Transactions Questions

1. Explain the ACID properties of a transaction with the help of a suitable example.
2. Apply the concept of precedence graphs to test whether the given schedule is conflict serializable. Draw the graph and justify your answer.
3. Differentiate between conflict serializability and view serializability. Analyse with suitable examples when a schedule might be view serializable but not conflict serializable.
4. Evaluate the effectiveness of different concurrency control protocols in ensuring serializability and recoverability. Support your answer with reasons.
5. Design a transaction schedule that results in cascading rollback. Explain how it occurs and suggest how it can be avoided.
6. List and describe the different states of a transaction. How does a system transition from one state to another?
7. Summarize the advantages of concurrent transaction execution and explain the challenges it introduces to consistency.
8. Illustrate, with SQL examples, how isolation levels affect transaction behaviour in a multi-user environment.

1. Explain the ACID properties of a transaction with the help of a suitable example.
Answer:
The ACID properties are fundamental principles that ensure reliable processing of database transactions:

1. **Atomicity**:
   A transaction is treated as a single unit which either **completes fully** or **does not execute at all**.
   Example: If money is transferred from Account A to B, and the system crashes after debiting A but before crediting B, the system will roll back to undo the debit.
2. **Consistency**:
   Transactions must transform the database from one **valid state to another**.
   Example: The total balance in the system remains unchanged before and after a transfer.
3. **Isolation**:
   Transactions must execute as if they are the **only transaction** in the system.
   Example: If two transactions access the same data concurrently, the results should be the same as if executed serially.
4. **Durability**:
   Once a transaction **commits**, its changes must be **permanent**, even in case of a crash.
   Example: Once money is transferred and the user is notified, the update persists even after power loss.

2. Apply the concept of precedence graphs to test whether the given schedule is conflict serializable. Draw the graph and justify your answer.
Answer:
Step-by-step Procedure:

1. **Identify conflicting operations**:
   Conflicts occur between Read-Write, Write-Read, or Write-Write on the same data item by different transactions.
2. **Example Schedule**:
   - T1: R(A), W(A)
   - T2: R(A), W(B)

    - T3: R(B), W(C)
3. **Conflicts**:
    - T1 → T2 (W(A) before R(A))
    - T2 → T3 (W(B) before R(B))
4. **Construct Precedence Graph**:
   Nodes: T1, T2, T3
   Edges: T1 → T2 → T3
5. **Check for Cycles**:
   No cycle → **Schedule is conflict serializable**.
6. **Conclusion**:
   Since the graph is acyclic, the schedule can be serialized in the order T1 → T2 → T3.

3. Differentiate between conflict serializability and view serializability. Analyse with suitable examples when a schedule might be view serializable but not conflict serializable.

Answer:

Aspect	**Conflict Serializability**	**View Serializability**
Basis	Swapping non-conflicting operations	Read-from and final-write conditions
Testability	Easy using precedence graph	Difficult (NP-Complete)
Coverage	Subset of view serializable schedules	Superset (includes conflict serializable ones)

Example:
Schedule:

- T1: W(A)
- T2: W(A)
- T3: R(A)
- T3 reads A written by T2. Final write is also by T2.
- This satisfies **view serializability**, but cannot be obtained by swapping non-conflicting operations ⇒ **not conflict serializable**.

4. Evaluate the effectiveness of different concurrency control protocols in ensuring serializability and recoverability. Support your answer with reasons.

Answer:

1. **Two-Phase Locking (2PL)**:
    - **Ensures conflict serializability**

    - Transactions acquire all locks before releasing any.
    - Can lead to **deadlocks**, which need to be detected or prevented.
2. **Strict 2PL**:
    - **Ensures serializability and recoverability**
    - Holds all exclusive locks till commit ⇒ prevents cascading rollbacks.
3. **Timestamp Ordering**:
    - Transactions are ordered using timestamps.
    - Prevents conflicts but may **abort frequently** due to out-of-order access.
4. **Multiversion Concurrency Control (MVCC)**:
    - Ensures **readers do not block writers** and vice versa.
    - Used in PostgreSQL, Oracle.
    - Requires more memory but gives high concurrency.

5. Design a transaction schedule that results in cascading rollback. Explain how it occurs and suggest how it can be avoided.
Answer:
Schedule Example:
T1: W(A)
T2: R(A)
T3: R(A)
T1: abort
Explanation:
- T2 and T3 read data written by T1.
- If T1 aborts, T2 and T3 must also **rollback** because they are based on uncommitted data from T1 ⇒ cascading rollback.

Avoidance:
- Use **cascadeless schedules**:
  Ensure no transaction reads data written by an uncommitted transaction.
- **Strict 2PL**: hold exclusive locks until commit, so no other transaction can read uncommitted data.

6. List and describe the different states of a transaction. How does a system transition from one state to another?
Answer:
States of a Transaction:

1. **Active**: Transaction is executing its operations.
2. **Partially Committed**: Final operation executed; awaiting commit.
3. **Committed**: All operations executed successfully and changes made permanent.
4. **Failed**: An error has occurred; cannot proceed.
5. **Aborted**: Changes rolled back to maintain consistency.

Transitions:

- **Active → Partially Committed**: Last statement executed.
- **Partially Committed → Committed**: Commit is successful.
- **Any state → Failed**: Error or crash occurs.
- **Failed → Aborted**: Rollback is performed.
- **Aborted → Active** (optional): Transaction restarts.

Conclusion:
Transitions ensure transaction integrity and database consistency.

7. Summarize the advantages of concurrent transaction execution and explain the challenges it introduces to consistency.
Answer:
Advantages:

1. **Improved Throughput**: Multiple transactions run in parallel.
2. **Better Resource Utilization**: CPU and I/O overlap.
3. **Reduced Response Time**: Short transactions need not wait for long ones.

Challenges:

1. **Inconsistency**: Simultaneous updates may violate data integrity.
2. **Lost Updates**: Concurrent writes overwrite changes.
3. **Dirty Reads**: A transaction reads uncommitted changes from another.
4. **Unrepeatable Reads**: Data read once may change on next read.
5. **Phantom Reads**: New records appear in repeated queries.

Solution:

- Use **concurrency control protocols** to ensure isolation and consistency.

8. Illustrate, with SQL examples, how isolation levels affect transaction behaviour in a multi-user environment.
Answer:
1. Serializable (Highest isolation)
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

- Prevents dirty reads, non-repeatable reads, and phantom reads.

- Ensures complete isolation.

2. Repeatable Read

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

- Prevents dirty and non-repeatable reads, but not phantom reads.

3. Read Committed

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

- Prevents dirty reads. Allows non-repeatable and phantom reads.

4. Read Uncommitted

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

- Allows dirty reads; lowest consistency but high performance.

Example Scenario:

T1 reads a balance; T2 updates it and commits.

- Under **Serializable**, T1 blocks till T2 finishes.
- Under **Read Committed**, T1 sees the committed update.
- Under **Read Uncommitted**, T1 may see T2’s changes **before commit**.