**Assignment 3:**

Explain the ACID properties of a transaction in your own words. Write SQL statements to simulate a transaction that includes locking and demonstrate different isolation levels to show concurrency control.

**ANS:**

ACID Properties : ACID stands for Atomicity, Consistency, Isolation, and Durability. These properties ensure reliable processing of database transactions.

**Atomicity**: Atomicity makes all operations in a transaction successfully. If any process fails, the whole transaction fails and returning the system to its original state.

Example: In E-commerce website. If a user wants to buy a product and makes transaction. But product is out of stock or payment fails. At that time transaction will be fail and none of the changes will applied.

**Consistency**: Database always in a consistent state even after multiple transactions by consistency. To maintain integrity in database.

Example: In E-commerce websites, stock doesn’t goes into negative numbers or below 0 and also valid payment entries.

**Isolation**: Isolation ensures that concurrent transactions execute independently of each other. Intermediate states of a transaction are invisible to other transactions

Example: While one transaction is updating a record, other transactions cannot access this record in an inconsistent state.

**Durability**: It ensures that once a transaction is committed, it will remain so, even if the system failure.

Example: Once, transaction (such as placing an order) is committed, the changes are permanently saved.

**SQL Statements to Simulate a Transaction with Locking and Isolation Levels**

Setting Up a Sample Database: Create a simple database with a bank\_accounts table

CREATE TABLE bank\_accounts (

account\_id INT PRIMARY KEY,

account\_holder VARCHAR(40),

balance DECIMAL(10, 2) );

INSERT INTO bank\_accounts (account\_id, account\_holder, balance) VALUES (49, 'Vinay, 1000.00);

INSERT INTO bank\_accounts (account\_id, account\_holder, balance) VALUES (50, Nikhil, 500.00);

Simulating a Transaction with Locking : To simulate a transaction, we use BEGIN TRANSACTION and COMMIT or ROLLBACK.

BEGIN TRANSACTION;

-- Select the balance of Vinay’s account with a shared lock

SELECT balance FROM bank\_accounts WHERE account\_id = 1 FOR UPDATE;

-- Update Vinay’s balance

UPDATE bank\_accounts SET balance = balance - 100.00 WHERE account\_id = 1;

-- Update Nikhil balance

UPDATE bank\_accounts SET balance = balance + 100.00 WHERE account\_id = 2;

COMMIT;

**Note:** If an error occurs, we would use ROLLBACK instead of COMMIT.

**ISOLATION LEVELS**

Databases support different isolation levels to control the visibility of changes made by concurrent transactions. Isolation levels control the visibility of changes made by one transaction to other concurrent transactions.

The four standard isolation levels are:

1. **Read Uncommitted**: Allows dirty reads.
2. **Read Committed**: Prevents dirty reads.
3. **Repeatable Read**: Prevents dirty reads and non-repeatable reads.
4. **Serializable**: Prevents dirty reads, non-repeatable reads, and phantom reads.
5. **Read Uncommitted**

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

-- Start a transaction

BEGIN TRANSACTION;

-- Read balance from Vinay’s account (possible dirty read)

SELECT balance FROM bank\_accounts WHERE account\_id = 1;

-- Commit the transaction

COMMIT;

1. **Read Committed :**

-- Set isolation level to Read Committed

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Start a transaction

BEGIN TRANSACTION;

-- Read balance from Vinay's account (no dirty reads)

SELECT balance FROM bank\_accounts WHERE account\_id = 1;

-- Commit the transaction

COMMIT;

1. **Repeatable Read :**

-- Set isolation level to Repeatable Read

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- Start a transaction

BEGIN TRANSACTION;

-- Read balance from Vinay's account (no dirty reads, no non-repeatable reads)

SELECT balance FROM bank\_accounts WHERE account\_id = 1;

-- Try to read balance again

SELECT balance FROM bank\_accounts WHERE account\_id = 1;

-- Commit the transaction

COMMIT;

1. **Serializable :**

-- Set isolation level to Serializable

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- Start a transaction

BEGIN TRANSACTION;

-- Read balance from Alice's account (no dirty reads, no non-repeatable reads, no phantom reads)

SELECT balance FROM bank\_accounts WHERE account\_id = 1;

-- Try to read balance again

SELECT balance FROM bank\_accounts WHERE account\_id = 1;

-- Commit the transaction

COMMIT;

### Demonstrating Concurrency Control

Consider two concurrent transactions.

-- Transaction 1: Transfer from Vinay to Nikhil

BEGIN TRANSACTION;

-- Read current balances and lock accounts

SELECT balance FROM bank\_accounts WHERE account\_id = 1 FOR UPDATE;

SELECT balance FROM bank\_accounts WHERE account\_id = 2 FOR UPDATE;

-- Decrease Vinay's balance

UPDATE bank\_accounts SET balance = balance - 200.00 WHERE account\_id = 1;

-- Increase Nikhil's balance

UPDATE bank\_accounts SET balance = balance + 200.00 WHERE account\_id = 2;

-- Commit transaction 1

COMMIT;

-- Transaction 2: Transfer from Nikhil to Vinay

BEGIN TRANSACTION;

-- Read current balances and lock accounts

SELECT balance FROM bank\_accounts WHERE account\_id = 2 FOR UPDATE;

SELECT balance FROM bank\_accounts WHERE account\_id = 1 FOR UPDATE;

-- Decrease Nikhil's balance

UPDATE bank\_accounts SET balance = balance - 300.00 WHERE account\_id = 2;

-- Increase Vinay's balance

UPDATE bank\_accounts SET balance = balance + 300.00 WHERE account\_id = 1;

-- Commit transaction 2

COMMIT;

**Conclusion:**

Concurrency control in bank\_accounts ensures that transactions involving financial data are processed correctly and consistently, even when multiple transactions access the same accounts concurrently.