Module 15

What to learn

Transaction
ACID Properties
Implicit and Explicit Transaction
Rollback Transaction
Isolation Level
Deadlock

Practice Exercise

Practice 1

Do the hands on the things provided in video and ppt.

Practice 2

Practice Exercise: Understanding Transactions and ACID Properties

Consider a banking system where you have two tables:

Accounts

Columns: AccountID (INT, PK), Balance (DECIMAL, NOT NULL).

Transactions

Columns: TransactionID (INT, PK), AccountID (FK), Amount (DECIMAL), TransactionDate (DATETIME).

Write an SQL query to perform both a credit and debit transaction on the *Accounts* table. Implement a solution where you ensure the ACID properties are satisfied when moving money from one account to another using a transaction.

Hint: Use COMMIT and ROLLBACK to manage the atomicity of your operations.

Practice 3

Practice Exercise: Implicit and Explicit Transactions

Using the *Accounts* table from the previous question, demonstrate the difference between implicit and explicit transactions in SQL. Write:

An update statement that works under an implicit transaction.
A set of update and insert operations wrapped into an explicit transaction using BEGIN TRANSACTION, COMMIT, and ROLLBACK.

Practice 4

Practice Exercise: Rollback on Transaction Failure

Using the *Accounts* table, simulate a transaction where funds are deducted from an account but fail when there are insufficient funds. Showcase how to rollback the transaction upon failure.

Challenge: Write a query where you check the balance before deduction and rollback the transaction if the available balance is less than the amount being debited.

Practice 5

Practice Exercise: Setting and Demonstrating Isolation Levels

Consider three tables simulating an *e-commerce* application:

Tables:

Orders (OrderID, ProductID, OrderDate, Quantity).
Products (ProductID, ProductName, StockCount).
Customers (CustomerID, CustomerName, ContactInfo).

Write SQL queries to simulate the following isolation levels:

READ UNCOMMITTED: Attempt to read data being modified by another transaction.
READ COMMITTED: Ensure that only committed data is visible in your transaction.
REPEATABLE READ: Lock the rows to prevent updates during a transaction.
SERIALIZABLE: Simulate row-level blocking to prevent any updates or inserts in a transaction scope.

Practice 6

Practice Exercise: Deadlock Resolution

Using the *Accounts* and *Transactions* tables from earlier, simulate a scenario where two transactions lead to a deadlock. Write SQL queries to:

Create the deadlock scenario by using locked rows.
Resolve the deadlock by changing the sequence of SQL statements.
Apply appropriate indexing strategies to avoid future deadlocks.

Note: Focus on logical and practical deadlock simulations.

Assignment Exercise

Assignment 1

Detroit Bank need to implement the transaction whenever the amount is transferred from one account to the another account.

Assignment 2

Assignment: Managing an Online Payment Portal's Database

Design and implement an online payment portal's database management system in SQL.

Functional Flow:

The portal manages transactions for customers using different payment methods. Use the following tables:

Tables:

Users (UserID, Username, Email, ContactNumber).
Transactions (TransactionID, UserID (FK), Amount, TransactionDate, PaymentMethod [Cash, Credit, Debit, Online]).
PaymentFailures (FailureID, TransactionID (FK), FailureReason).

Requirements:

Create the tables as per the schema with appropriate constraints.
Insert a minimum of 10 records into each table to represent a real-world scenario.
Simulate a money transfer from one user to another using explicit transactions, ensuring ACID properties.
Create a mechanism using SQL queries to log payment failure reasons into the *PaymentFailures* table, ensuring the transaction rollbacks are accurate upon failure.
Implement isolation levels for the *Transactions* table operations addressing concurrency issues such as dirty reads and phantom reads.
Simulate and resolve possible deadlocks when two users attempt concurrent transactions involving the same UserID.

Business Logic:

1. Only allow transactions between users if the source user has sufficient balance.
2. Automatically log transactions into a history table after successful completion.
3. Transactions paying more than $1000 should mandatorily require a valid online mode and log user details into a priority list for manual confirmation.
4. Re-attempt failed *Online* payments at least twice before marking them as failed and logging them into the *PaymentFailures* table.

Expected Output:

Functional integration of all SQL concepts learned: transactions, ACID properties, rollbacks, isolation levels, and deadlock prevention.

Reports listing users with multiple failed transactions and the top 5 users by transaction amount.
A dashboard view summarizing total transactions, successful payments, and failed payments grouped by payment method.

Online Reference

No online Reference

Introduction to Relational Databases

Introduction to Select Statement

Filtering Results with WHERE Statements

Utilizing Joins

Executing Sub queries and Unions

Aggregating Data

Advanced Data Aggregations

Built in Functions

Query Optimization

Modifying Data

Advanced Data Modification

Stored Procedure

Transaction

Error handling

Designing Tables

triggers