write** a data item, the system compares the transaction's timestamp with the data item's timestamps to decide whether to allow, delay, or abort the transaction.

Rules:

- **Read Rule:**
  - If a transaction's timestamp < W-timestamp(X), → **Abort** (because it wants to read an obsolete value).
  - Else, allow the read and set R-timestamp(X) if needed.
- **Write Rule:**
  - If a transaction's timestamp < R-timestamp(X) or W-timestamp(X), → **Abort** (conflict detected).
  - Else, allow the write operation.

Advantages:

- Simple to implement.
- Ensures conflict-serializable schedules.

Disadvantages:

- Frequent aborts in high-contention systems.
- Not suitable for long transactions.

Diagram:

```scss
`T1 (Timestamp = 10)
T2 (Timestamp = 20)

Data X:
  W-timestamp(X) = 15
  R-timestamp(X) = 12

T1 tries to write X → Timestamp(10) < W-timestamp(X)(15) → T1 Aborts
`
```

Would you also like a small table comparing **Lock-Based** vs **Timestamp-Based** concurrency control? It would be helpful if asked in exams! 📚

What is the need of Serializability?

Here's the **5 to 6 mark** style answer you asked for:

Need of Serializability:

Serializability is needed in database systems to ensure that **concurrent transactions** do not produce incorrect results and the database remains **consistent**.

When multiple transactions execute at the same time, they may access and modify the same data items. Without proper control, this can lead to problems like:

- **Lost Updates:** Two transactions update the same data without awareness of each other, and one update gets lost.

- **Temporary Inconsistency:** Intermediate results of one transaction affect another.

- **Uncommitted Data Usage:** One transaction reads data modified by another transaction that may later be aborted (dirty reads).

Serializability ensures that:

- The **final result** of concurrent transactions is **same** as if transactions were executed **one after another** (serially).

- **Database integrity** and **correctness** are maintained.

- **Concurrency** is allowed for better system performance, but **without compromising accuracy**.

Thus, **serializability is the key correctness criterion** for concurrent transaction execution.

In simple words:
"Serializability is needed to maintain correctness when many users are accessing or modifying the database at the same time."

Would you also like a small real-life example (like a bank transaction example) to make it even stronger if needed in exams? 🚀

c) Check whether given schedule is view serializable?

T1	T2	T3
Read(Q)		
	Write(Q)	
Write(Q)		
		Write(Q)
	OR	

Alright, let's solve it step-by-step:

Question:

Check whether the given schedule is view serializable:

T1	T2	T3
Read(Q)		
	Write(Q)	
Write(Q)		
		Write(Q)

Solution:

🔵 **View Serializability** checks three conditions:

1. **Initial Read:** The first read must match the original data.

2. **Read-From:** If a transaction reads from another transaction's write, the order must be preserved.

3. **Final Write:** The transaction doing the final write must be the same.

Step-by-step:

1. Initial Read

- **T1** reads **Q**.
- Since no prior write to Q happened yet, T1 reads the **original value** of Q.
  ✅ (OK)

---

2. Read-From

- T1 reads Q (original value) ➔ OK.
- T2 writes Q ➔ New value of Q.
- T1 then writes Q ➔ New value.
- T3 also writes Q ➔ Again new value.

Important:

- T1's read happens before any write, so no confusion.
- No transaction reads after writing.
  ✅ (OK)

---

3. Final Write

- The last write on Q is by **T3** (because T3 writes after T1 and T2). ✅ (OK)

Conclusion:

✅ The given schedule is **View Serializable**.

Final Answer:

> **The given schedule is View Serializable.**

Would you also like a simple diagram showing this visually with arrows? 🎯 (helps a lot in exams!)

👤

> What is Log Based Recovery? Explain Deferred Database Modification
> and Immediate Database Modification

Here's the **5 to 6 marks** solution you asked for — neat and clear:

Log-Based Recovery:

Log-Based Recovery is a technique in database systems where all changes made by transactions are recorded in a **log file**.
This log helps the database recover from failures by either **redoing** or **undoing** transactions based on their log entries.

✅ **Log contents include:**

- Transaction start and end.
- Operations like `WRITE` and `READ` with before and after values.
- Commit or Abort status.

If a system crashes:

- If a transaction is **committed**, its changes are **redone**.
- If a transaction is **not committed**, its changes are **undone**.

Deferred Database Modification:

Meaning:
In **Deferred Modification**, the database is **NOT** updated immediately when a transaction issues a `WRITE` command.

- All updates are kept **in memory (buffer)**.
- **Changes are written to the database only after the transaction commits.**

✅ **Key Points:**

- No changes are made before commit.

- If failure happens before commit → No action needed.

- Simple to handle during recovery.

✅ **Example:**

```pgsql
`T1: Write(Q) = 50
(Stored in log/buffer only)
T1: Commit
(Update Q = 50 in the database)
`
```

Immediate Database Modification:

Meaning:
In **Immediate Modification**, the database **may be updated immediately** after a `WRITE` operation, **even before** the transaction commits.

✅ **Key Points:**

- Changes are written to database and log at the same time.

- If transaction aborts → Undo operations are needed.

- More complex recovery because partial changes can happen.

✅ **Example:**

```pgsql
`T1: Write(Q) = 50
(Update database immediately)
T1: (Still not committed)
System crash occurs
(Need to undo changes of T1)
`
```

Difference Table:

Aspect	Deferred Modification	Immediate Modification
Database Update Timing	After transaction commit	As soon as write operation occurs
Recovery	Only Redo needed	Undo and Redo both needed
Complexity	Simple	Complex

Would you also like a quick small diagram showing both workflows side-by-side? 📈 (It'll make it even easier!)

Write a note on "Shadow Paging"

Here's your **5 to 6 mark** answer for "Shadow Paging":

Shadow Paging:

Shadow Paging is a **recovery technique** used in databases to provide **atomicity** and **durability** without using logs.

✅ **Idea:**

- Maintain **two page tables**:
    1. **Current Page Table** (for ongoing changes)
    2. **Shadow Page Table** (unchanged, backup)
- The **Shadow Page Table** always points to the **original, stable** database pages.
- Changes are made to **new copies** of the pages (not the originals).
- **Only after successful commit**, the current page table replaces the shadow page table.

Working Steps:

1. At the start of a transaction:
   ➜ Create a copy of the page table (**Shadow Page Table**).

2. During the transaction:
   ➜ Any modified page is copied and changed in a **new location**.

3. On commit:
   ➜ Current Page Table becomes the official table.
   ➜ Shadow Page Table is discarded.

4. On failure:
   ➜ Shadow Page Table is used to **restore** the database (no undo required).

Advantages:

- **No undo or redo** needed.
- **Simple** to implement.

Disadvantages:

- **Copying page tables** can be **expensive** for large databases.
- Poor **performance** for high-update transactions.

Diagram:

```sql
`[ Shadow Page Table ]   --> Points to Original Pages
[ Current Page Table ]   --> Points to New Pages (after changes)

On Commit:
Current Page Table → becomes the new database state
`
```

Final Line:

> "Shadow Paging ensures database recovery by safely preserving the original state until the transaction is successfully committed."

Would you like a real-life small example (like editing a Word document and saving after changes) to remember it better? 📄✨

What is deadlock? Explain how deadlock detection and prevention is done.

Here's the **5 to 6 marks** solution you asked for — clean and exam-ready:

Deadlock:

Deadlock occurs in a database when **two or more transactions** are **waiting for each other** to release locks, and **none of them can proceed further**.
It leads to **infinite waiting** and **halts the system**.

✅ **Simple Example:**

- **T1** locks **X**, wants **Y**.
- **T2** locks **Y**, wants **X**.
- Both keep waiting → **Deadlock!**

Deadlock Detection:

In **Deadlock Detection**, the system allows deadlocks to happen but **monitors** for them.

✅ **Steps:**

1. Construct a **Wait-For Graph (WFG)**:
   - Nodes = Transactions

- Edge from T1 → T2 if T1 is waiting for a resource held by T2.
  2. **Cycle Detection**:
     - If there is a **cycle** in the graph → Deadlock exists.
  3. **Resolution**:
     - Abort one or more transactions to break the cycle.

✅ **Diagram:**

```rust

`T1 --> T2 --> T3 --> T1 (Cycle Detected)
`
```

Deadlock Prevention:

In **Deadlock Prevention**, the system tries to **avoid** deadlocks **before they happen** by controlling how transactions acquire locks.

✅ **Common Techniques:**

- **Wait-Die Scheme:**
  - Older transaction waits for younger one.
  - Younger transaction aborts if it needs a lock held by older one.
- **Wound-Wait Scheme:**
  - Older transaction forces younger one to abort (wound).
  - Otherwise, younger transaction waits.

✅ **Key Idea:**

- Transactions are assigned **timestamps**.
- Locking is done based on timestamps to **avoid cycles**.

Quick Table:

Aspect	Deadlock Detection	Deadlock Prevention
When?	After deadlock occurs	Before deadlock occurs
Method	Wait-For Graph & cycle check	Timestamp ordering, protocols
Action	Abort transactions	Avoid blocking/force abort early

Final Line:

"Deadlock detection finds and breaks deadlocks after they occur, while deadlock prevention tries to stop them before happening."

Would you like a quick real-life example (like two cars blocking each other at a narrow road) to understand it even faster? 🚗🤝🚙

👤

Here's your **8 to 9 marks** detailed answer on **Timestamping Methods** — full explanation in simple words:

Timestamping Methods:

Timestamping is a **concurrency control technique** in databases.
Here, every transaction is assigned a unique **timestamp** when it starts.
The timestamp decides the **order** in which transactions are allowed to **read** and **write** the database items.

✅ **Goal:**

- Ensure that transactions **execute in timestamp order**.
- Maintain **serializability** and **avoid conflicts**.

Key Concepts:

- **Timestamp (TS):**
  ➜ A unique number given to each transaction based on **start time** or **counter**.

- **Read Timestamp (RTS) of a data item:**
  ➜ Largest timestamp of a transaction that successfully **read** the data item.

- **Write Timestamp (WTS) of a data item:**
  ➜ Largest timestamp of a transaction that successfully **wrote** the data item.

How Timestamping Works:

When a transaction **T** tries to perform operations:

1. Read Operation (Read(X)):

- If **TS(T) < WTS(X)** →
  ➜ **Reject** the read and **abort** the transaction (because newer value is already written).

- Else →
  ➜ Allow the read and set **RTS(X) = max(RTS(X), TS(T))**.

---

2. Write Operation (Write(X)):

- If **TS(T) < RTS(X)** →
  ➜ **Reject** the write and **abort** the transaction (because another newer transaction already read old value).

- If **TS(T) < WTS(X)** →
  ➜ **Reject** the write (because another newer transaction has already written to X).

- Else →
  ➜ Allow the write and set **WTS(X) = TS(T)**.

Types of Timestamp Ordering:

✅ **1. Basic Timestamp Ordering (TO):**

- All transactions must **obey the timestamp order**.
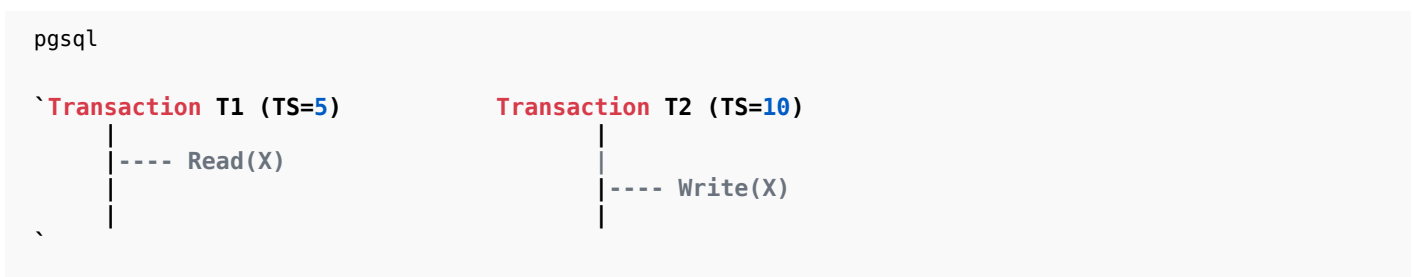- Strict rules based on **RTS** and **WTS**.

✅ **2. Strict Timestamp Ordering (Strict-TO):**

- A transaction can **read** or **write** an item **only after** the last transaction that wrote the item has **committed**.
- Prevents **cascading aborts**.

✅ **3. Thomas's Write Rule:**

- A **relaxation** of basic TO.
- If a transaction tries to write an outdated value (already a newer value exists), the write is **simply ignored**, **not aborted**.

Diagram:

```pgsql
`Transaction T1 (TS=5)        Transaction T2 (TS=10)
      |                              |
      |---- Read(X)                  |
      |                              |---- Write(X)
`     |                              |
```

✅ TS(T1) < TS(T2) ➜ Allowed if timestamp rules are followed.

Advantages of Timestamping:

- No need for locks → No deadlocks.
- Automatically maintains serializability.

Disadvantages:

- **Higher abort rate** if there are many conflicts.
- Starvation possible (younger transactions may be aborted many times).

Final Line:

"Timestamping methods control the order of transactions using timestamps, ensuring consistency without locking but at the cost of more transaction aborts."

Would you also like a small real-world example (like students submitting assignments with submission times)? 📘🚀 (It makes it even easier to understand!)

What are the possible causes of transaction failure? Explain the significance of ACID properties.

Here's your **8 to 9 marks** full answer — clearly structured and exam-perfect:

✅ What are the Possible Causes of Transaction Failure?

A **transaction failure** occurs when a database transaction **cannot complete successfully** and must be **aborted or rolled back**.
Here are the main causes:

---

1. System Crash:

- Due to hardware/software failure or power outage.
- System restarts, and the transaction data in memory may be lost.

---

2. Transaction Error:

- Logical errors in code (e.g., divide by zero, constraint violation).
- User or application mistakes.

---

3. Concurrency Issues:

- Conflicts between multiple transactions running at the same time (e.g., deadlocks, lost updates).

---

4. Disk/Storage Failure:

- Hard disk crash or data corruption causes transaction failure.

---

5. Deadlock Situations:

- Two or more transactions wait forever for each other's resources → system may force abort.

6. Violation of Integrity Constraints:

- If a transaction breaks rules like primary key, foreign key, etc., it must be rolled back.

✅ Significance of ACID Properties:

ACID stands for **Atomicity**, **Consistency**, **Isolation**, and **Durability** —
These are the **core principles** that ensure **reliable and safe transaction processing** in a database.

1. Atomicity:

- Transaction is **all-or-nothing**.
- If any part fails, the **entire transaction is rolled back**.
- Ensures database is not left in partial state.

  Example: If money is debited from one account but not credited to another — rollback both.

2. Consistency:

- Ensures that the database moves from **one valid state to another**.
- All **constraints and rules** are preserved after transaction execution.

  Example: Bank balance should not go negative if the rules disallow it.

3. Isolation:

- Multiple transactions can run **at the same time** without interfering.
- Each transaction should **appear as if it is running alone**.

  Example: Two users booking the same movie seat shouldn't be able to double-book.

4. Durability:

- Once a transaction is **committed**, its changes are **permanent**, even in case of crash or power loss.

  Example: After a ticket booking is confirmed, the booking should not vanish after system restart.

🔚 Final Line:

"ACID properties ensure that the database remains correct, stable, and reliable even in the face of failures, errors, or multiple users accessing it concurrently."

Want a simple table or mnemonic to remember ACID quickly for revision? 🧠✅

Explain log based recovery technique.

Here's your **8 to 9 marks** answer on **Log-Based Recovery Technique** — clear, simple, and detailed for exams:

✅ Log-Based Recovery Technique:

Log-Based Recovery is a method used in **DBMS** to restore the database to a **consistent state** after a crash or failure.

👉 The system keeps a **log file** where all the actions of transactions are **recorded before** they are applied to the database.

🔹 What is a Log?

A **log** is a **sequential record** of all operations (like reads, writes, start, commit, abort) performed by transactions.

Each log entry usually contains:

```pgsql
`<Transaction ID, Data item, Old value, New value>
`
```

✅ Types of Log Entries:

1. **<START T>** → Transaction T has started
2. **<T, X, old_value, new_value>** → T modified X
3. **<COMMIT T>** → T has successfully completed
4. **<ABORT T>** → T has failed and rolled back

🔄 Working of Log-Based Recovery:

🔸 1. Before a Transaction:

- A `<START T>` entry is added to the log.

◆ **2. During Execution:**

- For every `WRITE(X)`, log `<T, X, old, new>` is saved **before** applying the change to the database.

◆ **3. On Commit:**

- A `<COMMIT T>` is written in the log, then changes are made **permanent**.

💥 What Happens on System Crash?

➤ Recovery Process:

1. **Scan the log from the end**:
   - If `<COMMIT T>` is found → **Redo** the changes (to ensure they're in DB).
   - If `<START T>` but no `<COMMIT T>` → **Undo** the changes (transaction not completed).

✅ Two Log-Based Recovery Techniques:

1. Deferred Database Modification:

- Changes are written **only after commit**.
- Easier: only **REDO** needed.
- No actual changes during the transaction.

2. Immediate Database Modification:

- Changes may be applied to the database **before commit**.
- Need both **UNDO and REDO**.
- Requires tracking of **old values** in the log.

📊 Example Log:

```php-template
`<START T1>
<T1, A, 100, 200>
<COMMIT T1>
`
```

➡️ If system crashes after commit → REDO
➡️ If system crashes before commit → UNDO

✅ Advantages:

- Ensures **data integrity and atomicity**

- Handles both committed and incomplete transactions

- Supports **crash recovery**

Final Line:

"Log-based recovery guarantees that even after a system failure, the database can be restored to a consistent state by using the information stored in the log."

Want a simple flow diagram for **UNDO/REDO** after crash? 🧠📈