UNIT-4

a) Define serializability ? Serializability is the property that ensures that the concurrent execution of a set of transactions produces the same result as if these transactions were executed one after the other without any overlapping, i.e., serially.	**1M**
b) What is locking Protocol? Locking protocol in a Database Management System (DBMS) is all about controlling how multiple transactions can access and manipulate the data concurrently. Basically, when one transaction wants to read or modify a data item, it needs to lock it so that no other transaction can do so at the same time.	**1M**
c) Define multiple granularity Multiple granularity is all about enhancing concurrency control in a database system by allowing transactions to lock data items at different levels of granularity, or detail. Imagine a hierarchy, with the entire database at the top, and individual data items at the bottom. Locks can be placed at any level—database, table, page, or even a single row.	**1M**
d) Write about transaction states 1. **Active**: Transaction starts and operations are ongoing. 2. **Partially Committed**: All operations done, but changes not yet saved. 3. **Committed**: Changes are now permanently saved in the database. 4. **Failed**: Something went wrong; transaction can't be completed. 5. **Aborted**: Changes are rolled back to maintain consistency. 6. **Terminated**: Transaction lifecycle ends, either successful or rolled back. These states are crucial for keeping data consistent and reliable.	**1M**
e) What is the motivation for concurrent execution? Concurrent execution in DBMS is all about maximizing efficiency and system utilization. 1. **Improved System Throughput** 2. **Reduced Waiting Time** 3. **Resource Utilization**: 4. **Responsiveness**	**1M**

5. **Fairness**	
a) Explain the ACID Properties of transaction with examples.? A transaction is a single, logical unit of work that consists of one or more related tasks. A transaction is treated as a single, indivisible operation, which means that either all the tasks within the transaction are executed successfully, or none are. Here are a few key properties of transactions, often referred to by the acronym **ACID**: **ACID Properties** 1. **Atomicity** 2. **Consistency** 3. **Isolation** 4. **Durability** **Atomicity:** This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There is no such thing as partial transaction; if a transaction fails, all the changes made in the database so far by that transaction are rolled back, and the database remains unchanged. **Consistency:** The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before a transaction, then after execution of the transaction, the database must be back to its consistent state. **Isolation:** Each transaction is considered independent of other transactions. That is, the execution of multiple transactions concurrently will have the same effect as if they had been executed one after the other. But isolation does not ensure which transaction will execute first. **Durability:** The changes made to the database by a successful transaction persist even after a system failure.	**3M**
b) What is conflict serializable schedule? **Conflict Serializability** Conflict serializability is a form of serializability where the order of non-conflicting operations is not significant. It determines if the concurrent execution of several transactions is equivalent to some serial execution of those transactions. Two operations are said to be in conflict if: • They belong to different transactions. • They access the same data item. • At least one of them is a write operation.	**3M**

Examples of *conflicting* operations

```
   T1      |    T2
-----------|----------
Read(A)    |  Write(A)
Write(A)   |  Read(A)
Write(A)   |  Write(A)
```

A schedule is conflict serializable if it can be transformed into a serial schedule (i.e., a schedule with no overlapping transactions) by swapping non-conflicting operations. If it is not possible to transform a given schedule to any serial schedule using swaps of non-conflicting operations, then the schedule is not conflict serializable.

To determine if S is conflict serializable:

Precedence Graph (Serialization Graph): Create a graph where:

Nodes represent transactions.

Draw an edge from (T_i) to (T_j) if an operation in (T_i) precedes and conflicts with an operation in (T_j).

For the given example:

```
   T1      |    T2
-----------|----------
Read(A)    |
           |  Read(A)
Write(A)   |
           |  Read(B)
           |  Write(B)
```

(R1(A)) conflicts with W1(A), so there's an edge from T1 to T1, but this is ignored because they´re from the same transaction.
R2(A) conflicts with W1(A), so there's an edge from T2 to T1 (T2 -> T1).
No other conflicting pairs.

The graph has nodes T1 and T2 with an edge from T2 to T1. There are no cycles in this graph.

Decision: Since the precedence graph doesn't have any cycles,Cycle is a path using which we can start from one node and reach to the same node. the schedule S is conflict serializable. The equivalent serial schedules, based on the graph, would be T2 followed by T1.

3M

c) What is a deadlock?

Deadlocks in DBMS happen when two or more transactions are waiting for each other to release resources, creating a cycle of dependency with no way out. Think of it like a traffic jam where two

cars block each other because neither wants to back up.

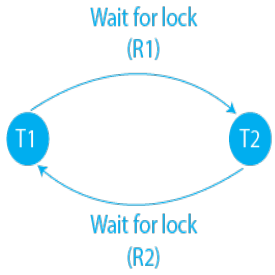
Imagine two transactions, T1 and T2:

- T1 locks Resource A and needs Resource B to proceed.
- T2 locks Resource B and needs Resource A to proceed.

T1 waits for T2 to release Resource B, while T2 waits for T1 to release Resource A. Both are stuck, creating a deadlock.

Consider two friends, Alice and Bob, trying to withdraw money:

- Alice locks Account X and waits for Account Y to become available.
- Bob locks Account Y and waits for Account X to become available.



Techniques used to avoid deadlocks in DBMS:

1. **Deadlock Prevention**:
    - Wait-Die Scheme
    - Wound-Wait Scheme
2. **Deadlock Avoidance**:
    - Banker's Algorithm
3. **Deadlock Detection and Recovery**:
    - Resource Allocation Graphs
    - Rollback
4. **Timeouts**

Each technique comes with its own set of advantages and trade-offs

d) Write about Thomas write rule. Thomas's Write Rule (TWR) is a clever technique used to optimize concurrency control in databases. Its main goal is to prevent unnecessary rollbacks of transactions, thereby improving system performance while ensuring data consistency. Here's how it works: • When a transaction attempts to write to a data item, TWR checks the transaction's timestamp. • If the timestamp of the transaction trying to write is **less** than the timestamp of the last committed transaction that wrote to the same data item, the write operation is ignored. **Example**: Consider two transactions, T1 and T2: • T1 has a timestamp of 15. • T2 has a timestamp of 25. If T1 tries to write to a data item that T2 has already written to and committed, TWR dictates that T1's write should be ignored because T1's timestamp (15) is less than T2's timestamp (25). This ensures that the most recent (and presumably most correct) data is preserved, avoiding any overwrite by an older transaction. **Why it's useful**: • **Performance**: By ignoring unnecessary writes, TWR reduces system overhead and improves transaction throughput. • **Consistency**: Ensures that only the most up-to-date information is kept, maintaining data integrity.	**3M**
e) What is recoverability? Recoverability refers to the ability of a system to restore its state to a point where the integrity of its data is not compromised, especially after a failure or an error. When multiple transactions are executing concurrently, issues may arise that affect the system's recoverability. The interaction between transactions, if not managed correctly, can result in scenarios where a transaction's effects cannot be undone, which would violate the system's integrity. **Importance of Recoverability:** The need for recoverability arises because databases are designed to ensure data reliability and consistency. If a system isn't recoverable: • The integrity of the data might be compromised. • Business processes can be adversely affected due to corrupted or inconsistent data. • The trust of end-users or businesses relying on the database will be diminished.	**3M**

Levels of Recoverability

1. Recoverable Schedules

A schedule is said to be recoverable if, for any pair of transactions (T_i) and (T_j), if (T_j) reads a data item previously written by (T_i), then (T_i) must commit before (T_j) commits. If a transaction fails for any reason and needs to be rolled back, the system can recover without having to rollback other transactions that have read or used data written by the failed transaction.

Example of a Recoverable Schedule

Suppose we have two transactions (T_1) and (T_2).

Transaction T1	Transaction T_2
Write(A)	
	Read(A)
Commit	
	Write(B)
	Commit

In the above schedule, (T_2) reads a value written by (T_1), but (T_1) commits before (T_2), making the schedule recoverable.

a) Explain about Validation -Based Protocol.?

5M

Validation-based protocols, also known as Optimistic Concurrency Control (OCC), are a set of techniques that aim to increase system concurrency and performance by assuming that conflicts between transactions will be rare. Unlike other concurrency control methods, which try to prevent conflicts proactively using locks or timestamps, OCC checks for conflicts only at transaction commit time.

Here's how a typical validation-based protocol operates:

1. **Read Phase**
   - The transaction reads from the database but does not write to it.
   - All updates are made to a local copy of the data items.
2. **Validation Phase**
   - Before committing, the system checks to ensure that this transaction's local updates won't cause conflicts with other transactions.
   - The validation can take many forms, depending on the specific protocol.
3. **Write Phase**
   - If the transaction passes validation, its updates are applied to the database.
   - If it doesn't pass validation, the transaction is aborted and restarted.

Validation-based protocols Example

Let's consider a simple scenario with two transactions (T1) and (T2):

- Both transactions read an item (A) with a value of 10.
- (T1) updates its local copy of (A) to 12.
- (T2) updates its local copy of (A) to 15.

- (T1) reaches the validation phase and is validated successfully (because there's no other transaction that has written to (A) since (T1) read it).
- (T1) moves to the write phase and updates (A) in the database to 12.
- (T2) reaches the validation phase. The system realizes that since (T2) read item (A), another transaction (i.e., (T1)) has written to (A). Therefore, (T2) fails validation.
- (T2) is aborted and can be restarted.

Advantages of Validation-based protocols

- **High Concurrency:** Since transactions don't acquire locks, more than one transaction can process the same data item concurrently.
- **Deadlock-free:** Absence of locks means there's no deadlock scenario.

Disadvantages of Validation-based protocols

- **Overhead:** The validation process requires additional processing overhead.
- **Aborts:** Transactions might be aborted even if there's no real conflict, leading to wasted processing.

Validation-based protocols work best in scenarios where conflicts are rare. If the system anticipates many conflicts, then the high rate of transaction restarts might offset the advantages of increased concurrency.

b) Explain ensuring atomicity and durability properties for a transaction by DBMS? **5M**

Implementation Atomicity

- When a transaction starts, changes are not immediately written to the original data pages. Instead, new copies of the data pages are created and modified. The original data remains untouched.
- If the transaction fails or needs to be aborted for some reason, the DBMS simply discards the new pages. Since the original data hasn't been altered, atomicity is maintained. No changes from the failed transaction are visible.

Implementation Durability

- Once a transaction is completed successfully, the directory that was pointing to the original pages is updated to point to the modified pages. This switch can be done atomically, ensuring durability.
- After the switch, the new pages effectively become the current data, while the old pages can be discarded or archived.
- Because the actual switching of the pointers is a very fast operation, committing a transaction can be done quickly, ensuring that once a transaction is declared complete, its changes are durable.

Shadow Database scheme

The Shadow Database scheme is a technique used for ensuring atomicity and durability in a database system, two of the ACID properties. The core idea is to keep a shadow copy of the database and make changes to a separate copy. Once the transaction is complete and it's time to commit, the

system switches to the new copy, effectively making it the current database. This allows for easy recovery and ensures data integrity even in the case of system failures.

Working Principle of Shadow Database

- **Initial State:** Initially, the database is in a consistent state. Let's call this the "Shadow" database.
- **Transaction Execution:** When transactions are executed, they are not applied directly to the shadow database. Instead, they are applied to a separate copy of the database.
- **Commit:** After a transaction is completed successfully, the system makes the separate copy the new shadow database.
- **Rollback:** If a transaction cannot be completed successfully, the system discards the separate copy, effectively rolling back all changes.
- **System Failure:** If a system failure occurs in the middle of a transaction, the original shadow database remains untouched and in a consistent state.

Shadow Database scheme Example

Let's consider a simple banking database that has one table named **`Account`** with two fields: **`AccountID`** and **`Balance`**.

Shadow Database - Version 1

AccountID	Balance
1	1000
2	2000

1. Transaction Start: A transaction starts to transfer 200 from AccountID 1 to AccountID 2.

2. Separate Copy: A separate copy of the database is made and the changes are applied to it.

Modified Copy

AccountID	Balance
1	800
2	2200

3. Commit: The transaction completes successfully. The modified copy becomes the new shadow database.

Shadow Database - Version 2

AccountID	Balance
1	800
2	2200

4. System Failure: If the system crashes after this point, the shadow database (Version 2) is already in a consistent state, ensuring durability and atomicity.

Shadow Database scheme Pros

- **Atomicity:** All changes are either committed entirely or not at all.
- **Durability:** Once changes are committed, they are permanent, even in the case of system failures.

Shadow Database scheme Cons

- **Disk I/O:** Requires more disk operations because of the separate copy.
- **Concurrency:** More complex to implement in a multi-user environment.

Shadow databases are particularly useful for systems where atomicity and durability are more critical than performance, such as in financial or medical databases.

c) Compile the compatibility matrix for multiple granularity schemes.? **5M**

In the context of database systems, **"granularity"** refers to the size or extent of a data item that can be locked by a transaction. The idea behind multiple granularity is to provide a hierarchical structure that allows locks at various levels, ranging from coarse-grained (like an entire table) to fine-grained (like a single row or even a single field). This hierarchy offers flexibility in achieving the right balance between concurrency and data integrity.

The concept of multiple granularity can be visualized as a tree. Consider a database system where:

- The entire database can be locked.
- Within the database, individual tables can be locked.
- Within a table, individual pages or rows can be locked.
- Even within a row, individual fields can be locked.

Lock Modes in multiple granularity

To ensure the consistency and correctness of a system that allows multiple granularity, it's crucial to introduce the concept of "intention locks." These locks indicate a transaction's intention to acquire a finer-grained lock in the future.

There are three main types of intention locks:

1. Intention Shared (IS): When a Transaction needs S lock on a node "K", the transaction would need to apply IS lock on all the precedent nodes of "K", starting from the root node. So, when a node is found locked in IS mode, it indicates that some of its descendent nodes must be locked in S mode.

Example: Suppose a transaction wants to read a few records from a table but not the whole table. It might set an IS lock on the table, and then set individual S locks on the specific rows it reads.

2. Intention Exclusive (IX): When a Transaction needs X lock on a node "K", the transation would need apply IX lock on all the precedent nodes of "K", starting from the root node. So, when a node is found locked in IX mode, it indicates that some of its descendent nodes must be locked in X mode.

Example: If a transaction aims to update certain records within a table, it may set an IX lock on the table and subsequently set X locks on specific rows it updates.

3. Shared Intention Exclusive (SIX): When a node is locked in SIX mode; it indicates that the node is explicitly locked in S mode and Ix mode. So, the entire tree rooted by that node is locked in S

mode and some nodes in that are locked in X mode. This mode is compatible only with IS mode.

Example: Suppose a transaction wants to read an entire table but also update certain rows. It would set a SIX lock on the table. This tells other transactions they can read the table but cannot update it until the SIX lock is released. Meanwhile, the original transaction can set X locks on specific rows it wishes to update.

Compatibility Matrix with Lock Modes in multiple granularity

A compatibility matrix defines which types of locks can be held simultaneously on a database object. Here's a simplified matrix:

	NL	IS	IX	S	SIX	X
NL	✓	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	✓	✗
IX	✓	✓	✓	✗	✗	✗
S	✓	✓	✗	✓	✗	✗
SIX	✓	✓	✗	✗	✗	✗
X	✓	✗	✗	✗	✗	✗

(NL = No Lock, S = Shared, X = Exclusive)

The Scheme operates as follows:-

a. A Transaction must first lock the Root Node and it can be locked in any mode.
b. Locks are granted as per the Compatibility Matrix indicated above.
c. A Transaction can lock a node in S or IS mode if it has already locked all the predecessor nodes in IS or IX mode.
d. A Transaction can lock a node in X or IX or SIX mode if it has already locked all the predecessor nodes in SIX or IX mode.
e. A transaction must follow two-phase locking. It can lock a node, only if it has not previously unlocked a node. Thus, schedules will always be conflict-serializable.
f. Before it unlocks a node, a Transaction has to first unlock all the children nodes of that node. Thus, locking will proceed in top-down manner and unlocking will proceed in bottom-up manner. This will ensure the resulting schedules to be deadlock-free.

Benefits of using multiple granularity

- **Flexibility:** Offers flexibility to transactions in deciding the appropriate level of locking, which can lead to improved concurrency.
- **Performance:** Reduces contention by allowing transactions to lock only those parts of the data that they need.

d) Explain about Concurrent execution of transactions. **5M**

Concurrent execution refers to the **simultaneous execution** of more than one transaction. This is a common scenario in multi-user database environments where many users or applications might be accessing or modifying the database at the same time. Concurrent execution is crucial for achieving high throughput and efficient resource utilization. However, it introduces the potential for conflicts

and data inconsistencies.

Advantages of Concurrent Execution

1. **Increased System Throughput:** Multiple transactions can be in progress at the same time, but at different stages
2. **Maximized Processor Utilization:** If one transaction is waiting for I/O operations, another transaction can utilize the processor.
3. **Decreased Wait Time:** Transactions no longer have to wait for other long transactions to complete.
4. **Improved Transaction Response Time:** Transactions get processed faster because they can be executed in parallel.

Potential Problems with Concurrent Execution

1. Lost Update Problem (Write-Write conflict):

Lost Update Problem: When two transactions update the same data item simultaneously, one update might overwrite the other, leading to data loss.

How It Happens:

1. **Concurrent Transactions**: Two or more transactions read the same data item.
2. **Independent Updates**: Each transaction makes changes based on the initial read value.
3. **Overwriting**: The final write operation from one transaction overwrites the changes made by the other transaction(s).

Example Scenario:

Consider two transactions, T1 and T2, both trying to update the balance of a bank account:

- **T1** reads the balance as $1000.
- **T2** also reads the balance as $1000.
- **T1** subtracts $200 and writes back $800.
- **T2** adds $300 and writes back $1300.

In this case, the update from T1 is lost because T2's update overwrites it, resulting in an incorrect final balance.

Examples:

```
   T1      |    T2
-----------|-----------
Read(A)    |
A = A+50   |
           |  Read(A)
           |  A = A+100
Write(A)   |
           |  Write(A)
```

Result: T1's updates are lost.

2.Temporary Inconsistency or Dirty Read Problem (Write-Read conflict):

Dirty Read Problem: A transaction reads data that has been modified by another transaction but not yet committed, leading to inconsistencies if the other transaction is rolled back.

How It Happens:

1. **Transaction T1** starts and updates a data item.
2. **Transaction T2** reads the updated data item before T1 commits.
3. **Transaction T1** rolls back, undoing its changes.
4. **Transaction T2** now has read data that never actually existed in the committed state.

Example Scenario:

Consider two transactions, T1 and T2:

- **T1** updates the balance of an account from $1000 to $1200 but hasn't committed yet.
- **T2** reads the balance as $1200.
- **T1** rolls back, reverting the balance to $1000.
- **T2** has read a balance of $1200, which is incorrect because T1's update was not committed.

Examples:

```
   T1               |    T2
--------------------|-----------
Read(A)             |
A = A+50            |
Write(A)            |
                    |  Read(A)
                    |  A = A+100
                    |  Write(A)
Read(A)(rollbacks)  |
                    |  commit
```

Result: T2 has a "dirty" value, that was never committed in T1 and doesn't actually exist in the database.

3. Unrepeatable Read Problem (Read-Write conflict):

when a single transaction reads the same row multiple times and observes different values each time. This occurs because another concurrent transaction has modified the row between the two reads.

Examples:

```
   T1      |    T2
-----------|----------
Read(A)    |
           |  Read(A)
           |  A = A+100
           |  Write(A)
Read(A)    |
```

Result: Within the same transaction, T1 has read two different values for the same data item. This inconsistency is the unrepeatable read.

Phantom Read Problem: A transaction reads a set of rows that satisfy a condition, but another transaction inserts or deletes rows that satisfy the same condition, causing the first transaction to see different sets of rows in subsequent reads.

Deadlock: Two or more transactions are waiting for each other to release resources, causing all of them to be blocked indefinitely.

Starvation: A transaction is perpetually delayed because other transactions are continuously given preference

To manage concurrent execution and ensure the consistency and reliability of the database, DBMSs use concurrency control techniques. These typically include locking mechanisms, timestamps, optimistic concurrency control, and serializability checks.

e) Explain about Strict Two-Phase Locking (Strict 2PL) in detail? **5M**

The two-phase locking protocol ensures a transaction gets all the locks it needs before it releases any. The protocol has two phases:

- **Growing Phase:** The transaction may obtain any number of locks but cannot release any.
- **Shrinking Phase:** The transaction may release but cannot obtain any new locks.

The point where the transaction releases its first lock is the end of the growing phase and the beginning of the shrinking phase.

This protocol ensures conflict-serializability and avoids many of the concurrency issues, but it doesn't guarantee the absence of deadlocks.

```
   T1      |    T2
-----------|----------
Lock-S(A)  |
           |
           |  Lock-S(A)
Lock-X(B)  |
Unlock(A)  |
           |  Lock-X(C)
Unlock(B)  |
           |  Unlock(A)
```

```
        |  Unlock(C)
```

Pros of Two-Phase Locking (2PL)

- **Ensures Serializability:** 2PL guarantees conflict-serializability, ensuring the consistency of the database.
- **Concurrency:** By allowing multiple transactions to acquire locks and release them, 2PL increases the concurrency level, leading to better system throughput and overall performance.
- **Avoids Cascading Rollbacks:** Since a transaction cannot read a value modified by another uncommitted transaction, cascading rollbacks are avoided, making recovery simpler.

Cons of Two-Phase Locking (2PL)

- **Deadlocks:** The main disadvantage of 2PL is that it can lead to deadlocks, where two or more transactions wait indefinitely for a resource locked by the other.
- **Reduced Concurrency (in certain cases):** Locking can block transactions, which can reduce concurrency. For example, if one transaction holds a lock for a long time, other transactions needing that lock will be blocked.
- **Overhead:** Maintaining locks, especially in systems with a large number of items and transactions, requires overhead. There's a time cost associated with acquiring and releasing locks, and memory overhead for maintaining the lock table.
- **Starvation:** It's possible for some transactions to get repeatedly delayed if other transactions are continually requesting and acquiring locks.

Categories of Two-Phase Locking in DBMS

1. **Strict Two-Phase Locking**
2. **Rigorous Two-Phase Locking**
3. **Conservative (or Static) Two-Phase Locking:**

Strict Two-Phase Locking

- Transactions are not allowed to release any locks until after they commit or abort.
- Ensures serializability and avoids the problem of cascading rollbacks.
- However, it can reduce concurrency.

Rigorous Two-Phase Locking

- A transaction can release a lock after using it, but it cannot commit until all locks have been acquired.
- Like strict 2PL, rigorous 2PL is deadlock-free and ensures serializability.

Conservative or Static Two-Phase Locking

- A transaction must request all the locks it will ever need before it begins execution. If any of the requested locks are unavailable, the transaction is delayed until they are all available.
- This approach can avoid deadlocks since transactions only start when all their required locks are available.

a) What is meant by concurrency control? Explain in detail.	**10M**

To manage concurrent execution and ensure the consistency and reliability of the database, DBMSs use concurrency control techniques. These typically include locking mechanisms, timestamps, optimistic concurrency control, and serializability checks.

1. Locking Mechanisms

Locking ensures exclusive access to a data item for a transaction. This means that while one transaction holds a lock on a data item, no other transaction can access that item.

- **Shared Lock (S-lock):** Allows a transaction to read an item but not write to it.
- **Exclusive Lock (X-lock):** Allows a transaction to read and write an item. No other transaction can read or write until the lock is released.
- **Two-phase Locking (2PL):** This protocol ensures that a transaction acquires all the locks before it releases any. This results in a growing phase (acquiring locks and not releasing any) and a shrinking phase (releasing locks and not acquiring any).

2. Timestamp-based protocols are concurrency control mechanisms used in databases to ensure serializability and to avoid conflicts without the need for locking. The main idea behind these protocols is to use a timestamp to determine the order in which transactions should be executed. Each transaction is assigned a unique timestamp when it starts.

Here are the primary rules associated with timestamp-based protocols:

1. Timestamp Assignment: When a transaction (T) starts, it is given a timestamp (TS(T)). This timestamp can be the system's clock time or a logical counter that increments with each new transaction.

2. Reading/Writing Rules: Timestamp-based protocols use the following rules to determine if a transaction can read or write an item:

- **Read Rule:** If a transaction (T) wants to read an item that was last written by transaction (T') with (TS(T') > TS(T)), the read is rejected because it's trying to read a value from the future. Otherwise, it can read the item.
- **Write Rule:** If a transaction (T) wants to write an item that has been read or written by a transaction (T') with (TS(T') > TS(T)), the write is rejected. This avoids overwriting a value that has been read or written by a younger transaction.

3. Handling Violations: When a transaction's read or write operation violates the rules, the transaction can be rolled back and restarted or aborted, depending on the specific protocol in use.

3. Validation-based protocols

Validation-based protocols, also known as Optimistic Concurrency Control (OCC), are a set of techniques that aim to increase system concurrency and performance by assuming that

conflicts between transactions will be rare. Unlike other concurrency control methods, which try to prevent conflicts proactively using locks or timestamps, OCC checks for conflicts only at transaction commit time.

Here’s how a typical validation-based protocol operates:

1. **Read Phase**
    - The transaction reads from the database but does not write to it.
    - All updates are made to a local copy of the data items.
2. **Validation Phase**
    - Before committing, the system checks to ensure that this transaction's local updates won't cause conflicts with other transactions.
    - The validation can take many forms, depending on the specific protocol.
3. **Write Phase**
    - If the transaction passes validation, its updates are applied to the database.
    - If it doesn't pass validation, the transaction is aborted and restarted.

4. Serializability

Serializability ensures that even when transactions are executed concurrently, the database remains consistent, producing a result that's equivalent to a serial execution of these transactions.

Testing for serializability in DBMS

Testing for serializability in a DBMS involves verifying if the interleaved execution of transactions maintains the consistency of the database. The most common way to test for serializability is using a precedence graph (also known as a serializability graph or conflict graph).

Types of Serializability

1. **Conflict Serializability**
2. **View Serializability**

Conflict Serializability

Conflict serializability is a form of serializability where the order of non-conflicting operations is not significant. It determines if the concurrent execution of several transactions is equivalent to some serial execution of those transactions.

Two operations are said to be in conflict if:

- They belong to different transactions.
- They access the same data item.
- At least one of them is a write operation.

View Serializability

View Serializability is one of the types of serializability in DBMS that ensures the

consistency of a database schedule. Unlike conflict serializability, which cares about the order of conflicting operations, view serializability only cares about the final outcome.

That is, two schedules are view equivalent if they have:

- **Initial Read:** The same set of initial reads (i.e., a read by a transaction with no preceding write by another transaction on the same data item).
- **Updated Read:** For any other writes on a data item in between, if a transaction (T_j) reads the result of a write by transaction (T_i) in one schedule, then (T_j) should read the result of a write by (T_i) in the other schedule as well.
- **Final Write:** The same set of final writes (i.e., a write by a transaction with no subsequent writes by another transaction on the same data item).

b) What is transaction? Explain the properties of transaction. **10M**

A transaction is a single, logical unit of work that consists of one or more related tasks. A transaction is treated as a single, indivisible operation, which means that either all the tasks within the transaction are executed successfully, or none are.

Here are a few key properties of transactions, often referred to by the acronym **ACID**:

ACID Properties

5. **Atomicity**
6. **Consistency**
7. **Isolation**
8. **Durability**

Atomicity: This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There is no such thing as partial transaction; if a transaction fails, all the changes made in the database so far by that transaction are rolled back, and the database remains unchanged.

Consistency: The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before a transaction, then after execution of the transaction, the database must be back to its consistent state.

Isolation: Each transaction is considered independent of other transactions. That is, the execution of multiple transactions concurrently will have the same effect as if they had been executed one after the other. But isolation does not ensure which transaction will execute first.

Durability: The changes made to the database by a successful transaction persist even after a system failure.

Shadow Database scheme Example for implementaion of Automicity and durabilty

Let's consider a simple banking database that has one table named **`Account`** with two fields: **`AccountID`** and **`Balance`**.

Shadow Database - Version 1

AccountID	Balance
1	1000
2	2000

1. Transaction Start: A transaction starts to transfer 200 from AccountID 1 to AccountID 2.

2. Separate Copy: A separate copy of the database is made and the changes are applied to it.

Modified Copy

AccountID	Balance
1	800
2	2200

3. Commit: The transaction completes successfully. The modified copy becomes the new shadow database.

Shadow Database - Version 2

AccountID	Balance
1	800
2	2200

4. System Failure: If the system crashes after this point, the shadow database (Version 2) is already in a consistent state, ensuring durability and atomicity.

Implementation of Isolation

Implementing isolation typically involves concurrency control mechanisms. Here are common mechanisms used:

1. Locking Mechanisms

Locking ensures exclusive access to a data item for a transaction. This means that while one transaction holds a lock on a data item, no other transaction can access that item.

- **Shared Lock (S-lock):** Allows a transaction to read an item but not write to it.
- **Exclusive Lock (X-lock):** Allows a transaction to read and write an item. No other transaction can read or write until the lock is released.
- **Two-phase Locking (2PL):** This protocol ensures that a transaction acquires all the locks

before it releases any. This results in a growing phase (acquiring locks and not releasing any) and a shrinking phase (releasing locks and not acquiring any).

2. Timestamp-based Protocols

Every transaction is assigned a unique timestamp when it starts. This timestamp determines the order of transactions. Transactions can only access the database if they respect the timestamp order, ensuring older transactions get priority.

c) Explain the Time Stamp - Based Concurrency Control protocol? **10M**

Timestamp-based protocols are concurrency control mechanisms used in databases to ensure serializability and to avoid conflicts without the need for locking. The main idea behind these protocols is to use a timestamp to determine the order in which transactions should be executed. Each transaction is assigned a unique timestamp when it starts.

Here are the primary rules associated with timestamp-based protocols:

1. Timestamp Assignment: When a transaction (T) starts, it is given a timestamp (TS(T)). This timestamp can be the system's clock time or a logical counter that increments with each new transaction.

2. Reading/Writing Rules: Timestamp-based protocols use the following rules to determine if a transaction can read or write an item:

- **Read Rule:** If a transaction (T) wants to read an item that was last written by transaction (T') with (TS(T') > TS(T)), the read is rejected because it's trying to read a value from the future. Otherwise, it can read the item.
- **Write Rule:** If a transaction (T) wants to write an item that has been read or written by a transaction (T') with (TS(T') > TS(T)), the write is rejected. This avoids overwriting a value that has been read or written by a younger transaction.

3. Handling Violations: When a transaction's read or write operation violates the rules, the transaction can be rolled back and restarted or aborted, depending on the specific protocol in use.

Let's look at examples to better understand these rules:

Example on Timestamp-based protocols

Suppose two transactions (T1) and (T2) with timestamps 5 and 10 respectively:

1. (T1) reads item (A).
2. (T2) writes item (A).
3. (T1) tries to write item (A).

According to the write rule, (T1) can't write item (A) after (T2) has written it, because (TS(T2) >

TS(T1)). Thus, (T1)'s write operation will be rejected.

Example-2

Suppose two transactions (T1) and (T2) with timestamps 5 and 10 respectively:

1. (T2) writes item (A).
2. (T1) tries to read item (A).

According to the read rule, (T1) can't read item (A) after (T2) has written it, as (TS(T2) > TS(T1)). So, (T1)'s read operation will be rejected.

Advantages of Timestamp-based Protocols

1. **Deadlock-free:** Since there are no locks involved, there's no chance of deadlocks occurring.
2. **Fairness:** Older transactions have priority over newer ones, ensuring that transactions do not starve and get executed in a fair manner.
3. **Increased Concurrency:** In many situations, timestamp-based protocols can provide better concurrency than lock-based methods.

Disadvantages of Timestamp-based Protocols

1. **Starvation:** If a transaction is continually rolled back due to timestamp rules, it may starve.
2. **Overhead:** Maintaining and comparing timestamps can introduce overhead, especially in systems with a high transaction rate.
3. **Cascading Rollbacks:** A rollback of one transaction might cause rollbacks of other transactions.

One of the most well-known timestamp-based protocols is the **Thomas Write Rule**, which modifies the write rule to allow certain writes that would be rejected under the basic timestamp protocol. The idea is to ignore a write that would have no effect on the outcome, rather than rolling back the transaction. This reduces the number of rollbacks but can result in non-serializable schedules.

In practice, timestamp-based protocols offer an alternative approach to concurrency control, especially useful in systems where locking leads to frequent deadlocks or reduced concurrency. However, careful implementation and tuning are required to handle potential issues like transaction starvation or cascading rollbacks.