Shadow Paging Method

Explain shadow paging method in detail.

Shadow Paging Method (5-6 Marks)

The **Shadow Paging** method is a technique used for ensuring the **consistency** and **reliability** of a database during transactions, particularly in systems that rely on **page-level** storage. It is an important concept in **database recovery** and **transaction management**.

In the shadow paging method, the database uses **two copies** of the database pages: the **shadow page** and the **current page**.

Key Features of Shadow Paging:

1. **Shadow Page**: This is a **read-only copy** of the database page. It contains the previous state of the data before any updates or modifications. These pages are not altered during the transaction.
2. **Current Page**: This contains the **latest state** of the database after the update operation. These pages are mutable and store the modified data during the transaction.

Steps in Shadow Paging:

1. **Initial Setup**: At the beginning of a transaction, the database system creates a **shadow copy** of the current database pages. The shadow pages contain the same data as the original pages at the start of the transaction.
2. **Transaction Processing**: During the transaction, the database system **modifies** the current pages. The shadow pages remain unmodified and continue to reflect the state of the database before the transaction started.
3. **Commit Phase**: When the transaction is ready to commit, the system verifies if the changes made in the current pages are correct and consistent.
   - If the transaction is **successful**, the modified pages in the current copy are **committed** to the original pages, and the shadow pages are discarded.
   - If the transaction **fails** or is **rolled back**, the database will use the **shadow pages** (which contain the pre-transaction data) to restore the database to its previous consistent state.
4. **Rollback**: In case of an error or failure, the system **reverts** to the shadow pages, ensuring that no partial updates or corrupted data are committed to the actual database.

Advantages of Shadow Paging:

- **No Logging Required**: Unlike other recovery methods (like Write-Ahead Logging), shadow paging does not require logging of individual database operations. This simplifies the recovery process.

- **Instant Rollback**: In case of a failure, the rollback is immediate because the shadow pages already contain the data before the transaction began.
- **Simplified Recovery**: As the shadow pages contain the original state, recovery from failure is straightforward and can be done without the need to read transaction logs.

Disadvantages of Shadow Paging:

- **Overhead**: Maintaining shadow copies of the entire database increases storage requirements, leading to potential **space overhead**.
- **Concurrency Issues**: Shadow paging can lead to performance bottlenecks in high-concurrency systems due to the need to maintain and manage two copies of the database pages.

In conclusion, shadow paging is a simple yet effective method of ensuring consistency in a database system, providing strong transaction guarantees through the use of shadow copies of the database pages. However, it is better suited for systems with low to moderate update rates due to its storage and performance overhead.

[6]
Define view and conflict serializable schedule with suitable example.

View and Conflict Serializable Schedules (5-6 Marks)

In database management systems, a **schedule** refers to the sequence in which the operations (like read and write) of multiple transactions are executed. The correctness of a schedule is crucial to ensure that the transactions' results are consistent and follow the **ACID properties** (Atomicity, Consistency, Isolation, Durability). There are two important types of serializability used to analyze schedules: **View Serializability** and **Conflict Serializability**.

1. Conflict Serializable Schedule:

A schedule is **conflict serializable** if it can be transformed into a **serial schedule** by swapping non-conflicting operations. In other words, if two operations do not conflict with each other, their order can be interchanged without affecting the correctness of the final result.

- **Conflicting operations**: Two operations conflict if they satisfy all of the following:
  1. They belong to different transactions.
  2. They access the same data item.
  3. At least one of the operations is a write.
- **Conflict Serializable Schedule Example:**

  Consider the following schedule S:

Operation	Transaction
R1(A)	T1

Operation	Transaction
W1(A)	T1
R2(A)	T2
W2(A)	T2

In this schedule:

- $R1(A)$ and $W2(A)$ are **conflicting** (since both access the same data item A and one is a write).
- $W1(A)$ and $R2(A)$ are **conflicting** (same data item, and one is a write).

The schedule can be serialized as:

1. **T1** → **T2**, where T1 completes all its operations before T2 starts. Thus, we can convert this schedule into a serial schedule without any issues. Hence, the schedule is **conflict serializable**.

2. View Serializable Schedule:

A schedule is **view serializable** if it produces the same final result as a serial schedule, even though the operations may be interleaved. In view serializability, the **read** and **write** operations are examined to ensure that the final state (or view) of the database is consistent with some serial execution of the transactions.

- **View Equivalence**: Two schedules are view equivalent if:
    1. The **initial read** of a data item in both schedules is the same.
    2. The **values written** to a data item in both schedules are the same.
    3. The **final write** of a data item in both schedules is the same.
- **View Serializable Schedule Example:**

    Consider the schedule S:

Operation	Transaction
R1(A)	T1
W1(A)	T1
R2(A)	T2
W2(A)	T2

This schedule can be considered view serializable because:

- Transaction T1 reads and writes A.
- Transaction T2 also reads and writes A after T1, and the final write on A is done by T2.

We can convert this schedule into a serial one as:

1. **T1** → **T2**: T1 reads and writes A, followed by T2, which reads and writes A, resulting in the same final view as a serial execution.

Thus, this schedule is **view serializable**, even though the operations might have been interleaved.

Key Differences Between Conflict and View Serializability:

1. **Conflict Serializability**:
   - Based on the conflict between operations.
   - More restrictive than view serializability.
   - If a schedule is conflict serializable, it is also view serializable.
2. **View Serializability**:
   - Based on the final state (or view) of the data.
   - Less restrictive than conflict serializability.
   - A schedule can be view serializable without being conflict serializable.

In conclusion, **conflict serializability** is a stricter criterion, while **view serializability** allows more flexibility but ensures the same final outcome. Both concepts are important in ensuring that the schedule of transactions does not violate consistency and isolation properties in a multi-user database system.

Give test for conflict serializability. Check whether following schedule is conflict serializable. [8]

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

SOLVE

Something went wrong, please try reloading the conversation.

What is concurrency control? Explain time stamping method.

Concurrency Control and Time Stamping Method (5-6 Marks)

Concurrency Control refers to the management of concurrent access to the database in a multi-user environment. The goal of concurrency control is to ensure that database transactions are executed in such a way that the **ACID properties** (Atomicity, Consistency, Isolation, and Durability) are preserved, even when multiple transactions are executed simultaneously. This ensures that the final database state is consistent and correct, despite the interleaving of transactions.

Time Stamping Method:

The **Time Stamping** method is a **concurrency control technique** that uses **timestamps** to manage the order of transactions and ensure serializability (i.e., the execution of concurrent transactions results in a final state equivalent to a serial execution of those transactions). In this method, every transaction is given a unique **timestamp** at the time of its initiation. The timestamps are used to determine the order of conflicting operations between transactions.

Working of Time Stamping Method:

1. **Assigning Timestamps**:
   - Each transaction is assigned a unique **timestamp** when it begins. Typically, the timestamp is assigned based on the system's clock or a counter that increments with each new transaction. For example, transaction $T1$ might have timestamp 1, $T2$ might have timestamp 2, and so on.
2. **Basic Rules of Time Stamping**:
   - **Read and Write Rules**: The time stamping method enforces rules that determine the order in which conflicting operations (read and write) are executed.
     - **Read Rule**: If a transaction reads a data item, it must read the value that was written by the transaction with the **earlier timestamp**.
     - **Write Rule**: If a transaction writes a data item, it must write the value if no other transaction has written it with a timestamp later than the current transaction's timestamp.

   These rules prevent scenarios where a transaction reads outdated or inconsistent data, ensuring serializability.
3. **Conflict Resolution**: When two transactions conflict (i.e., they both access the same data item and at least one of them is a write operation), the transaction with the **earlier timestamp** has precedence. The system will **reject** any operation that violates the timestamp order and cause inconsistency.
4. **Transaction Execution**:
   - A transaction executes its operations based on its timestamp. If at any point, a conflicting transaction with an earlier timestamp attempts to access the data, it will either be delayed or rolled back to maintain the proper order of execution.

Example of Time Stamping Method:

Let's consider the following scenario:

- **Transaction T1** (Timestamp 1) wants to **write** data item A.
- **Transaction T2** (Timestamp 2) wants to **read** data item A.
- **Transaction T3** (Timestamp 3) wants to **write** data item A.

If the timestamps follow the order $T1 < T2 < T3$, the system must enforce the following:

1. **T2** can read **A** only if it was written by **T1** (the earlier transaction).
2. **T3** can only write **A** if no other transaction has already written it with a **later timestamp** than its own.

Thus, the operations are executed in a way that ensures **serializability** based on their timestamps, meaning that the final result will be equivalent to some serial execution of the transactions.

Advantages of Time Stamping Method:

1. **Simplicity**: The timestamp method is simple and does not require the maintenance of complex locks.
2. **Deadlock-Free**: Since there are no locks involved in this method, the issue of deadlocks (where transactions wait indefinitely for resources) does not arise.
3. **Global Ordering**: The method guarantees a **global order** for transaction execution, ensuring consistency across the system.

Disadvantages of Time Stamping Method:

1. **Rollback Overhead**: If a transaction violates the timestamp order, it may need to be rolled back, which introduces overhead.
2. **Timestamp Management**: Ensuring that the timestamps are unique and correctly ordered is essential for the method's effectiveness, which might involve additional system complexity.
3. **Non-Serializable Transactions**: The system may allow non-serializable schedules if the timestamp rules are not correctly implemented, leading to potential inconsistencies.

Conclusion:

The **Time Stamping Method** is an efficient concurrency control mechanism that ensures **serializability** by assigning unique timestamps to transactions and using these timestamps to order conflicting operations. While it eliminates deadlocks and simplifies transaction management, it can also incur additional overhead in terms of rollbacks and timestamp management.

When do deadlock happens? How to prevent them?

Deadlock in Databases (5-6 Marks)

Deadlock in a database system occurs when two or more transactions are in a state where each transaction is waiting for another to release resources, and none of them can proceed. This leads to a situation where the transactions are indefinitely waiting for each other to release the resources they need to complete their execution, resulting in a **standstill**.

Deadlock typically happens when the following four conditions are met:

1. **Mutual Exclusion**: At least one resource is in a non-shareable mode, meaning it can only be used by one transaction at a time.

2. **Hold and Wait**: A transaction is holding one resource and is waiting for additional resources that are currently being held by other transactions.
3. **No Preemption**: Resources cannot be forcibly taken from transactions holding them; they must be released voluntarily.
4. **Circular Wait**: A set of transactions exist such that each transaction is waiting for a resource held by the next transaction in the set, forming a **circle of dependencies**.

Example of Deadlock:

Consider the following scenario:

- **Transaction T1** holds a lock on resource R1 and waits for resource R2.
- **Transaction T2** holds a lock on resource R2 and waits for resource R1.

In this case, both transactions are waiting for each other to release the resources they need, causing a deadlock.

How to Prevent Deadlocks?

Deadlock prevention aims to **avoid** the conditions that lead to deadlocks. Several strategies can be employed to prevent deadlocks:

1. Resource Allocation Graph (RAG):

- One method for preventing deadlock is to maintain a **Resource Allocation Graph**. In this graph:
  - Nodes represent both transactions and resources.
  - A directed edge from a transaction to a resource indicates that the transaction is requesting the resource.
  - A directed edge from a resource to a transaction indicates that the transaction holds the resource.
- **Deadlock Prevention**: The system checks the graph for cycles. If a cycle is detected (indicating a circular wait), it prevents the request that would complete the cycle, thus avoiding deadlock.

2. Enforcing the "No Wait" Condition (Locking Order):

- To prevent deadlocks, the system can enforce a strict **ordering of resource acquisition**. Each transaction must request the resources in a predefined order, and this order must be followed strictly.
- For example, if resource R1 must always be requested before R2, then transactions must acquire R1 before R2, and any transaction that requests R2 first would be blocked until R1 is acquired first.

3. Timeouts:

- A transaction can be allowed to wait for a resource only for a **limited time**. If the transaction does not acquire the required resource within the timeout period, it will be **rolled back** or **aborted** to break the deadlock cycle.

- This ensures that a transaction doesn't wait indefinitely for a resource, and other transactions are not blocked.

4. Resource Preemption:

- This involves **preempting** resources from one transaction and reallocating them to another. For example, if Transaction T1 is waiting for R2 but holding R1, and Transaction T2 is waiting for R1 but holding R2, the system might preempt one resource (e.g., R1 from T1) and allocate it to the waiting transaction.
- Preemption may cause transaction rollbacks, but it ensures that transactions are not stuck in a deadlock.

5. Transaction Ordering:

- **Transaction timestamps** can be used to prevent deadlocks. Each transaction is assigned a timestamp when it starts. If a transaction tries to acquire a resource that is already held by another transaction with an **older timestamp**, it is forced to wait. This ensures that transactions follow an order and prevents circular waiting.

6. Wait-Die and Wound-Wait Schemes:

These are two deadlock prevention schemes used to manage resource allocation between transactions:

- **Wait-Die Scheme**:
  - If an older transaction (based on timestamp) requests a resource held by a younger transaction, it is allowed to wait.
  - If a younger transaction requests a resource held by an older transaction, it is **killed (aborted)** (i.e., the younger transaction "dies").
- **Wound-Wait Scheme**:
  - If an older transaction requests a resource held by a younger transaction, the older transaction **preempts** the resource and forces the younger transaction to roll back.
  - If a younger transaction requests a resource held by an older transaction, it must wait.

Conclusion:

Deadlocks occur when transactions are stuck in a cycle of waiting for resources, and each transaction holds a resource that the others need. To prevent deadlocks, methods like **resource allocation graphs**, **locking order**, **timeouts**, **preemption**, and **transaction ordering** can be employed. These techniques aim to eliminate at least one of the conditions necessary for a deadlock, ensuring that the system remains deadlock-free and that transactions can complete successfully.

Explain the concept of Transaction. Describe ACID properties of transaction.

Concept of Transaction (5-6 Marks)

A **transaction** in a database management system (DBMS) refers to a **logical unit of work** that consists of one or more operations (such as read, write, or update) on the database. A transaction is typically associated with a user action or a series of actions that must be executed in a consistent manner. In a DBMS, transactions are used to manage changes to the database to ensure that the system remains **consistent** and **reliable**.

Transactions are designed to ensure that a database maintains its **ACID properties** (Atomicity, Consistency, Isolation, Durability) even when multiple transactions are happening concurrently or if the system experiences failures.

ACID Properties of Transaction

The **ACID properties** are a set of four key properties that ensure reliable processing of database transactions:

1. **Atomicity**:
    - Atomicity guarantees that a transaction is treated as a single unit, meaning it either **completes entirely** or **does not execute at all**.
    - If any part of the transaction fails, the **entire transaction is rolled back**, and the database remains in its initial state before the transaction began.
    - Atomicity ensures that partial updates or incomplete operations are **not saved** to the database, maintaining data integrity.

    Example: Suppose a transaction involves transferring money from one bank account to another:

    - If the transaction is atomic, both the withdrawal from one account and the deposit into the other account must either **both succeed** or **both fail**. If the deposit fails after the withdrawal, the transaction is rolled back, and the database remains unchanged.
2. **Consistency**:
    - Consistency ensures that a transaction brings the database from one **valid state** to another, maintaining all defined **constraints, rules, and relationships**.
    - At the end of the transaction, the database should satisfy all integrity constraints (e.g., foreign key constraints, uniqueness constraints), ensuring that the data remains **consistent** with the database schema.

    Example: In a banking system, a consistency rule might dictate that the total amount of money across all accounts must always be equal to the sum of individual account balances. A transaction that violates this rule would not be allowed, ensuring that the database stays consistent.
3. **Isolation**:
    - Isolation ensures that **concurrent transactions** are executed in such a way that the intermediate states of a transaction are not visible to other transactions.
    - Each transaction appears to be executed in isolation, even if multiple transactions are running concurrently. This prevents one transaction's operations from being **interfered with** by another transaction.

- The level of isolation can vary, and DBMSs provide different **isolation levels** to balance performance and isolation, such as **Read Uncommitted**, **Read Committed**, **Repeatable Read**, and **Serializable**.

  Example: Suppose two transactions, T1 and T2, are trying to update the same record. Isolation ensures that T2 will not see the intermediate changes made by T1 until T1 has committed its changes, thus preventing **dirty reads** or **inconsistent data**.

4. **Durability**:
   - Durability ensures that once a transaction is committed, its effects are **permanent** and will not be lost, even in the event of a system crash or power failure.
   - After a transaction is successfully committed, the changes made to the database are guaranteed to persist in non-volatile storage (such as a disk), and the system will recover to the last consistent state after any crash.

   Example: If a transaction transfers funds between two bank accounts and the transaction is committed, even if the system crashes immediately after the commit, the changes to the accounts' balances will still be retained in the database when the system recovers.

Summary of ACID Properties:

Property	Description
Atomicity	The transaction is all-or-nothing (either complete or not executed).
Consistency	The transaction takes the database from one consistent state to another.
Isolation	The transaction is isolated from the effects of other concurrent transactions.
Durability	Once a transaction is committed, its changes are permanent and will survive system crashes.

Conclusion:

A **transaction** is a fundamental concept in DBMSs, ensuring that data modifications are reliable and consistent. The **ACID properties** — **Atomicity**, **Consistency**, **Isolation**, and **Durability** — provide the foundation for managing transactions, ensuring that they execute correctly and that the database maintains its integrity even in the face of failures or concurrency. These properties help prevent data corruption, ensure consistency, and provide reliable processing of database operations.

Explain Deferred database modification.

Deferred Database Modification (5-6 Marks)

Deferred Database Modification is a **transaction management technique** used to ensure that changes made by a transaction are **not written** to the database until the transaction has been **successfully committed**. In this method, all the updates and modifications made by a transaction are stored in a **log** during the transaction's execution, and the actual modification to the database is deferred until the **commit phase**.

This approach is primarily used to ensure **durability** and to **prevent partial or inconsistent updates** to the database if a transaction fails before it is committed. Deferred database modification helps to ensure the **atomicity** and **durability** of transactions.

How Deferred Database Modification Works:

1. **Transaction Starts**:
    - When a transaction begins, it performs all of its **operations (read, write)** on data, but the **modifications are not immediately written** to the database.
    - Instead of writing the changes to the database, the system stores the **log records** (containing the intended changes) in a **transaction log**.
2. **Transaction Executes**:
    - The transaction performs its operations as usual. The system **buffers** the changes (e.g., update, delete, or insert) but doesn't commit them to the database.
3. **Log Records**:
    - All changes are recorded in the **log file** with enough information to undo or redo the modifications, if necessary. The log contains:
        - The **old values** (before the modification).
        - The **new values** (after the modification).
        - The **transaction ID** and **timestamp**.
    - These log records help in **recovering** the database in case of system failure or crash.
4. **Commit**:
    - When the transaction successfully completes all its operations and is ready to commit, the system writes the **modifications to the database**. This ensures that all updates are done together, and the database reflects the final consistent state of the transaction.
    - Once the transaction commits, the system **flushes** the modifications to the database, and the changes become **permanent**.
5. **Rollback** (in case of failure):
    - If a failure occurs before the transaction is committed, the changes made by the transaction are **not** written to the database.
    - The system can use the **log records** to undo any changes made by the transaction, restoring the database to its state before the transaction began.
    - This ensures that partial or inconsistent updates do not get saved in the database.

Advantages of Deferred Database Modification:

1. **Reduced Disk I/O**:
    - Since modifications are only written to the database when the transaction commits, there is a reduction in the amount of disk I/O operations, which can improve performance.
2. **Ensures Consistency**:
    - By deferring changes until the commit, it ensures that no intermediate or partial changes are ever visible to other transactions, which helps in maintaining database consistency.

3. **Simplified Recovery**:
   - During recovery (in case of system failure), the system only needs to **rollback uncommitted transactions**. As no changes are made to the database until the commit, recovery is simpler and faster.
4. **Atomicity**:
   - The technique ensures **atomicity**, as all changes made by a transaction are either **fully committed** or **fully discarded**. In case of a failure, no partial changes will remain in the database.

Disadvantages of Deferred Database Modification:

1. **Longer Waiting Time for Modifications**:
   - Since changes are not immediately written to the database, users or other transactions may have to wait longer for the actual changes to be reflected.
2. **Increased Log File Size**:
   - Storing all changes in the transaction log until the commit can result in large log files, especially for long-running transactions or transactions that modify a significant amount of data.
3. **Not Ideal for Long Transactions**:
   - The approach may not be suitable for transactions that last a long time, as it would delay the actual modification of data, potentially causing delays in the system.

Example of Deferred Database Modification:

Consider a bank transaction in which funds are transferred from **Account A** to **Account B**:

1. **Transaction Start**: Transaction T1 begins and reads the balances of Account A and Account B.
2. **Transaction Executes**: T1 prepares to update Account A's balance by **decreasing** the amount and Account B's balance by **increasing** the amount. However, these changes are not yet written to the database. Instead, the system stores these updates in the transaction log.
3. **Commit**: Once T1 successfully completes all operations, it commits the transaction, and the changes are written to the database, reflecting the updated balances of both accounts.

If a failure occurs before the commit:

- The changes are not written to the database, and T1's operations are rolled back, ensuring the database remains in a consistent state.

Conclusion:

Deferred Database Modification is a transaction management technique where updates are delayed and only written to the database upon transaction commit. It ensures **atomicity** and **durability** by preventing partial updates in case of transaction failure. However, it may increase log size and delay the visibility of changes, making it more suitable for short transactions or environments where system failures are rare.