22.1

1. Why Data Inconsistency and Data Loss Might Occur

- **Concurrent Access**: When multiple users or processes access the same data at the same time, without proper handling, they may read or write incomplete or outdated data. This concurrent access can lead to data inconsistencies, as different users may see or modify different versions of the data.
- **Lack of Atomicity**: If a transaction (a series of database operations) is interrupted midway, it may leave the database in an inconsistent state. Without ensuring that transactions are atomic (fully completed or fully rolled back), some updates may be only partially applied, leading to inconsistencies.
- **Lost Updates**: When two transactions simultaneously update the same data, one transaction's changes may overwrite the others without warning, resulting in data loss or overwriting of essential information.
- **Dirty Reads**: In some cases, a transaction may read data that another transaction has modified but not yet committed. If the second transaction rolls back, the first transaction will have read invalid data, causing inconsistencies.

2. Why It's Necessary to Manage Concurrent Transactions

- **Ensuring Data Integrity**: Managing concurrent transactions helps to maintain the integrity of the data by ensuring that only one transaction can modify a piece of data at a time or by using mechanisms like versioning to track changes.
- **Preventing Anomalies**: Techniques like locking, timestamps, or multi-version concurrency control (MVCC) prevent anomalies, such as lost updates and dirty reads, by coordinating access among multiple users.
- **Guaranteeing ACID Properties**: Proper management of concurrent transactions ensures that databases meet the ACID (Atomicity, Consistency, Isolation, Durability) properties, which are essential for reliable and predictable data management.
- **Improving Reliability and User Experience**: By ensuring that each user or process receives accurate and consistent data, concurrency control improves application reliability and user trust. It prevents scenarios where different users see conflicting information or where critical updates are lost.

22.2

1. Atomicity

- **Definition**: Atomicity ensures that a transaction is treated as a single "unit of work," meaning that either all its operations are completed, or none are. If any part of the transaction fails, the entire transaction is rolled back, leaving the database in its previous state.
- **Concurrency Control and Recovery**: In concurrency control, atomicity prevents partial changes from affecting other transactions. If a transaction fails midway, recovery mechanisms like rollback ensure that any changes made are undone, protecting the integrity of other transactions.
- **Example**: Consider a bank transfer between two accounts, A and B, where $100 is deducted from A and added to B. If the transfer completes, both accounts reflect the transaction. If there is a failure after deducting $100 from A but before adding it to B, the transaction must roll back to keep both accounts consistent.

2. Consistency

- **Definition**: Consistency ensures that a transaction brings the database from one valid state to another, maintaining all predefined rules, constraints, and relationships.
- **Concurrency Control and Recovery**: Concurrency control helps maintain consistency by enforcing data integrity rules even when multiple transactions occur. Recovery mechanisms ensure that any transaction violating consistency is aborted or rolled back.
- **Example**: Suppose an inventory system has a constraint where stock levels cannot be negative. A transaction attempting to reduce stock below zero would violate consistency. If multiple transactions try to adjust the same stock level concurrently, concurrency control (like locking) ensures that only valid transactions succeed, maintaining the integrity of stock levels.

3. Isolation

- **Definition**: Isolation ensures that transactions are executed independently of each other. Transactions do not see intermediate states of other concurrent transactions, and their outcomes do not affect each other directly.
- **Concurrency Control and Recovery**: Concurrency control mechanisms like locking, timestamps, or multi-version concurrency control (MVCC) manage isolation by preventing phenomena like dirty reads, non-repeatable reads, and phantom reads. Recovery mechanisms can roll back transactions that violate isolation requirements.

- **Example**: In a hotel booking system, if two users try to book the same room at the same time, isolation ensures that only one transaction succeeds. Concurrency control mechanisms like locking prevent both transactions from double-booking the room.

4. Durability

- **Definition**: Durability ensures that once a transaction is committed, its changes are permanently saved, even in the event of a system crash.
- **Concurrency Control and Recovery**: While durability itself is not directly impacted by concurrency, it is crucial for recovery. Recovery mechanisms, like write-ahead logging and checkpointing, help ensure that committed data persists and can be restored after unexpected failures.
- **Example**: In an e-commerce transaction, when a user's order is placed, the transaction commits, and the order details are saved permanently. Even if the system crashes afterward, durability ensures that the order remains recorded and can be recovered without loss.

22.3

In a multi-user environment where concurrent access to a database is allowed, several types of issues can arise due to uncoordinated transactions. These issues generally stem from the lack of proper isolation between transactions and can lead to inconsistent data, loss of updates, and other anomalies. Here are the main types of problems that can occur:

1. Lost Update

- **Definition**: A lost update occurs when two or more transactions read the same data and then update it based on that initial value. The last update overwrites the previous updates, resulting in data loss.
- **Example**: Suppose two clerks are updating the inventory of a product with an initial quantity of 100 units. Clerk A reads the quantity as 100 and reduces it by 10 (resulting in 90). At the same time, Clerk B also reads the quantity as 100 and reduces it by 5 (resulting in 95). If Clerk A's update is applied last, Clerk B's update is lost, and the final quantity becomes 90 instead of 85.

2. Dirty Read

- **Definition**: A dirty read occurs when a transaction reads data that has been modified by another transaction that has not yet committed. If the other transaction is rolled back, the data read is invalid.

Example: Transaction T1 updates a customer's balance from $1,000 to $900. Before T1 commits, Transaction T2 reads the new balance as $900 and proceeds with other operations. If T1 is rolled back, T2's operations are based on an invalid balance, leading to inconsistent results.

3. Non-Repeatable Read

- **Definition**: A non-repeatable read occurs when a transaction reads the same data multiple times and gets different values each time because another concurrent transaction modified the data in between reads.
- **Example**: Transaction T1 reads a customer's balance as $1,000. While T1 performs other tasks, Transaction T2 updates the balance to $900 and commits. When T1 reads the balance again, it now sees $900 instead of the original $1,000, leading to inconsistencies in its operations based on the balance.

4. Phantom Read

- **Definition**: A phantom read occurs when a transaction reads a set of rows that match certain criteria but gets a different set of rows when it re-reads because another transaction inserted, updated, or deleted rows in between.
- **Example**: Transaction T1 reads all orders placed in the last 24 hours and calculates a total. Meanwhile, Transaction T2 inserts a new order within the same time frame and commits. When T1 re-reads the orders, it sees the new order, causing a discrepancy in the calculations.

5. Write Skew

- **Definition**: Write skew occurs when two transactions read the same data but update different, dependent fields. Because they do not directly modify each other's values, they both succeed but leave the data in an inconsistent state.
- **Example**: Consider two doctors who are required to be on call at the hospital at any time. Transaction T1 checks that Doctor A is on call and decides to remove Doctor B from the on-call list. Simultaneously, Transaction T2 checks that Doctor B is on call and decides to remove Doctor A. Both transactions commit successfully, but the hospital ends up with no doctors on call, leading to a write skew.

22.4

One effective mechanism for concurrency control that prevents issues like lost updates, dirty reads, non-repeatable reads, and phantom reads is **Two-Phase Locking (2PL)**. This locking protocol ensures serializability (i.e., transactions are executed in a way that produces the same results as if they were executed sequentially) by managing locks in two distinct phases: the *growing phase* and the *shrinking phase*. Here's a detailed look at how 2PL works, how it prevents concurrency issues, and how it interacts with the transaction mechanism.

Two-Phase Locking (2PL) Mechanism

The 2PL protocol ensures that:

1. **In the Growing Phase**: A transaction can acquire locks as needed but cannot release any locks.
2. **In the Shrinking Phase**: Once a transaction releases a lock, it cannot acquire any new locks.

How 2PL Prevents Concurrency Problems

1. **Lost Update**:

If two transactions attempt to update the same inventory count, the first transaction to acquire the lock completes its update and releases the lock before the second transaction can proceed. This ensures that both transactions work with a consistent inventory count, preventing overwrites.

2. **Dirty Read**:

Suppose Transaction T1 updates a customer's balance but has not yet committed. Transaction T2 attempts to read this balance; under 2PL, T2 cannot read it until T1 releases its lock upon commit. This ensures that T2 only reads committed data, avoiding dirty reads.

3. **Non-Repeatable Read**:

If Transaction T1 reads a customer's balance twice, no other transaction can modify the balance in between T1's reads because T1 holds a shared lock on the data item. This guarantees consistent reads and prevents non-repeatable reads.

4. Phantom Read
   Transaction T1 reads all orders in a specific range. To prevent phantom reads, it acquires a lock on the entire range of orders. If another transaction, T2, tries to insert an order within that range, it will be blocked until T1 commits. This prevents any changes that could alter T1's result set, thereby avoiding phantom reads.

Benefits and Limitations of 2PL

- **Benefits**: Ensures consistency, prevents all discussed concurrency issues, and maintains ACID properties.
- **Limitations**: 2PL can introduce performance bottlenecks, as transactions are sometimes forced to wait for locks to be released. This can lead to reduced throughput and potential deadlocks, which are typically resolved with deadlock detection or timeout mechanisms.

22.5

1. **Serial Schedule**

- Characteristics: Serial schedules are the simplest and safest form of execution because they eliminate the possibility of concurrency issues by avoiding any interaction between transactions.
- Example: If we have two transactions, T1 and T2, a serial schedule would either execute T1 completely followed by T2, or T2 followed by T1. There would be no interleaving of operations from T1 and T2.

2. **Non-Serial Schedule**

- Characteristics: Non-serial schedules can lead to concurrency problems like dirty reads, lost updates, or non-repeatable reads if not managed properly. However, not all non-serial schedules cause issues; some may still produce correct results if they maintain serializability.
- Example: If T1 and T2 are interleaved such that some of T1's operations are executed, followed by some of T2's, and then more operations from T1, this is a non-serial schedule.

3. **Serializable Schedule**

- Characteristics: A schedule is serializable if the outcome of executing the interleaved transactions is equivalent to executing them in some serial order. Serializable schedules allow for concurrent processing without risking data anomalies.
- Example: If T1 and T2 are interleaved in such a way that the final outcome is identical to either executing T1 fully before T2 or vice versa, the schedule is serializable.

Rules for Equivalence of Schedules

To determine if two schedules are equivalent, they must meet certain criteria, which help in identifying serializability. The primary rules for equivalence are **result equivalence**, **conflict equivalence**, and **view equivalence**.

1. Result Equivalence

- **Definition**: Two schedules are result equivalent if they produce the same final state on the database.
- **Use**: Result equivalence is a basic form of schedule equivalence but does not consider the order of operations in detail. It simply ensures that the end result of executing two schedules is the same.

2. Conflict Equivalence

- **Definition**: Two schedules are conflict equivalent if they contain the same set of conflicting operations in the same order.
- **Conflicting Operations**: Two operations conflict if they belong to different transactions, access the same data item, and at least one of them is a write operation. Conflict equivalence is more rigorous than result equivalence because it considers the order of conflicting operations.
- **Example**: If T1 and T2 both have operations on a shared data item, a conflict equivalent schedule would maintain the same sequence of these operations as in a serial schedule, preserving consistency.

3. View Equivalence

- **Definition**: Two schedules are view equivalent if they satisfy the following conditions:
    - **Initial Reads**: Each transaction in both schedules reads the same initial value for any data it accesses.
    - **Read-Write Dependency**: If a transaction reads a value written by another transaction in one schedule, it must read the same value in the other schedule.

    - **Final Writes**: The final write operations on each data item in both schedules must be the same.

- **Use**: View equivalence is the strictest form of equivalence and ensures that both schedules appear identical in terms of both initial reads and results. It is often used when conflict equivalence alone is insufficient to determine serializability.

22.9

How Timestamp-Based Protocols Work:

- **Basic Principle**: The timestamp of a transaction is used to decide whether a transaction should be allowed to access a particular data item, based on the timestamps of other transactions that have already accessed or modified the same data item.
- **Ordering**: If a transaction with a lower timestamp tries to perform an operation that conflicts with a transaction with a higher timestamp, the transaction with the lower timestamp is rejected, and usually, it will be rolled back to maintain serializability.

Basic Operations:

1. **Read Operation**: When a transaction attempts to read a data item, it is allowed if no other transaction has written the data item after the reading transaction's timestamp.
2. **Write Operation**: When a transaction attempts to write to a data item, it is allowed if no other transaction has read or written that data item after the writing transaction’s timestamp.

Locking-Based Protocols vs. Timestamp-Based Protocols

Aspect	Timestamp-Based Protocols	Locking-Based Protocols
Control Mechanism	Uses timestamps to determine the order of transactions	Uses locks to manage access to data items
Transaction Rollback	Transactions are rolled back if they violate order	Transactions may be blocked or aborted, especially in deadlocks
Deadlocks	No deadlocks, but rollbacks may occur	Deadlocks can occur and need detection or resolution
Concurrency	Limited by frequent rollbacks in high-contention scenarios	Can allow high concurrency unless lock contention occurs
Performance	May suffer from rollbacks, but simpler to implement	May suffer from lock contention and overhead, but generally better concurrency

22.13

In a database environment, several types of failures can occur, each with distinct impacts on data integrity:

1. **Transaction Failures**: Occur when individual transactions cannot complete, often due to invalid inputs, deadlocks, or logical errors. These failures require rollbacks to prevent partial changes from affecting data consistency.
2. **System (or Server) Failures**: Caused by unexpected crashes or power outages, leading to the loss of in-progress transactions. This often results in an inconsistent state if unsaved changes are lost.
3. **Media (or Disk) Failures**: Result from physical damage to storage, such as a hard disk crash, which may lead to permanent data loss. Restoring from backups and reapplying recent changes from logs are essential here.
4. **Application Errors**: Bugs in applications interacting with the database can lead to incorrect updates or deletions. Recovery mechanisms must correct these errors to maintain integrity.
5. **Natural Disasters**: Events like floods or fires can damage infrastructure, leading to extensive data loss if no off-site backups or disaster recovery plans are in place.

Importance of Recovery Mechanisms

Recovery mechanisms are critical in multi-user DBMSs to ensure **data consistency** and **availability**. They help maintain **atomicity and durability** by ensuring that transactions are either fully completed or rolled back if incomplete. Recovery systems minimize **downtime** and data loss, allowing the DBMS to quickly resume operations after a failure. They also help manage **concurrent transactions** and prevent anomalies from cascading failures, keeping the database in a stable state for all users.

Key Recovery Techniques

To handle failures, DBMSs use methods such as **transaction logging** (recording operations for rollback or replay), **checkpointing** (saving consistent states to simplify recovery), **shadow paging** (storing pages until transactions are complete), and **backup/restoration** for handling major failures like media or natural disasters.

22.18

(a) read (T_1, bal_x), read(T_2, bal_x), write(T_1, bal_x), write(T_2, bal_x), commit(T_1), commit(T_2)

- **Conflict Serializable**: No, because there's a write-write conflict between T_1 and T_2 on bal_x that cannot be ordered to achieve serializability.
- **Recoverable**: Yes, T_2 commits after T_1, so there's no issue of cascading aborts.

(b) read (T_1, bal_x), read(T_2, bal_x), write(T_3, bal_x), read(T_1, bal_y), read(T_2, bal_y), commit(T_1), commit(T_2), commit(T_3)

- **Conflict Serializable**: Yes, because there are no conflicting operations that would prevent serializability.
- **Recoverable**: Yes, each transaction reads data that was not written by any uncommitted transaction.

(c) read (T_1, bal_x), write(T_2, bal_x), write(T_1, bal_x), commit(T_2), commit(T_1)

- **Conflict Serializable**: No, due to a write-read conflict where T_1 reads data after T_2 has written it, making it difficult to reorder without conflicts.
- **Recoverable**: Yes, T_1 commits after T_2, ensuring that no cascading aborts occur.

(d) write (T_1, bal_x), read(T_2, bal_x), write(T_2, bal_x), commit(T_2), commit(T_1)

- **Conflict Serializable**: No, because T_1 writes bal_x and then T_2 reads and writes it, creating a read-after-write dependency that cannot be reordered.
- **Recoverable**: Yes, T_2 commits before T_1, ensuring there's no risk of cascading aborts.

(e) read(T_1, bal_x), write(T_2, bal_x), write(T_1, bal_x), read(T_3, bal_x), commit(T_1), commit(T_2), commit(T_3)

- **Conflict Serializable**: No, T_1 and T_2 have write conflicts that make serializability unachievable.
- **Recoverable**: Yes, as all transactions commit without dependency conflicts, there's no cascading abort issue.

Analysis (Conflict Serializability & Recoverability)

1. **(a)**:
    - **Conflict Serializable**: Depends on the operations given (not visible in this image).
    - **Recoverable**: To determine this, I would need more details on specific operations in this example.
2. **(c)**:
    - **Conflict Serializable**: Depends on specifics not shown here.
    - **Recoverable**: Similarly, recoverability cannot be confirmed without specific transaction details.
3. **(d)**:
    - **Conflict Serializable**: Requires more info.
    - **Recoverable**: Similarly undetermined without more info.
4. **(e)**:
    - **Conflict Serializable**: Requires additional details to confirm.
    - **Recoverable**: To assess recoverability, further transaction info is needed.

22.19

Schedule (a)

Operations:
read(T_1, bal_x), read(T_2, bal_x), write(T_1, bal_x), write(T_2, bal_x), commit(T_1), commit(T_2)

1. Conflict between write(T_1, bal_x) and write(T_2, bal_x) (Write-Write conflict).
2. Directed edge from T_1 to T_2 due to T_1's write happening before T_2's write.

Precedence Graph:

- Nodes: T_1, T_2
- Edge: $T_1 \rightarrow T_2$ (from the write-write conflict)

Result: No cycle, so **serializable**.

Schedule (b)

Operations:
read(T_1, bal_x), read(T_2, bal_x), write(T_3, bal_x), read(T_1, bal_y), read(T_2, bal_y), commit(T_1), commit(T_2), commit(T_3)

1. Conflict between read(T_1, bal_x) and write(T_3, bal_x) (Read-Write conflict).
2. Conflict between read(T_2, bal_x) and write(T_3, bal_x) (Read-Write conflict).

Precedence Graph:

- Nodes: T_1, T_2, T_3
- Edges:
    - $T_1 \rightarrow T_3$ (from the read-write conflict on bal_x)
    - $T_2 \rightarrow T_3$ (from the read-write conflict on bal_x)

Result: No cycle, so **serializable**.

Schedule (c)

Operations:
read(T_1, bal_x), write(T_2, bal_x), write(T_1, bal_x), commit(T_2), commit(T_1)

1. Conflict between read(T_1, bal_x) and write(T_2, bal_x) (Read-Write conflict).

2. Conflict between write(T_2, bal_x) and write(T_1, bal_x) (Write-Write conflict).

Precedence Graph:

- Nodes: T_1, T_2
- Edges:
    - $T_1 \rightarrow T_2$ (from the read-write conflict on bal_x)
    - $T_2 \rightarrow T_1$ (from the write-write conflict on bal_x)

Result: There is a cycle ($T_1 \rightarrow T_2 \rightarrow T_1$), so **not serializable**.

Schedule (d)

Operations:
write(T_1, bal_x), read(T_2, bal_x), write(T_2, bal_x), commit(T_2), commit(T_1)

1. Conflict between write(T_1, bal_x) and read(T_2, bal_x) (Write-Read conflict).
2. Conflict between write(T_1, bal_x) and write(T_2, bal_x) (Write-Write conflict).

Precedence Graph:

- Nodes: T_1, T_2
- Edges:
    - $T_1 \rightarrow T_2$ (from the write-read conflict on bal_x)
    - $T_1 \rightarrow T_2$ (from the write-write conflict on bal_x)

Result: No cycle (edges do not form a loop), so **serializable**.

Schedule (e)

Operations:
read(T_1, bal_x), write(T_2, bal_x), write(T_1, bal_x), read(T_3, bal_x), commit(T_1), commit(T_2), commit(T_3)

1. Conflict between read(T_1, bal_x) and write(T_2, bal_x) (Read-Write conflict).
2. Conflict between write(T_2, bal_x) and write(T_1, bal_x) (Write-Write conflict).
3. Conflict between write(T_1, bal_x) and read(T_3, bal_x) (Write-Read conflict).

Precedence Graph:

- Nodes: T_1, T_2, T_3

- Edges:
    - $T_1 \rightarrow T_2$ (from the read-write conflict on bal_x)
    - $T_2 \rightarrow T_1$ (from the write-write conflict on bal_x)
    - $T_1 \rightarrow T_3$ (from the write-read conflict on bal_x)

Result: There is a cycle ($T_1 \rightarrow T_2 \rightarrow T_1$), so **not serializable**.

(b) What is a Checkpoint?

A **checkpoint** is a mechanism in database management systems (DBMS) used for recovery and efficiency. During a checkpoint, the DBMS saves a consistent state of the database to disk, including information on in-progress transactions. In case of a failure, the system can restart from the last checkpoint rather than from scratch, reducing the amount of work required for recovery. Checkpoints help ensure that committed transactions are saved, while any uncommitted transactions can be rolled back efficiently.

(c) What is a Lock?

A **lock** is a control mechanism used in databases to manage concurrent access to data. When a transaction locks a data item, other transactions are prevented from accessing it in a way that would cause conflicts. Locks help maintain data integrity by preventing issues like lost updates and dirty reads. Locks can be **shared** (allowing multiple transactions to read but not write) or **exclusive** (allowing one transaction to read or write exclusively).

(d) What is 2PL (Two-Phase Locking)?

Two-Phase Locking (2PL) is a concurrency control protocol that divides a transaction's locking actions into two distinct phases:

1. **Growing Phase**: The transaction acquires all necessary locks but cannot release any.
2. **Shrinking Phase**: After acquiring all locks, the transaction releases locks but cannot acquire new ones. 2PL ensures conflict serializability, meaning the schedule of transactions behaves as if transactions are executed serially, avoiding concurrency issues.

(e) What is a Deadlock?

A **deadlock** occurs when two or more transactions are waiting indefinitely for each other to release locks on resources. For example, if Transaction 1 locks Resource A and waits for Resource B (locked by Transaction 2), while Transaction 2 waits for Resource A, both transactions are stuck, unable to proceed. Deadlocks can halt progress in a database, requiring mechanisms like deadlock detection and resolution.

(f) What is Meant by Granularity? Give Examples.

Granularity refers to the size or level of data items locked by a transaction. It can range from a fine level (e.g., individual rows or records) to a coarse level (e.g., tables or entire databases). The finer the granularity, the more precise the lock but the higher the overhead. For example:

- **Fine-grained locks**: Row-level locks allow high concurrency as different transactions can lock different rows in the same table.
- **Coarse-grained locks**: Table-level locks prevent other transactions from accessing any row within the table, reducing concurrency but simplifying lock management.