1. Consider the following transactions with data items P and Q initialized to zero:
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);
Solve and find any non-serial interleaving of T1 and T2 for concurrent execution leads to a serializable
schedule or non-serializable schedule. Explain

To determine if a concurrent execution of transactions T1 and T2 leads to a serializable schedule or a non-serializable schedule, we'll examine the possible interleavings of these transactions. A schedule is considered serializable if its outcome is equivalent to some serial execution of the same transactions. Let's analyze the interleavings of T1 and T2:

T1: read(P); read(Q); *T2:* read(Q); read(P);

Here's a possible interleaving:

1. T1 reads P (P=0).
2. T1 reads Q (since Q is initially zero, Q=0).
3. T1 checks the condition P=0 (which is true) and increments Q by 1.
4. T2 reads Q (Q=1 now).
5. T2 reads P (since P is initially zero, P=0).
6. T2 checks the condition Q=0 (which is false, so this condition doesn't trigger).

Now let's analyze the possible outcomes based on this interleaving:

- After T1 and T2 both finish, Q=1 (incremented by T1) and P=0 (not modified).

This outcome is equivalent to the serial execution of T1 followed by T2. Both T1 and T2 have completed, and the final state is consistent with the effects of the serial execution of T1 followed by T2.

Since this interleaving leads to a schedule that is equivalent to a serial execution, the concurrent execution of T1 and T2 in this particular interleaving is serializable.

It's important to note that this analysis is based on one specific interleaving. To ensure serializability, we need to examine all possible interleavings of the transactions, but in this case, we've shown that at least one interleaving is serializable.

2 Analyze which of the following concurrency control protocols

ensure both conflict serializability and freedom from deadlock?
(a)2 – phase Locking
(b)Time stamp – ordering
A)
(a) z – phase Locking is a concurrency control method that guarantees serializability. The protocol utilizes locks, applied by a transaction to data, which
may block (interpreted as signals to stop) other transactions from accessing the same data.during the transaction's life. 2PL may be lead to deadlocks that result from the mutual blocking of two or more transactions.
(b)Time stamp – ordering concurrency protocol ensures both conflict serializability and freedom from deadlock.
Only (b) is correct.

3 Suppose that we have only two types of transactions, T1 and T2. Transactions preserve database consistency when run individually. We have defined several integrity constraints such that the DBMS never executes any SQL statement that brings the database into an inconsistent state. Assume that the DBMS does not perform any concurrency control. Give an example schedule of two transactions T 1 and T 2 that satisfies all these conditions, yet produces a database instance that is not the result of any serial execution of T 1 and T 2.

To create an example schedule that satisfies the given conditions but leads to a database state that is not achievable through any serial execution of transactions T1 and T2, we need to focus on non-serializable behavior due to lack of concurrency control. Let's construct such a scenario:

Suppose we have a simple database with a single data item X, initially set to 0.

Integrity Constraint: X should always be a non-negative integer.

Now, let's define two transactions, T1 and T2, as follows:

T1: read(X); X := X + 1; write(X); (increments the value of X by 1)
T2: read(X); X := X - 1; write(X); (decrements the value of X by 1)

Both transactions, when executed individually, preserve the integrity constraint since the database starts at 0 and both T1 and T2 only modify X by adding or subtracting 1.

Now, let's consider the following schedule:

1. T1 reads X (X=0)
2. T2 reads X (X=0)
3. T1 increments X by 1 (X=1)
4. T2 decrements X by 1 (X=0)
5. T1 writes the new value of X (X=1)
6. T2 writes the new value of X (X=0)

In this schedule, both T1 and T2 run successfully, and the integrity constraint is maintained throughout the execution of each individual transaction. However, the final database state after the execution of the schedule is X=0, which is not achievable through any serial execution of T1 and T2, as both transactions individually result in a change in the value of X (increment by T1 and decrement by T2).

This scenario demonstrates that without proper concurrency control, it's possible to have a schedule that satisfies integrity constraints individually but doesn't produce a result that can be obtained through a serial execution of the transactions. This highlights the importance of concurrency control mechanisms to ensure serializability and consistent database states in a multi-user environment.

4 Suppose that there is a database system that never fails. Analyze whether a recovery manager required for this system.

In a database system that never fails, the need for a traditional recovery manager is significantly reduced or even eliminated. The primary purpose of a recovery manager in a database system is to handle failures and ensure data consistency and durability in the face of various types of failures, such as hardware crashes, software errors, power outages, etc. Since we are assuming that the database system never fails, the key motivations for having a recovery manager do not apply in this scenario.

A recovery manager typically includes features such as:

1. *Transaction Logging*: Recording changes made by transactions in a log to enable recovery in case of failure.
2. *Checkpoints*: Periodically saving the current state of the database to reduce

the amount of work needed during recovery.
3. *Undo/Redo Mechanisms*: Reverting or applying changes as needed to bring the database back to a consistent state after a failure.

In a system that never fails, these features are not necessary because there are no failures that require recovery. The assumption of a failure-free environment implies that once a transaction is committed, its changes are guaranteed to be durable and permanent, without the risk of being lost due to system crashes.

However, it's important to consider that the real world is rarely perfect, and it's almost impossible to create a truly failure-free system. Even the most robust and reliable systems can face unexpected challenges. Therefore, while a recovery manager might not be required in the same way as in a system with frequent failures, it's still a good practice to have some level of data protection and redundancy to handle unforeseen events.

For example, if the data itself is valuable and needs to be protected against human errors or external threats (e.g., accidental deletions, malicious attacks), having a backup and data protection strategy is essential, even in a system that seldom experiences internal failures.

In summary, while a traditional recovery manager may not be required for a database system that never fails as per the assumption, it's essential to consider other aspects of data management and protection to ensure the security, availability, and integrity of the data in a real-world context.

5 Explain the Immediate database Modification technique for using the Log to Ensure transaction atomicity despite failures?

The Immediate Database Modification (IM) technique is a method used to ensure transaction atomicity despite failures in a database system. This technique leverages transaction logs to achieve this goal. Let's break down how the IM technique works:

1. *Transaction Logs*: In a database system, a transaction log is maintained. This log records all the changes made to the database by various transactions. The log contains a chronological sequence of operations, such as record updates, inserts, and deletes.

2. *Write-Ahead Logging (WAL)*: The IM technique relies on the principle of Write-Ahead Logging. This means that before any actual database modification

occurs, the corresponding log entries for the modification are written to the transaction log. This ensures that the log reflects the intended changes before they are applied to the actual database.

3. *Immediate Modification*: The key aspect of the IM technique is that, for each modification (insert, update, delete) made by a transaction, the modification is applied immediately to the database, but the change is also recorded in the transaction log.

4. *Transaction Commit*: When a transaction is committed, a special log entry is written to indicate the completion and success of the transaction.

5. *Transaction Rollback*: In the case of a transaction failure (e.g., system crash), the transaction manager uses the transaction log to determine the state of the partially executed transaction. It can then decide to rollback the transaction (undo the changes made by the transaction) or continue with recovery procedures to ensure the atomicity, consistency, isolation, and durability (ACID properties) of the transactions.

6. *Recovery*: During system recovery after a failure, the transaction log is used to bring the database to a consistent state. The log is analyzed, and transactions are re-executed or rolled back based on the recorded log entries. This ensures that the database is brought back to a consistent state as if the failure never occurred.

The IM technique, with its use of immediate modification and careful logging, ensures that even if a failure occurs, the atomicity of transactions is maintained. The transaction log acts as a record of all changes, allowing the system to recover and bring the database to a consistent state.

It's important to note that the IM technique, while providing transaction atomicity, requires careful management of the transaction log and proper recovery mechanisms to handle failures and maintain the overall integrity of the database system.

6. Consider the following actions taken by transaction T1 on database objects X and Y : R(X), W(X), R(Y), W(Y) Give an example of another transaction T 2 that, if run concurrently to transaction T without some form of concurrency control, could interfere with T 1. 1. Explain how the

use of Strict 2PL would prevent interference between the two transactions. 2. Strict 2PL is used in many database systems. Give two reasons for its popularity.

Let's start by analyzing the actions taken by transaction T1 on database objects X and Y:

1. R(X): T1 reads the value of database object X.
2. W(X): T1 writes a new value to database object X.
3. R(Y): T1 reads the value of database object Y.
4. W(Y): T1 writes a new value to database object Y.

To demonstrate interference with T1, let's consider another transaction T2:

T2: R(Y), W(X)

Now, let's explore how T2 could interfere with T1 if run concurrently without some form of concurrency control:

1. T1 performs R(Y), reading the value of database object Y.
2. T2 performs R(Y), reading the value of Y as well, and doesn't modify Y.
3. T1 continues with W(Y), writing a new value to Y.
4. T2 performs W(X), writing a new value to database object X.

The interference occurs in this scenario because T2 has read Y before T1 modified it, and T2's write to X could be based on the original value of Y, not the updated value that T1 wrote. This could lead to an inconsistent state if the values of X and Y are related in some way.

Now, let's discuss how the use of Strict Two-Phase Locking (Strict 2PL) would prevent interference between the two transactions:

1. **Locks**: Strict 2PL requires that a transaction acquires all the locks it needs before it starts executing. In this case, T1 would acquire a lock on both X and Y before it starts executing. Similarly, T2 would need to acquire a lock on Y before it starts executing.

2. **Release Locks**: Strict 2PL also ensures that locks are released only after the transaction has completed, i.e., after the transaction has released all its locks, it cannot acquire any new locks. This ensures that no transaction reads a value that another transaction intends to modify.

Applying Strict 2PL to the scenario described above:

1. T1 acquires locks on both X and Y before it starts executing.
2. T2, if it attempts to run concurrently, cannot acquire a lock on Y because T1 already has the lock on it. Therefore, T2 must wait until T1 releases the lock on Y.

The use of Strict 2PL prevents T2 from reading the value of Y before T1 writes to it, thus ensuring that T2 does not interfere with T1.

Reasons for the popularity of Strict 2PL in many database systems:

1. **Simplicity**: Strict 2PL is relatively easy to implement and reason about. It provides a clear structure for managing locks, making it a practical choice for many database systems.
2. **Deadlock Avoidance**: Strict 2PL, when combined with proper deadlock detection and prevention mechanisms, can effectively avoid deadlocks. This is crucial for maintaining system availability and preventing transaction stalls due to deadlocks.

7. Suppliers(sid: integer, sname:string, address: string) Parts(pid: integer, pname:string, color: string) Catalog(sid: integer, pid:integer, cost: real) The Catalog relation lists the prices charged for parts by Suppliers. For each of the following transactions, state the SQL isolation level that you would use and explain why you chose it. 1. A transaction that adds a new part to a suppliers catalog. 2. A transaction that increases the price that a supplier charges for a part.

The choice of SQL isolation level depends on the specific requirements of each transaction and the desired trade-off between concurrency and data consistency. SQL provides several isolation levels to handle concurrent transactions with varying degrees of data visibility and consistency. Let's analyze each transaction and choose an appropriate isolation level for each:

1. **A transaction that adds a new part to a supplier's catalog**:

For this type of transaction, the main concern is ensuring that the new part is properly added to the catalog without interfering with other transactions that might be accessing the same catalog. It's essential to prevent other transactions

from seeing the new part in an incomplete or inconsistent state.

Isolation Level: SERIALIZABLE

Explanation: The SERIALIZABLE isolation level ensures the highest level of data consistency by preventing other transactions from accessing the catalog table until the current transaction is complete. This prevents any anomalies, such as dirty reads, unrepeatable reads, or phantom reads, from occurring while the new part is being added. Since adding a new part involves multiple steps (inserting into the Catalog table), this isolation level ensures that the new part becomes visible to other transactions only when the transaction is fully committed, thus maintaining data integrity.

2. **A transaction that increases the price that a supplier charges for a part**:

For this type of transaction, the main concern is ensuring that the price update is applied consistently and does not conflict with other transactions that might be reading or updating the same catalog.

Isolation Level: READ COMMITTED

Explanation: The READ COMMITTED isolation level ensures that each read operation within a transaction sees only committed data. This prevents dirty reads, meaning that if another transaction is updating the price of a part, the read operation in this transaction won't see the intermediate, uncommitted state. However, it allows other transactions to see the new price after the current transaction has committed. This level of isolation provides a good balance between concurrency and data consistency. Since changing the price of a part is a straightforward update, this level of isolation works well, allowing other transactions to read the updated price once the updating transaction is committed, without waiting for the entire transaction to complete as in the SERIALIZABLE level.

In summary, the choice of isolation level depends on the nature of the transaction, the desired level of data consistency, and the potential for conflicts with other transactions accessing the same data. SERIALIZABLE ensures the highest consistency but may lead to more contention for resources, while READ COMMITTED allows for a higher degree of concurrency but still ensures that transactions see only committed data.

8 Answer each of the following questions briefly. The questions are based on the following relational schema: Emp(eid:integer, ename: string,

age:integer, salary: real, did: integer) Dept(did: integer,dname: string, floor: integer) and on the following update command: replace (salary = 1.1 * EMP.salary) where EMP.ename = Santa 1. Give an example of a query that would conflict with this command (in a concurrency control sense) if both were run at the same time. 2. Explain what could go wrong, and how locking tuples would solve the problem. 3.Give an example of a query or a command that would conflict with this command, such that the conflict could not be resolved by just locking individual tuples or pages but requires index locking.

Sure, I'll address each question briefly:

1. **Example of a conflicting query**: Suppose we have another query that calculates the total salary of employees in a specific department while the update command is running: `SELECT SUM(salary) FROM Emp WHERE did = 123;`

2. **What could go wrong and how locking tuples would solve the problem**: Without proper concurrency control, the conflicting query in (1) might read some of the old salary values before the update command is completed, leading to an inconsistency. To solve this, locking the tuples that are being updated in the `replace` command would ensure that the conflicting query waits until the update command is finished before reading the updated salary values, maintaining consistency.

3. **Example of a query/command requiring index locking**: Suppose we have a command that deletes all employees in a department with a specific name (e.g., "Marketing") while the update command is running: `DELETE FROM Emp WHERE did IN (SELECT did FROM Dept WHERE dname = 'Marketing');` In this case, to prevent the possibility of new employees with the name "Santa" being added while the delete command is running (since the index on `ename` might be modified), index locking on the `ename` index would be necessary to ensure proper serialization and consistency.

10.What are the roles of the Analysis, Redo, and Undo phases in ARIES?

ANALYSIS PHASE :-
- The dirty page table has been formed by analysing the pages from the buffer and

also a set of active transactions has been identified. When the system

encounters crash, ARIES recovery manager starts the analysis phone by finding the last checkpoint log record after that, it prepares dirty pages tables this phone mainly prepares a set of active transactions that are needed to be undo analysis phone, after getting the last checkpoint log record, log record is scanned in forward direction, & update of the set of active transactions, transaction value, &
dirty page table are done in the following manner :-

- 1st recovery manager finds, any transaction which is not in the active transaction
set, then add that transaction in that set if it finds an end log record, then that record has been deleted from the transaction table.
- If it finds a log record that describes an update operation, the log record has been added to the dirty page table.

REDO PHASE:-

- Redo phase is the second phase where all the transactions that are needed to be
executed again take place.
- It executes those operations whose results are not reflected in the disk.
- It can be done by finding the smallest LSN of all the dirty page in dirty page table
that defines the log positions, & the Redo operation will start from this position
- This position indicates that either the changes that are made earlier are I the main memory or they have already been flaunted to the disk.
- Thus, for each change recorded in the log, the Redo phase determines whether or not the operations have been re-executed.

UNDO PHASE:-

- In the Undo phase, all the transaction, that is listed in the active transaction set here to be undone.

- Thus the log should be scanned background from the end & the recovery manages should Undo the necessary operations.
- Each time an operations is undone, a compensation log recorded has been written to the log.
- This process continues until there is no transaction left in the active transaction
set.
- After the successful competition of this phase, database can resume its normal operations.