Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

OVERVIEW OF TRANSACTIONS AND ACID PROPERTIES

CHAPTER 1: INTRODUCTION TO TRANSACTIONS IN DATABASE SYSTEMS

What is a Database Transaction?

A **transaction** in database systems refers to a sequence of one or more database operations (e.g., INSERT, UPDATE, DELETE) that are treated as a single unit of work. This unit of work must be **executed entirely** or **not at all**, ensuring consistency and integrity in the database.

The purpose of transactions is to guarantee the integrity of the database in cases of system crashes, power outages, or any other unexpected errors. For example, in a banking system, when transferring money between two accounts, the transaction must either **complete entirely** (money deducted from one account and added to another) or **not happen at all** (if any part of the transaction fails, neither account should be affected).

Transactions ensure **data consistency**, **recoverability**, and **atomicity**. They form the backbone of reliable database systems, particularly in systems with multiple users and complex operations. Each transaction can contain multiple operations, such as reading data, performing computations, and writing data back to the database. However, the database ensures that these operations are either **committed** (successfully applied) or **rolled back** (aborted if an error occurs).

Chapter 2: The ACID Properties of Transactions

Understanding ACID Properties

The **ACID properties** are a set of principles that ensure that database transactions are processed reliably. The acronym stands for **Atomicity, Consistency, Isolation, and Durability**. These properties are critical to maintaining the integrity and reliability of data in a multi-user environment. Let's take a deeper look at each of these properties:

Atomicity: All or Nothing

Atomicity ensures that a transaction is treated as a **single, indivisible unit**. This means that all operations within the transaction must succeed, or the transaction is considered **failed**. If even one operation fails, the database will **rollback** all changes made by the transaction, ensuring the database is in a consistent state.

Example of Atomicity:
Consider a banking system where you are transferring money from Account A to Account B. The transaction involves two operations:

1. Deducting money from Account A.
2. Adding money to Account B.

If the first operation succeeds but the second operation fails (for example, due to a network issue), the system ensures that Account A is **not debited** unless Account B is successfully credited, maintaining the database's consistency. The transaction would **rollback** all changes and not commit any part of it.

Consistency: Maintaining Database Integrity

Consistency ensures that the database starts in a valid state and ends in a valid state, preserving all defined rules, constraints, and relationships during a transaction. In simple terms, transactions can only bring the database from one valid state to another.

Example of Consistency:
In a university database, a transaction might involve enrolling a student in a course. If there's a rule that students cannot enroll in more than five courses, consistency ensures that if a student attempts to enroll in a sixth course, the transaction will be **rejected**, preserving the validity of the database.

Isolation: Ensuring Transaction Independence

Isolation ensures that multiple transactions happening simultaneously do not interfere with each other. Each transaction must be executed as if it is the only transaction being processed by the database, regardless of other concurrent transactions. The outcome of each transaction should not be visible to others until it has been fully committed.

Example of Isolation:
Imagine two bank customers, Alice and Bob, trying to transfer money from their accounts at the same time. Isolation ensures that the transactions do not interfere with each other. One transaction's operations must not be visible to the other until it is committed, even if they are accessing the same accounts.

Durability: Permanent Effects of a Committed Transaction

Durability ensures that once a transaction has been **committed**, its changes are permanent, even in the event of a system crash or power failure. The database guarantees that committed transactions will persist in storage, and the data will not be lost.

Example of Durability:
After a customer successfully makes an online purchase, the transaction is committed. Even if the system crashes immediately after the transaction is committed, the purchase details will be saved and can be retrieved once the system is back online.

CHAPTER 3: PRACTICAL APPLICATION OF ACID PROPERTIES

Example Scenario: Online Shopping System

Let's consider a **real-world scenario** of an online shopping system where a customer places an order for a product. This system involves multiple operations, such as updating the inventory, charging the customer's payment method, and generating an order confirmation. Here's how ACID properties would be applied in this case:

1. **Atomicity**: The transaction includes multiple operations: deducting the quantity of the product, charging the customer's credit card, and creating an order. If one of these operations fails (e.g., the payment is declined), the entire transaction is **rolled back**, and no operation is completed. This ensures the customer is not charged and the product remains in stock.

2. **Consistency**: The transaction adheres to business rules, such as ensuring a product's stock does not drop below zero. If the inventory system detects that a product is out of stock, the transaction will be rejected, ensuring that the database does not enter an inconsistent state.

3. **Isolation**: If multiple customers are placing orders simultaneously, the database ensures that each order transaction is processed as if it were the only one happening. One customer's order should not affect another customer's order, even if they are purchasing the same product.

4. **Durability**: Once the order is placed, the transaction's data is **permanently saved** in the database. Even if the server crashes immediately after the order is placed, the system will recover the transaction, and the order will remain in the system once it restarts.

Exercise: Implementing Transactions in SQL

1. Write a **transaction** that deducts money from a customer's account and adds it to another customer's account (e.g., in a banking system).

2. Modify the transaction so that if any part of it fails, the database **rolls back** all changes.

3. Test the **atomicity** of your transaction by simulating a failure during the transfer (e.g., using ROLLBACK in SQL).

4. Add **consistency checks** to ensure that account balances cannot go negative.

Case Study: Transaction Management in a Banking System

Scenario

Consider a **banking system** where money is being transferred between two accounts. A customer may request a transfer of funds from their checking account to their savings account. The transaction needs to ensure that the debit from one account and the credit to another are both completed, and the database must guarantee that these changes are reflected consistently.

Transaction Breakdown

1. **Atomicity**: If the debit operation succeeds but the credit operation fails, the entire transaction will **fail**. No money is transferred, and both accounts remain unchanged. This ensures that the database is **never left in an inconsistent state**.

2. **Consistency**: The transaction ensures that both the checking account and the savings account adhere to business rules (e.g., account balance cannot go negative). If the debit exceeds the available balance in the checking account, the transaction will be rejected.

3. **Isolation**: The transaction should be isolated from other transactions, ensuring that one customer's transfer does not interfere with another customer's transaction. Even if two customers are transferring money simultaneously, their operations should not impact each other.

4. **Durability**: Once the transaction is committed, the changes are **permanent**. Even if the system crashes after the transaction is committed, the debit and credit will persist, and the system will recover correctly.

DEEP DIVE INTO TRANSACTION ISOLATION LEVELS AND THEIR IMPACT ON PERFORMANCE

CHAPTER 1: INTRODUCTION TO TRANSACTION ISOLATION LEVELS

What is Transaction Isolation?

Transaction isolation refers to the degree to which the operations in one transaction are isolated from the operations in other concurrent transactions. In database systems, multiple transactions can occur at the same time, which may lead to conflicts if these transactions affect the same data. Transaction isolation levels are used to define the visibility of one transaction's data modifications to other concurrent transactions.

There are four standard **isolation levels** defined by the SQL standard, and they define how the **changes** made by one transaction are visible to others. The isolation level directly affects both the **performance** of the database system and its ability to handle concurrent operations without data inconsistency. The four isolation levels are:

1. **Read Uncommitted**
2. **Read Committed**
3. **Repeatable Read**
4. **Serializable**

Each isolation level balances the **trade-off** between concurrency (allowing multiple transactions to be executed simultaneously) and

data consistency (ensuring the data is correct and consistent across all transactions). A lower isolation level can improve performance by allowing more concurrency, but it may lead to undesirable side effects like dirty reads, non-repeatable reads, or phantom reads.

CHAPTER 2: UNDERSTANDING THE FOUR ISOLATION LEVELS

1. Read Uncommitted (Lowest Isolation Level)

At the **Read Uncommitted** isolation level, transactions are allowed to **read uncommitted changes** made by other transactions. This means that one transaction can see the intermediate, uncommitted changes made by another transaction. While this can increase concurrency and improve performance, it also introduces the risk of **dirty reads**, where one transaction might read data that another transaction later rolls back.

Example of Read Uncommitted:

Transaction A updates a row in the Orders table but has not yet committed the change.
Transaction B can read the uncommitted row, potentially using incorrect data. If Transaction A later **rolls back** the update, Transaction B will have used invalid data.

Impact on Performance:

- **Increased concurrency** because transactions are not blocked from reading uncommitted data.
- **Higher risk of inconsistency** and **data anomalies** like dirty reads.
- This isolation level is typically used in **situations where performance** is more important than absolute consistency,

such as in **reporting systems** where accuracy may be less critical.

---

2. Read Committed (Standard Isolation Level)

The **Read Committed** isolation level ensures that a transaction can only **read committed data**—that is, data that has been committed by other transactions. This prevents **dirty reads** but does not protect against **non-repeatable reads**, where data read by a transaction might change if another transaction modifies it.

Example of Read Committed:

Transaction A updates a row in the Orders table and commits the changes.
Transaction B can then read the updated data. However, if Transaction B reads the data and then Transaction A modifies it again, the result will not be repeatable.

Impact on Performance:

- **Prevents dirty reads**, ensuring that transactions only read data that has been **committed** by others.
- **Less concurrency** than Read Uncommitted, as transactions may have to wait for other transactions to commit.
- **Vulnerable to non-repeatable reads**, which can lead to inconsistencies when a value is read twice during the same transaction but changes between the reads.
- This level is commonly used in many databases as a balance between **data integrity** and **performance**.

---

3. Repeatable Read (Higher Isolation Level)

The **Repeatable Read** isolation level ensures that if a transaction reads a value, it will always see the same value if it reads it again, even if another transaction changes that value. This prevents **non-repeatable reads** but does not prevent **phantom reads**, where a transaction reads a range of data and sees different sets of rows when the transaction is re-executed due to changes made by other transactions.

Example of Repeatable Read:

Transaction A reads all rows from the Products table where the price is above $100.
Transaction B then inserts a new product with a price above $100.
If Transaction A re-queries the Products table, it will not see the newly inserted row because Repeatable Read ensures the transaction will always see the same data it read at the beginning of the transaction.

Impact on Performance:

- **Prevents dirty reads and non-repeatable reads**, ensuring consistent data visibility.
- **Decreases concurrency** compared to Read Committed because it holds **locks** on the data it reads to ensure the data is not modified by other transactions.
- **Increased locking and potential for blocking** in high-concurrency systems, leading to **performance degradation** in heavy transactional workloads.

---

4. Serializable (Highest Isolation Level)

The **Serializable** isolation level is the strictest, ensuring complete isolation between transactions. It guarantees that the results of a transaction will be the same as if it had been the **only transaction running on the system**. This level prevents all anomalies, including **dirty reads, non-repeatable reads, and phantom reads**. It achieves this by ensuring that transactions are executed in a serializable order, meaning that one transaction must be fully completed before another can start.

Example of Serializable:

Transaction A locks a range of rows in the Orders table. While Transaction A is running, no other transaction can modify or insert new rows within that range. This prevents **phantom reads** and ensures that no other transaction can interfere.

Impact on Performance:

- **Ensures the highest level of consistency** but can lead to **severe performance penalties**.
- **Concurrency is significantly reduced** because transactions must be processed **serially**, which can create contention and lead to **deadlocks** and **blocking**.
- This isolation level is useful for scenarios that require **strict consistency** such as **financial transactions**, where absolute correctness is critical, but it is **not suitable for high-concurrency environments** due to its impact on performance.

CHAPTER 3: PERFORMANCE TRADE-OFFS OF TRANSACTION ISOLATION LEVELS

Impact of Isolation Levels on Performance

The **trade-off** between transaction isolation levels lies in the balance between **data consistency** and **concurrency**. Lower isolation levels allow more **concurrent transactions** but at the cost of possible inconsistencies, while higher isolation levels prioritize **data integrity** but reduce concurrency. Here is a comparison of the **performance implications**:

Isolation Level	**Concurrency**	**Data Consistency**	**Common Use Cases**	**Performance Impact**
Read Uncommitted	High	Low	Reporting, logging	Fast, but prone to dirty reads
Read Committed	Moderate	Moderate	General transactional systems	Balanced, but may have non-repeatable reads
Repeatable Read	Low	High	Banking systems, order systems	Slower, due to increased locking
Serializable	Very Low	Very High	Critical financial systems	Slowest, due to strict locking and serial execution

Performance Considerations

- **Lower Isolation Levels** (Read Uncommitted, Read Committed) improve **concurrency** but may introduce data anomalies like dirty reads or non-repeatable reads. These are suitable for environments where performance is prioritized over absolute consistency, such as reporting systems.

- **Higher Isolation Levels** (Repeatable Read, Serializable) ensure **stronger consistency** and data integrity but at the cost of **reducing concurrency**, leading to performance bottlenecks in high-transaction systems.

Exercise: Exploring Transaction Isolation in SQL

1. Implement a query using each of the four isolation levels in a test database and observe how concurrency and data consistency are impacted.
2. Test how deadlocks are triggered when multiple transactions use the **Serializable** isolation level.
3. Write a report discussing the **trade-offs** between performance and consistency for different transaction isolation levels.

Case Study: Managing Isolation in an E-Commerce System

Scenario

In an e-commerce system, multiple users place orders simultaneously. The database handles transactions for order placement, inventory updates, and payment processing. High concurrency is essential for performance, but **data consistency** must be maintained to ensure accurate order processing. The system must balance **performance** with the need for **data integrity**.

Solution

For **order placement**, the **Read Committed** isolation level is used to ensure that users are not able to read uncommitted data, while allowing for moderate concurrency. For critical transactions, such as **inventory updates**, the **Repeatable Read** isolation level is chosen to avoid non-repeatable reads and ensure that stock quantities are not updated incorrectly. For financial transactions, such as **payment processing**, the **Serializable** isolation level is employed to ensure complete isolation and consistency in transactions involving sensitive financial data.

Outcome

The system **optimized performance** by applying different isolation levels based on the nature of the transaction, balancing **data consistency** and **system throughput**.

PRACTICAL SESSION ON DESIGNING TRANSACTION-BASED SYSTEMS

CHAPTER 1: INTRODUCTION TO TRANSACTION-BASED SYSTEMS

What is a Transaction-Based System?

A **transaction-based system** is a system that performs a sequence of operations as a **single unit of work**, often referred to as a transaction. These systems are designed to handle operations where multiple actions must be completed successfully to ensure the integrity and consistency of the system. Examples of transaction-based systems include **banking systems**, **e-commerce platforms**, **inventory management systems**, and **order processing systems**.

The primary goal of transaction-based systems is to ensure that either all operations of a transaction **succeed** (commit), or none of them are applied (rollback). This is achieved using the **ACID properties** (Atomicity, Consistency, Isolation, and Durability) that guarantee reliability in the database, especially in multi-user and concurrent environments.

Transaction-based systems are essential for any system that requires data consistency and integrity, particularly when dealing with critical operations such as financial transactions, inventory updates, or order processing.

CHAPTER 2: KEY CONCEPTS IN DESIGNING TRANSACTION-BASED SYSTEMS

1. Understanding Transactions in the Context of Databases

In a **transaction-based system**, a transaction can be thought of as a **logical unit of work**. Each transaction is composed of one or more **database operations** (e.g., INSERT, UPDATE, DELETE, SELECT) that need to be completed together to ensure consistency.

For instance, in an online shopping system, when a customer places an order, the following steps could be involved in a transaction:

1. **Decreasing inventory stock** – Ensuring the product is deducted from the available stock.
2. **Processing payment** – Charging the customer's payment method.
3. **Generating an order record** – Saving the order details in the database.

All these actions are part of a **single transaction**. If one action fails (for instance, payment processing fails), the entire transaction is **rolled back**, and the system returns to its original state, ensuring no partial data is committed.

2. The ACID Properties in Action

For a transaction-based system to function correctly, it must adhere to the **ACID properties**:

- **Atomicity** ensures that all operations of a transaction are treated as a single unit. If any part fails, the entire transaction is rolled back.
- **Consistency** ensures that the database is always in a valid state before and after the transaction.
- **Isolation** ensures that concurrent transactions do not affect each other's operations, preventing data anomalies.

- **Durability** guarantees that once a transaction is committed, its effects are permanent, even in case of system crashes.

CHAPTER 3: DESIGNING TRANSACTION-BASED SYSTEMS - STEP-BY-STEP

Step 1: Identifying Business Processes and Transaction Requirements

Before you can design a transaction-based system, you need to understand the **business requirements** and **operations** that need to be supported. For example, if you're designing a **banking system**, you would need to consider operations like:

- **Money transfers** between accounts
- **Deposits** and **withdrawals**
- **Balance inquiries**

Each of these operations requires its own transaction, ensuring that either all steps in the operation succeed, or none do.

Step 2: Defining Transaction Boundaries

A key aspect of designing transaction-based systems is to **define the boundaries of each transaction**. A transaction should include everything that must be executed together to achieve the desired outcome. For example, in a **sales system**, a transaction could involve:

- Updating the customer's order status
- Reducing the inventory count
- Charging the customer's credit card

- Creating a shipment record

Each of these steps is part of a single transaction. If any of them fail, the entire transaction is **rolled back**. Properly defining these boundaries helps ensure data consistency.

Step 3: Handling Concurrent Transactions

In real-world systems, multiple transactions are often executed concurrently. This can lead to problems such as **deadlocks**, **lost updates**, or **dirty reads**. To handle these issues, **isolation levels** are used.

- **Read Uncommitted** allows maximum concurrency but is prone to dirty reads.
- **Read Committed** prevents dirty reads but still allows non-repeatable reads.
- **Repeatable Read** ensures that data read by a transaction cannot change during the transaction, but still allows phantom reads.
- **Serializable** is the highest isolation level, preventing all concurrency anomalies but decreasing performance.

Designing a transaction-based system requires choosing the appropriate isolation level to balance **data integrity** and **concurrency**.

CHAPTER 4: DESIGNING AND IMPLEMENTING A TRANSACTION-BASED SYSTEM

1. Implementing Transaction Management in SQL

Here's an example of how to design a transaction for an **e-commerce system** where a customer places an order:

```
BEGIN TRANSACTION;


-- Step 1: Reduce inventory

UPDATE Products

SET Stock = Stock - 1

WHERE ProductID = 101;


-- Step 2: Charge customer's credit card

UPDATE Accounts

SET Balance = Balance - 50

WHERE CustomerID = 123;


-- Step 3: Create an order record

INSERT INTO Orders (CustomerID, ProductID, Quantity, OrderDate)

VALUES (123, 101, 1, CURRENT_TIMESTAMP);


-- Step 4: Commit transaction if all steps succeed

COMMIT;
```

In this example, all the operations (updating inventory, charging the customer, creating the order) are part of a single transaction. If one step fails, the **rollback** ensures that no partial changes are saved.

2. Handling Errors and Rollbacks

In case any part of the transaction fails, we use **ROLLBACK** to undo all changes made so far, ensuring the database remains in a consistent state. For example:

```
BEGIN TRANSACTION;


-- Step 1: Try to update inventory

BEGIN TRY

   UPDATE Products

   SET Stock = Stock - 1

   WHERE ProductID = 101;


   -- Step 2: Try to charge the customer's account

   UPDATE Accounts

   SET Balance = Balance - 50

   WHERE CustomerID = 123;


   -- Step 3: Insert order record

   INSERT INTO Orders (CustomerID, ProductID, Quantity,
OrderDate)
```

```
    VALUES (123, 101, 1, CURRENT_TIMESTAMP);


    COMMIT;  -- If all steps succeed
END TRY
BEGIN CATCH
    ROLLBACK;  -- If any step fails, rollback all changes
    PRINT 'Transaction failed. Changes rolled back.';
END CATCH;
```

This ensures that **partial transactions** are never saved, maintaining **data integrity**.

Chapter 5: Testing and Performance Considerations in Transaction-Based Systems

1. Testing Transactions for Integrity and Reliability

When designing a transaction-based system, it's crucial to test how well the transactions handle **edge cases** and **failures**:

- Test for **deadlocks**: Simulate multiple transactions accessing the same data and ensure the system handles conflicts properly.
- Test for **rollbacks**: Ensure that transactions are properly rolled back in case of errors, and no partial changes are saved.
- Test for **high concurrency**: Check how the system performs under a high load of concurrent transactions.

2. Optimizing for Performance

While transaction integrity is critical, performance must also be considered. Some strategies to optimize performance in transaction-based systems include:

- **Indexing frequently queried columns** to speed up transaction lookups.
- **Batching transactions** where possible to reduce the overhead of committing individual transactions.
- Using appropriate **isolation levels** to balance data consistency and concurrency.

Exercise: Designing a Transaction-Based System

1. Design a **transaction-based system** for an online banking system. Your system should include transactions for transferring funds, withdrawing money, and checking balances.
2. Implement the transactions using SQL queries and ensure that **ACID properties** are adhered to.
3. Write a report on the **isolation levels** you would choose for each transaction, considering performance and consistency.
4. Test your transaction system with multiple concurrent users and simulate errors to check if transactions are properly rolled back.

Case Study: E-Commerce Transaction System

Scenario

An e-commerce platform needs to handle user orders. The system should:

- Decrease product inventory when an order is placed.
- Charge the customer's credit card.
- Generate a shipment record for each order.
- Ensure all these operations are executed as part of a single transaction.

Solution

The e-commerce system is designed using **transaction-based principles**. Each order consists of multiple actions (inventory update, payment, and shipment) that must either all succeed or all fail. The system uses **ACID-compliant transactions** and **Read Committed isolation level** to ensure both **data integrity** and **reasonable performance**.

By handling concurrent orders with careful attention to isolation and transaction management, the platform ensures no order is processed partially, maintaining **transactional integrity** and **customer satisfaction**.

LOCKING PROTOCOLS, OPTIMISTIC VS. PESSIMISTIC CONCURRENCY

CHAPTER 1: INTRODUCTION TO CONCURRENCY CONTROL IN DATABASES

What is Concurrency Control?

In database systems, **concurrency control** refers to the management of simultaneous operations on the database without conflicting with each other. When multiple transactions are executed concurrently, the database management system (DBMS) must ensure that the results of these transactions are consistent and reliable. **Concurrency control mechanisms** prevent issues such as **data corruption**, **dirty reads**, and **lost updates** by ensuring that multiple transactions do not interfere with each other.

Concurrency control is essential for systems where multiple users or applications interact with the database simultaneously. The two primary types of concurrency control mechanisms are:

1. **Locking protocols**
2. **Optimistic concurrency control**

These methods dictate how transactions are allowed to access and modify data, ensuring the integrity of the database while maintaining performance.

CHAPTER 2: LOCKING PROTOCOLS IN DATABASE SYSTEMS

What Are Locking Protocols?

Locking protocols are used to prevent conflicts between concurrent transactions by **locking** the data that a transaction is accessing. When a transaction locks a piece of data, other transactions are restricted from modifying or reading that data until the lock is released. Locking ensures that only one transaction can access a piece of data at a time, preventing anomalies such as **dirty reads** or **lost updates**.

There are several types of locks and locking protocols used in database systems, including:

- **Shared Locks (S-locks)**
- **Exclusive Locks (X-locks)**
- **Intention Locks**
- **Two-Phase Locking (2PL)**

Shared vs. Exclusive Locks

- **Shared Lock (S-lock)**: Allows multiple transactions to read the data simultaneously but prevents any transaction from modifying the data.
- **Exclusive Lock (X-lock)**: Grants a transaction complete access to modify the data, preventing other transactions from reading or modifying it until the lock is released.

For example, if two transactions are trying to read and write to the same row in a table:

- If Transaction A holds an **exclusive lock** on the row, Transaction B must wait until the lock is released.
- If Transaction A only holds a **shared lock**, Transaction B can also read the row, but cannot modify it until Transaction A releases the lock.

Two-Phase Locking (2PL)

One of the most widely used locking protocols is **Two-Phase Locking (2PL)**. It guarantees **serializability**, the highest level of isolation, by requiring that:

1. A transaction must acquire all its locks before releasing any.
2. Once a transaction releases a lock, it cannot acquire any new locks.

Example:
In a bank system, if one transaction is transferring money from one account to another, it will lock both accounts during the transfer. The locks will be released once the transaction is completed, ensuring no other transactions can interfere.

Advantages and Disadvantages of Locking Protocols

- **Advantages**:
  - Ensures **data consistency** by preventing conflicting operations.
  - Provides **strong isolation** between transactions.
- **Disadvantages**:
  - **Blocking**: Transactions might wait for locks to be released, leading to delays.
  - **Deadlocks**: Multiple transactions can wait on each other to release locks, causing a situation where none of the transactions can proceed.

CHAPTER 3: OPTIMISTIC CONCURRENCY CONTROL

What is Optimistic Concurrency Control?

Optimistic concurrency control (OCC) is an approach that assumes that conflicts between transactions are rare. Instead of locking data, OCC allows transactions to execute without acquiring locks and checks for conflicts only when the transaction is ready to commit. If a conflict is detected at commit time, the transaction is rolled back and must be retried.

OCC relies on the principle that most transactions will not interfere with each other, so it is typically used in environments where:

- **Transactions are short-lived**.
- **Data contention is low** (few transactions access the same data at the same time).

How Optimistic Concurrency Works

The process of OCC typically involves the following three phases:

1. **Read Phase**: The transaction reads the data and works on it without acquiring locks.
2. **Validation Phase**: Before committing the transaction, the DBMS checks if any other transaction has modified the data that this transaction has worked on.
3. **Write Phase**: If no conflicts are found during the validation phase, the transaction writes its changes to the database. If a conflict is found, the transaction is rolled back and has to restart.

Example:
In an online shopping system, when two users are attempting to purchase the same product, the system allows both users to add the item to their cart. At the time of checkout, the system checks if the

product is still available and performs the necessary update if no other transaction has purchased it. If another transaction has purchased the item, the transaction is **rejected** and the user is prompted to try again.

Advantages and Disadvantages of Optimistic Concurrency

- **Advantages**:
  - Allows for higher **concurrency** because transactions are not blocked by locks.
  - **Better performance** in systems with low contention or short transactions.
- **Disadvantages**:
  - Transactions may need to be rolled back, causing **higher overhead** in cases of conflicts.
  - Not suitable for high-contention environments, as the number of conflicts increases, reducing the benefits of OCC.

CHAPTER 4: PESSIMISTIC CONCURRENCY CONTROL

What is Pessimistic Concurrency Control?

Pessimistic concurrency control assumes that conflicts are likely to happen. In this model, data is **locked** when a transaction begins and remains locked until the transaction is committed or rolled back. The database prevents other transactions from accessing the locked data, ensuring that no conflicting updates happen.

Pessimistic concurrency is typically used in environments where:

- **Data contention is high** (many transactions try to access the same data).
- **Transaction duration is long**, and it is critical to ensure that no other transaction modifies the data while the transaction is in progress.

How Pessimistic Concurrency Works

In pessimistic concurrency control, the database system will automatically acquire **locks** for the data that a transaction reads or modifies. Other transactions must wait for the locks to be released before they can access the same data.

Example:
In a bank system, when a transaction is transferring funds between two accounts, it locks the involved accounts to prevent other transactions from modifying the accounts while the transfer is in progress. The system ensures that no other transactions can access the data during this process, providing strong isolation between transactions.

Advantages and Disadvantages of Pessimistic Concurrency

- **Advantages**:
    - Guarantees **data consistency** and prevents conflicts such as dirty reads, lost updates, and non-repeatable reads.
    - **More predictable behavior** in high-contention environments.
- **Disadvantages**:
    - **Reduced concurrency** due to locking, potentially leading to performance bottlenecks.

- **Deadlocks** can occur if multiple transactions are waiting for locks held by others, leading to a situation where no transaction can proceed.

CHAPTER 5: COMPARING OPTIMISTIC AND PESSIMISTIC CONCURRENCY CONTROL

Performance Trade-offs

The choice between **optimistic** and **pessimistic concurrency** depends on the specific needs of the application:

- **Optimistic concurrency control** is ideal for environments with **low contention** and **short transactions**. It allows for higher concurrency but risks **rollback** when conflicts occur.
- **Pessimistic concurrency control** is best for high-contention environments where data consistency is crucial. However, it results in **lower concurrency** and may cause **deadlocks**.

Example Comparison

Consider an **inventory management system**:

- With **optimistic concurrency**, multiple employees can simultaneously update inventory levels without locking data. However, if two employees attempt to update the same inventory record, one will be rolled back.
- With **pessimistic concurrency**, when an employee locks the inventory record, others must wait until the lock is released, ensuring no conflicts. However, this could result in slower system performance, especially during peak hours when many employees try to update inventory records.

Exercise: Implementing Concurrency Control

1. Create an inventory management scenario with both **optimistic** and **pessimistic concurrency control** mechanisms implemented.
2. Test the performance of both approaches under different levels of data contention.
3. Analyze the benefits and trade-offs in each case and recommend the best approach for different types of applications.

Case Study: Optimistic vs. Pessimistic Concurrency in Online Booking System

Scenario

In an online booking system, multiple users are attempting to book the same hotel room.

- **Optimistic concurrency** allows users to reserve rooms without locking the data initially. At checkout, if another user has already booked the room, the transaction is rejected and the user is prompted to choose another room.
- **Pessimistic concurrency** locks the room as soon as a user starts the booking process, ensuring that no other users can reserve the room until the current transaction is completed.

Outcome

- **Optimistic concurrency** allowed for better system performance when there were fewer conflicts but resulted in higher rejection rates.

- **Pessimistic concurrency** provided more consistent availability but reduced system throughput due to the locking mechanism.

DEADLOCK DETECTION, RESOLUTION, AND RECOVERY STRATEGIES

CHAPTER 1: UNDERSTANDING DEADLOCKS IN DATABASE SYSTEMS

What is a Deadlock?

A **deadlock** is a situation in a database system where two or more transactions are unable to proceed because each is waiting for the other to release resources (such as data, locks, or other system resources). Essentially, the system becomes **stuck**, as the transactions cannot continue without intervention. Deadlocks pose a significant challenge in systems with **concurrent transactions**, especially in databases where many transactions might attempt to access and modify the same data concurrently.

Deadlocks can occur in any system that involves resource allocation, but they are most common in systems using **locking protocols** for concurrency control. The problem arises when transactions hold locks on some resources and simultaneously request locks on other resources, which are held by other transactions. This cycle of waiting can result in a state where no transaction can complete its operation, thus leading to a deadlock.

CHAPTER 2: CONDITIONS FOR DEADLOCK

The Four Necessary Conditions for a Deadlock

A deadlock can only occur if the following four **necessary conditions** are simultaneously present:

1. **Mutual Exclusion**: At least one resource (e.g., a lock on a data item) must be held in a non-shareable mode. Only one transaction can access the resource at a time.
2. **Hold and Wait**: A transaction that is holding one resource is waiting to acquire additional resources that are currently being held by other transactions.
3. **No Preemption**: Once a resource is allocated to a transaction, it cannot be forcibly taken away from the transaction. It can only be released voluntarily.
4. **Circular Wait**: A set of transactions must exist such that each transaction in the set is waiting for a resource that another transaction in the set is holding, forming a cycle of waiting.

When all four of these conditions are met, the system is in a **deadlock state**, where transactions are blocked and cannot proceed.

CHAPTER 3: DEADLOCK DETECTION

Deadlock Detection Approach

Deadlock detection is the process of identifying whether a set of transactions is in a deadlock state. It involves periodically checking the system to see if there are any transactions that cannot proceed due to circular dependencies in resource allocation. The primary methods for detecting deadlocks are:

1. **Wait-for Graph**:
   A **wait-for graph** is a directed graph where each node represents a transaction, and an edge from transaction T1 to transaction T2 indicates that T1 is waiting for a resource that T2 holds. A deadlock exists if there is a **cycle** in the graph.

Steps to create a wait-for graph:

- **Create a node** for each transaction.
- **Create a directed edge** from one transaction to another if the first transaction is waiting for a resource held by the second.
- **Detect cycles**: If any cycle is found in the graph, a deadlock exists.

Example:

- Transaction T1 holds a lock on resource R1 and is waiting for R2.
- Transaction T2 holds a lock on R2 and is waiting for R1. The wait-for graph would contain a cycle, indicating a deadlock.

2. **Transaction Timeout**:
   Another simple method is to set a **timeout** for each transaction. If a transaction does not complete within a certain time limit, the system assumes that the transaction might be involved in a deadlock and performs checks or intervention.

Example:
If Transaction T1 is waiting for a lock but hasn't acquired it after a pre-defined period, the system will check if a deadlock exists by analyzing the wait-for graph. If so, appropriate actions like rollback or timeout are taken.

Advantages of Deadlock Detection:

- It's proactive and doesn't require additional restrictions on transactions.

- It's easy to implement and efficient in systems where deadlocks occur infrequently.

CHAPTER 4: DEADLOCK RESOLUTION

Deadlock Resolution Strategies

Once a deadlock is detected, it must be resolved to allow the transactions to proceed. There are two common approaches to deadlock resolution:

1. **Transaction Rollback (Abort)**:
   One or more transactions involved in the deadlock cycle are chosen to be **rolled back**. The system then frees the resources held by the aborted transaction, allowing the other transactions to proceed. There are two main strategies for choosing which transaction(s) to abort:

   - **Random Rollback**: Select a transaction arbitrarily and roll it back.
   - **Victim Selection**: Rollback the transaction that causes the least disruption, typically by considering factors such as:
     - Transaction **age** (younger transactions may be chosen for rollback).
     - The number of **resources held** by the transaction (a transaction holding fewer resources might be chosen).
     - **Rollback cost** (transactions that are simpler to roll back might be selected).

2. **Resource Preemption**:
   In this strategy, the system forcibly takes resources away from one or more transactions to break the deadlock. Preemption can lead to **starvation** (if the transaction is repeatedly preempted), but it can be used in high-priority applications. Preempted transactions are either restarted or rescheduled after some time.

Example:
Consider three transactions (T1, T2, and T3) involved in a deadlock:

- T1 holds R1 and requests R2, which is held by T2.
- T2 holds R2 and requests R3, which is held by T3.
- T3 holds R3 and requests R1, which is held by T1.

The system can select T1 to **rollback** and free R1, allowing T2 and T3 to proceed.

CHAPTER 5: DEADLOCK RECOVERY

Deadlock Recovery Mechanisms

After resolving a deadlock, the system needs to ensure that all database states are **consistent**, and the effects of the rolled-back transactions are properly handled. This process is known as **recovery**. The following recovery strategies are commonly employed:

1. **Transaction Restart**:
   After a transaction is rolled back, it can be **restarted** from the point where it last committed or from the beginning, depending on the nature of the transaction and its dependency on other transactions.

Example:
After rolling back Transaction T1, the system can restart it by re-executing the same operations, ensuring it doesn't violate consistency.

2. **Logging and Undo/Redo**:
   A **log-based recovery** system maintains a **log** of all changes made by transactions. If a transaction is aborted, the system can **undo** the changes made by that transaction. Similarly, if a transaction is completed, it can be **redone** to ensure the changes are reflected.

Example:

- **Undo**: If Transaction T1 updates a record but is rolled back, the system will undo all changes made by T1.
- **Redo**: If T1 is committed, its actions will be redone to ensure durability.

Exercise: Implementing Deadlock Detection and Resolution

1. **Create a simple transaction scenario** involving two transactions that might cause a deadlock.
2. **Simulate deadlock detection** using a **wait-for graph** approach.
3. **Apply deadlock resolution** by rolling back one of the transactions.
4. **Test recovery** by re-executing the rolled-back transaction and ensuring consistency is maintained.

Case Study: Deadlock Detection and Resolution in a Reservation System

Scenario

In a hotel reservation system, multiple users may attempt to reserve the same rooms simultaneously. Each transaction involves locking the rooms, updating availability, and processing payments. Deadlocks might occur if two users try to book the same rooms at the same time.

Solution

1. **Deadlock Detection**: The system detects deadlocks using a wait-for graph, where each transaction (booking request) is checked for circular dependencies.
2. **Deadlock Resolution**: The system chooses the transaction with the least amount of resources held as the victim for rollback.
3. **Recovery**: After a rollback, the user is prompted to retry the booking, ensuring system consistency and user satisfaction.

Hands-on Exercises Simulating Concurrent Transactions

Chapter 1: Introduction to Concurrent Transactions

What Are Concurrent Transactions?

In a database system, **concurrent transactions** refer to multiple transactions being executed simultaneously. The goal of concurrency control is to allow multiple transactions to access and modify data concurrently while ensuring that the results are consistent and accurate. However, simultaneous access to the same data can result in conflicts, such as **dirty reads**, **lost updates**, or **non-repeatable reads**, which can violate **ACID properties** (Atomicity, Consistency, Isolation, Durability).

In real-world systems, such as **banking**, **e-commerce**, or **inventory management**, transactions often need to occur concurrently to improve **performance** and **system throughput**. Handling concurrent transactions effectively is crucial for ensuring that database systems can scale and remain **responsive** under high workloads.

This chapter will focus on **simulating concurrent transactions** through hands-on exercises, helping you understand how different transaction isolation levels and concurrency control mechanisms affect the integrity of the database.

Chapter 2: Setting Up the Environment for Concurrent Transactions Simulation

Prerequisites for Simulation

Before diving into the hands-on exercises, ensure you have the following tools and environment ready to simulate concurrent transactions:

1. **Database Management System (DBMS)**:
   - MySQL, PostgreSQL, or any relational database system of your choice.
   - Ensure that **transaction management** features such as **ACID compliance** and **isolation levels** are available.
2. **Sample Database**:
   - For this exercise, we will use a **simple bank database** consisting of two tables: Accounts and Transactions.
   - The Accounts table will store customer account information, and the Transactions table will store all transaction details.
3. **Transaction Simulation Script**:
   - Create a script to simulate multiple concurrent transactions, such as deposits and withdrawals, on the same account.

Creating Sample Database Tables

```
CREATE TABLE Accounts (

  AccountID INT PRIMARY KEY,

  AccountHolder VARCHAR(100),
    Balance DECIMAL(10, 2)
);

CREATE TABLE Transactions (
    TransactionID INT AUTO_INCREMENT PRIMARY KEY,
    AccountID INT,
    Amount DECIMAL(10, 2),
    TransactionType VARCHAR(20),
    TransactionDate TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)
);

-- Insert some sample accounts
INSERT INTO Accounts (AccountID, AccountHolder, Balance)
VALUES
(1, 'Alice', 5000.00),
(2, 'Bob', 3000.00);
```

Chapter 3: Simulating Concurrent Transactions

Exercise 1: Simulating Two Concurrent Transactions with Different Isolation Levels

In this exercise, we will simulate two transactions occurring concurrently. Transaction 1 will withdraw $100 from Alice's account, while Transaction 2 will deposit $200 into Bob's account. The goal is to observe how different isolation levels impact the visibility of data changes between the two transactions.

Step 1: Start Transaction 1 (Withdraw $100 from Alice's account)

START TRANSACTION;

UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;

SELECT * FROM Accounts WHERE AccountID = 1;

Step 2: Start Transaction 2 (Deposit $200 into Bob's account)

START TRANSACTION;

UPDATE Accounts SET Balance = Balance + 200 WHERE AccountID = 2;

SELECT * FROM Accounts WHERE AccountID = 2;

Now, you will observe the effects of running these two transactions concurrently with **different isolation levels**.

Step 3: Test with Different Isolation Levels

- **Read Uncommitted**: This isolation level allows **dirty reads**. One transaction may read uncommitted changes made by another transaction.
- SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
- **Read Committed**: This isolation level prevents **dirty reads** but allows **non-repeatable reads**.

- SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
- **Repeatable Read**: This isolation level prevents both **dirty reads** and **non-repeatable reads**, ensuring that the data read by a transaction cannot change during the transaction.
- SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
- **Serializable**: This isolation level ensures that the transactions are executed **serially**, avoiding all concurrency issues but possibly reducing performance.
- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Step 4: Observe the Results

- For **Read Uncommitted**, you may see one transaction's uncommitted changes visible to the other.
- With **Read Committed**, you will see that Transaction 2 will not read uncommitted data from Transaction 1.
- **Repeatable Read** ensures that the data read by one transaction will remain consistent during the transaction.
- **Serializable** will ensure that transactions run in sequence, preventing any concurrency conflicts.

Exercise 2: Handling Deadlocks with Concurrent Transactions

Deadlocks occur when two or more transactions hold locks on resources and are waiting for each other to release resources, leading to a situation where none of the transactions can proceed.

Step 1: Transaction 1 (Lock Account 1)

START TRANSACTION;

UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1; -- Lock Account 1

Step 2: Transaction 2 (Lock Account 2)

START TRANSACTION;

UPDATE Accounts SET Balance = Balance - 50 WHERE AccountID = 2; -- Lock Account 2

Step 3: Transaction 1 Attempts to Lock Account 2

UPDATE Accounts SET Balance = Balance + 50 WHERE AccountID = 2; -- Transaction 1 waits for Transaction 2

Step 4: Transaction 2 Attempts to Lock Account 1

UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 1; -- Transaction 2 waits for Transaction 1

At this point, both transactions are **waiting on each other**, creating a **deadlock**. The database system will detect the deadlock and choose one of the transactions to be rolled back to break the cycle.

Step 5: Deadlock Detection and Resolution

Once the deadlock is detected, one of the transactions will be **rolled back** to free up the locked resources. The system may choose a victim based on factors such as transaction age or the number of resources held by the transactions. Once the transaction is rolled back, the other transaction can proceed.

CHAPTER 4: PERFORMANCE CONSIDERATIONS WITH CONCURRENT TRANSACTIONS

Exercise 3: Simulating High Concurrency and Performance Bottlenecks

In this exercise, we will simulate high concurrency by executing multiple transactions simultaneously. The goal is to observe how **isolation levels**, **lock contention**, and **transaction delays** affect performance.

Step 1: Create Multiple Transactions

Simulate multiple transactions where each transaction is modifying a different part of the data concurrently.

START TRANSACTION;

UPDATE Accounts SET Balance = Balance + 50 WHERE AccountID = 1;

COMMIT;

Step 2: Test Performance Under Concurrent Load

Run multiple instances of the above transaction simultaneously, modifying different accounts or the same account, depending on the scenario.

- **Scenario 1: Different Accounts** – Run the transactions with different account IDs to minimize lock contention.
- **Scenario 2: Same Account** – Run transactions that modify the same account to simulate **high contention** and **lock conflicts**.

Observe how the system handles concurrent transactions under different isolation levels. The **Serializable** isolation level will likely reduce performance due to the strict locking, whereas **Read Uncommitted** will allow more concurrency but increase the risk of data anomalies.

CHAPTER 5: ANALYZING THE RESULTS AND LESSONS LEARNED

Key Takeaways from Simulating Concurrent Transactions

1. **Isolation Levels and Their Impact**: The **higher the isolation level**, the more it impacts concurrency. While **Serializable** ensures the highest level of data consistency, it reduces performance due to more restrictive locking. Conversely, **Read Uncommitted** allows for higher concurrency but risks **dirty reads** and data inconsistency.
2. **Deadlock Handling**: Deadlocks are inevitable in environments with high concurrency, but **deadlock detection** and **resolution strategies** (such as **transaction rollback**) allow the system to recover and maintain database consistency.
3. **Performance Bottlenecks**: When simulating high concurrency, we observe that **lock contention** can severely degrade performance, especially when many transactions are trying to access the same data. **Optimistic concurrency control** can help mitigate these bottlenecks in certain scenarios.

Exercise Reflection

- What impact did the different isolation levels have on the concurrency of your system?
- How did deadlock resolution strategies affect the behavior of your transactions?
- What were the performance implications of running multiple concurrent transactions with different resource contention scenarios?

Assignment Solution: Simulating a Multi-User Transaction Environment Using Case Studies

Objective:

In this assignment, we will simulate a multi-user transaction environment where multiple transactions are executed concurrently. We will analyze transaction schedules and implement **concurrency control** measures to ensure **data integrity**.

Step 1: Understanding the Case Study and Setup

We will use the case study of a **banking system** where two users, Alice and Bob, are attempting to perform concurrent transactions on their bank accounts. Alice wants to **withdraw money** from her account, and Bob wants to **deposit money** into his account. Both transactions involve reading and updating account balances, so we need to simulate the environment with **concurrency control** measures.

Database Schema for Bank System

Let's create the necessary **tables** for the simulation. We will have two main tables: Accounts and Transactions.

```
CREATE TABLE Accounts (

   AccountID INT PRIMARY KEY,
    AccountHolder VARCHAR(100),

    Balance DECIMAL(10, 2)

);


CREATE TABLE Transactions (

    TransactionID INT AUTO_INCREMENT PRIMARY KEY,

    AccountID INT,

    TransactionType VARCHAR(20),

    Amount DECIMAL(10, 2),

    TransactionDate TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);


-- Inserting initial account data for Alice and Bob

INSERT INTO Accounts (AccountID, AccountHolder, Balance) VALUES

(1, 'Alice', 1000.00),

(2, 'Bob', 500.00);
```

Step 2: Simulating Concurrent Transactions

We will simulate the following two transactions in a **multi-user environment**:

1. **Transaction 1**: Alice attempts to withdraw $200 from her account.
2. **Transaction 2**: Bob attempts to deposit $300 into his account.

Transaction 1: Alice's Withdrawal

```
-- Transaction 1: Alice withdraws $200

START TRANSACTION;


UPDATE Accounts

SET Balance = Balance - 200

WHERE AccountID = 1;


-- Check balance after withdrawal

SELECT * FROM Accounts WHERE AccountID = 1;


COMMIT;
```

Transaction 2: Bob's Deposit

```
-- Transaction 2: Bob deposits $300

START TRANSACTION;


UPDATE Accounts

SET Balance = Balance + 300
```

```
WHERE AccountID = 2;


-- Check balance after deposit

SELECT * FROM Accounts WHERE AccountID = 2;


COMMIT;
```

Step 3: Analyzing the Transaction Schedule

To simulate the **multi-user transaction environment**, we can interleave the operations of these two transactions. The schedule will involve both transactions executing at the same time, which can lead to potential issues like **lost updates** or **inconsistent data** if not properly controlled.

Interleaving Example (Without Concurrency Control):

Here's one possible interleaving of these two transactions:

1. **T1** (Alice's withdrawal) begins and locks the Accounts table for Alice's account (AccountID = 1).
2. **T2** (Bob's deposit) begins and locks the Accounts table for Bob's account (AccountID = 2).
3. **T1** updates the balance for Alice's account to $800 (1000 - 200).
4. **T2** updates the balance for Bob's account to $800 (500 + 300).
5. Both transactions are committed.

Result: In this scenario, the operations can be correctly completed, as the transactions operate on different accounts. However, in scenarios where the same data is being modified, **concurrency control measures** are necessary to prevent issues such as **dirty reads**, **lost updates**, or **non-repeatable reads**.

Step 4: Implementing Concurrency Control Measures

Concurrency Control Using Locking Protocols

In order to ensure that transactions do not interfere with each other and maintain **data integrity**, we can implement **locking protocols**. Here we will use **Two-Phase Locking (2PL)**, a widely used protocol for ensuring **serializability**. The 2PL protocol works by acquiring all necessary locks before releasing any locks.

Step 1: Implementing Locks

- **Shared locks (S-locks)** are used when data is being read.
- **Exclusive locks (X-locks)** are used when data is being updated or modified.

Example Using Two-Phase Locking (2PL):

1. **Transaction 1: Alice's Withdrawal**
2. -- Acquire an exclusive lock on Alice's account before updating balance
3. START TRANSACTION;
4.
5. SELECT * FROM Accounts WHERE AccountID = 1 FOR UPDATE; -- Lock account for update

6.
7. UPDATE Accounts
8. SET Balance = Balance - 200
9. WHERE AccountID = 1;
10.
11. -- Check balance after withdrawal
12. SELECT * FROM Accounts WHERE AccountID = 1;
13.
14. COMMIT;
15. **Transaction 2: Bob’s Deposit**
16. -- Acquire an exclusive lock on Bob's account before updating balance
17. START TRANSACTION;
18.
19. SELECT * FROM Accounts WHERE AccountID = 2 FOR UPDATE; -- Lock account for update
20.
21. UPDATE Accounts
22. SET Balance = Balance + 300
23. WHERE AccountID = 2;
24.
25. -- Check balance after deposit

26. SELECT * FROM Accounts WHERE AccountID = 2;

27.

28. COMMIT;

How Two-Phase Locking Works:

- The transactions **lock** the rows (using FOR UPDATE) when they need to update the data.
- Once the transaction is finished, the locks are **released** by the **COMMIT** statement.
- With 2PL, no new locks are acquired after releasing any lock, ensuring that transactions do not conflict with each other.

Step 5: Deadlock Detection and Resolution

In a multi-user environment, deadlocks can occur if two or more transactions are waiting for resources that are locked by each other. To avoid this, we need to **detect and resolve deadlocks**.

Deadlock Detection:

Deadlock detection involves analyzing the **wait-for graph**, where each node represents a transaction and an edge from T1 to T2 indicates that T1 is waiting for a resource held by T2. A **cycle** in the graph indicates a deadlock.

Deadlock Detection Example:

1. Transaction 1 locks Account 1 and waits for Account 2 to be unlocked.
2. Transaction 2 locks Account 2 and waits for Account 1 to be unlocked.

3. The system detects the cycle and selects one of the transactions (usually the one holding fewer resources) to **abort and rollback** to break the deadlock.

STEP 6: TESTING CONCURRENCY CONTROL IN THE BANK SYSTEM

To test the system, simulate concurrent transactions with different isolation levels:

1. **Read Committed Isolation**: This ensures that dirty reads are avoided, but non-repeatable reads can still occur. It's suitable for cases where performance is more important than absolute consistency.
2. **Serializable Isolation**: This is the highest isolation level and ensures that transactions are executed serially, avoiding any concurrency issues at the cost of performance.

Testing Scenarios:

- **Scenario 1**: Run two concurrent transactions that modify different accounts (Alice and Bob). Both should complete without deadlock or data inconsistency.
- **Scenario 2**: Simulate deadlock by having two transactions wait on each other to access the same account. Observe how the system detects and resolves the deadlock.

STEP 7: ANALYZING THE RESULTS AND CONCLUSION

Analysis:

- After running the transactions with **Two-Phase Locking (2PL)**, we ensured that no transaction could interfere with another when modifying the same data, and deadlocks were detected and resolved automatically.

- The **Serializable isolation level** ensured that no data anomalies occurred, but performance was slightly impacted due to the restrictive locks.
- In the **Read Committed** scenario, transactions could read committed data, but potential issues like **non-repeatable reads** were observed in cases where a transaction's data was modified after it was read.

CONCLUSION:

- **Locking mechanisms** like **Two-Phase Locking** and **deadlock detection** ensure **data integrity** in a multi-user environment.
- The **Serializable isolation level** provides the highest consistency but can lead to **lower concurrency** and **performance bottlenecks**.
- **Concurrency control** methods should be chosen based on system requirements: **optimizing performance** for high-concurrency systems or **maximizing consistency** in mission-critical applications.