CHAROTAR UNIVERSITY OF SCIENCE & TECHNOLOGY
Faculty of Technology and Engineering
Devang Patel Institute of Advance Technology and Research (DEPSTAR)
Fundamentals of Database Management Systems (ITUC202)
Practical List – EVEN 2025-26
Index

Field	Content (For Students)
Practical Number	Practical – 1 (Open-Ended Foundation Practical)
CO/PO	CO1, CO2 • PO1, PO2, PO4
Problem Definition (Open-Ended with Detailed Process Guidance)	In this practical, you are provided with a **real-world banking system description**, and you must **analyze the problem, identify entities, design the schema, and implement the database in SQL**. ◆ **Business Scenario Given to Students:** A bank operates through multiple branches located in different cities. Each branch employs staff members working in different job roles. The bank serves thousands of customers. A customer can open one or more bank accounts in any branch. Accounts can be of multiple types such as Savings, Current, or Salary. Customers perform transactions on their accounts such as Credit and Debit. The bank also provides loans such as Home Loan, Car Loan, and Personal Loan to customers. Every loan has a specific interest rate and issue date. Transactions are processed by employees. **You must perform the following tasks step-by-step:** **Step 1 – Requirement Analysis:** Identify all main **entities**, **attributes**, and **relationships** from the given scenario. **Step 2 – Entity Identification:** List all tables required (minimum 6–8 tables expected). **Step 3 – Attribute Design:** Decide suitable columns and **appropriate data types** for each table. **Step 4 – Key Design:** Identify **Primary Keys and Foreign Keys** for all tables. **Step 5 – Relationship Mapping:** Define **one-to-many and many-to-one** relationships correctly. **Step 6 – Schema Creation:** Write full `CREATE TABLE` statements with constraints. **Step 7 – Data Population:** Insert **minimum 5 realistic records in each table**. **Step 8 – Validation:** Run `DESCRIBE`, `SELECT`, and constraint-violation tests to validate your schema. **Important:** There is **no single correct design**. Marks will be given for **logical correctness, completeness, and real-world validity** of your design.
Key Questions / Analysis / Interpretation to be Evaluated During/After Implementation	• Which entities did you identify from the problem and why? • How did you decide the Primary Key for each table? • Why did you choose specific data types for Balance, Loan Amount, and Interest Rate? • How did you map Employee–Transaction and Customer–Account relationships? • What problems appeared when you inserted wrong or inconsistent data? • How does your design prevent duplicate and invalid records?

Supplementary Problems (Advanced – Design Exploration)	1) Extend your design to support **ATM transactions** with machine location tracking. 2) Modify your design to support **joint accounts** (multiple customers per account). 3) Add support for **loan repayment tracking** with installment numbers and payment dates. 4) Introduce a **Customer Status system** (Active, Dormant, Blacklisted).
Key Skills to be addressed	Problem analysis, Entity identification, Attribute selection, Key design, Normalization thinking, Schema creation
Applications	Core Banking System Design, Financial Software Engineering, ERP Banking Modules
Learning Outcome	After completing this practical, you will be able to **independently analyze a real-world business problem, design a complete relational database schema from scratch, apply keys and constraints, and implement a working SQL database system without being given a predefined design**.
Dataset/Test Data (Source and Description If Applicable)	Student-generated realistic banking data (customers, employees, branches, accounts, transactions, loans)
Tools/Technology To Be Used	SQL Server / MySQL / PostgreSQL
Total Hours of Problem Definition Implementation	**2 Hours**
Total Hours of Engagement	**2 Hours**
Post Laboratory Work Description (Design Justification & Reflection)	1) Draw a **proper ER Diagram** for your final implemented design. 2) Write a **1–2 page design justification report** explaining why you chose each table and key. 3) Identify at least **3 design mistakes** you initially made and how you corrected them. 4) Compare your design with one of your classmates and note **2 structural differences**.
Rubrics (Practical Evaluation/Viva)	Requirement Analysis – **20%** Correctness of Entity & Relationship Design – **25%** Quality of SQL Schema – **20%** Data Validity & Testing – **15%** Viva & Design Explanation – **20%**

Schema: Global Trust Bank – Enterprise Database System

Global Trust Bank is a mid-sized financial institution expanding its digital operations. The bank requires a comprehensive relational database to manage:

- Branch information
- Employees & job profiles
- Customers
- Accounts & transactions
- Loans & loan repayments

This schema will be used across **all 8 practicals**, each focusing on different DBMS concepts.

SCHEMA DESIGN

Below is the permanent schema that will be used for all practicals:

1. Branch

Column	Type	Description

BranchID	INT (PK)	Unique Branch Code
BranchName	VARCHAR(50)	Name of Branch
City	VARCHAR(30)	City of branch
IFSC	VARCHAR(15) UNIQUE	Branch IFSC code

2. Job

Column	Type	Description
JobID	VARCHAR(15) (PK)	Job Code
JobTitle	VARCHAR(40)	Position Name
MinSalary	DECIMAL(10,2)	Minimum salary
MaxSalary	DECIMAL(10,2)	Maximum salary

3. Employee

Column	Type	Description
EmpID	INT (PK)	Employee ID
EmpName	VARCHAR(40)	Employee Name
Email	VARCHAR(50) UNIQUE	Email ID
Salary	DECIMAL(10,2)	Base Salary
Commission	DECIMAL(10,2) NULL	Commission
BranchID	INT (FK)	Works At Which Branch
JobID	VARCHAR(15) (FK)	Job Role

4. Customer

Column	Type	Description
CustomerID	INT (PK)	Customer ID
CustName	VARCHAR(50)	Customer Full Name
City	VARCHAR(30)	Customer City
Phone	VARCHAR(15)	Contact Number
Email	VARCHAR(50) UNIQUE	Communication Email

5. Account

Column	Type	Description
AccountNo	BIGINT (PK)	Unique Account Number
CustomerID	INT (FK)	Owner of Account
BranchID	INT (FK)	Branch where account exists
AccountType	VARCHAR(20)	Savings, Current, Salary
Balance	DECIMAL(12,2)	Current Account Balance
OpeningDate	DATE	Date of opening

6. Transaction

Column	Type	Description
TransID	BIGINT (PK)	Transaction ID
AccountNo	BIGINT (FK)	Linked Account
TransType	VARCHAR(10)	Credit / Debit
Amount	DECIMAL(10,2)	Amount
TransDate	DATETIME	Timestamp
PerformedByEmpID	INT (FK)	Employee who processed transaction

7. Loan

Column	Type	Description
LoanID	INT (PK)	Loan ID
CustomerID	INT (FK)	Loan Owner
BranchID	INT (FK)	Branch that issued loan
LoanType	VARCHAR(20)	Home, Car, Personal
Amount	DECIMAL(12,2)	Loan Amount
InterestRate	DECIMAL(5,2)	Annual Interest Rate
IssueDate	DATE	Date of Issue

Field	Content (For Students)
Practical Number	Practical – 2
CO/PO	CO1, CO2 • PO1, PO2, PO3
Problem Definition (Advance Level with Compulsory Queries)	In this practical, you will **design, implement, and validate a complete enterprise-level relational database schema** for the **Global Trust Bank System**. You must carefully apply **data types, primary keys, foreign keys, unique constraints, NOT NULL constraints, and CHECK conditions** to ensure **data integrity, consistency, and real-world correctness**. After designing the schema, You **must execute and verify the following compulsory queries**: **1)** Create the **Branch** table with proper constraints on BranchID and IFSC. **2)** Create the **Job** table with salary range validation using CHECK on MinSalary and MaxSalary. **3)** Create the **Employee** table with FOREIGN KEY references to Branch and Job, including UNIQUE email enforcement. **4)** Create **Customer** and **Account** tables with correct relationship using FOREIGN KEY on CustomerID and BranchID. **5)** Create the **Transaction** table with validation for Credit/Debit using CHECK constraint and foreign keys to Account and Employee. **6)** Create the **Loan** table with InterestRate strictly between 1% and 20% using CHECK constraint. **7)** Insert **minimum 10 realistic records** in **each table** while maintaining referential integrity. **8)** Display the **complete structure of all tables** using DESCRIBE / system views. **9)** Retrieve and verify **Primary Key–Foreign Key relationships** using system catalog queries. **10)** Intentionally attempt **constraint violation (duplicate email, invalid FK, negative balance)** and observe generated error messages.This practical must be executed as if you are creating a **real banking production database**.
Key Questions / Analysis / Interpretation	• Why is Primary Key mandatory for every table in a relational system? • What types of errors occur during Foreign Key violations? • Why UNIQUE constraint is applied on Email and IFSC fields in banking databases? • Why CHECK constraint is important for Salary, Balance, and InterestRate? • What is the difference between logical integrity and referential integrity? • How does schema design impact system performance and security?

Supplementary Problems (Advanced Level – Detailed)	1) Add a new column **AlternatePhone** in Customer table with UNIQUE constraint. 2) Add a **Status column (Active / Blocked)** in Account table using CHECK constraint. 3) Add a **CreatedAt timestamp column** in Transaction table with default current date/time. 4) Modify the **size of AccountType column** to support future banking schemes. 5) Drop and recreate the **Employee–Branch Foreign Key** and observe dependency behavior. 6) Introduce a **new test table**, populate it, TRUNCATE it, and compare with DELETE and DROP behavior.
Key Skills to be addressed	Enterprise database design, constraint engineering, schema validation, integrity enforcement, metadata analysis
Applications	Core Banking Systems, Financial Transaction Systems, Customer Relationship Management (CRM), Loan Processing Platforms
Learning Outcome	At the end of this practical, students will be able to **design a fully normalized enterprise database schema**, apply **all integrity constraints**, handle **real-world schema violations**, and validate database structure using **system metadata and catalog views**.
Dataset/Test Data (Source and Description If Applicable)	Self-generated **realistic banking dataset** including multiple branches, employees, customers, accounts, transactions, and loans
Tools/Technology To Be Used	SQL Server / MySQL / PostgreSQL
Total Hours of Problem Definition Implementation	**2 Hours**
Total Hours of Engagement	**2 Hours**
Post Laboratory Work Description (Advanced – Similar Problems)	1) Redesign the same schema for a **Microfinance Bank** with minimum 2 modifications per table. 2) Add a new table **Locker** and establish relationship with Customer and Branch. 3) Simulate **wrong schema design**, identify anomalies, and correct them. 4) Write a short **schema validation report** highlighting PK, FK, UNIQUE, and CHECK usage.
Rubrics (Practical Evaluation/Viva)	Schema correctness – 20% Data quality – 20% Constraint implementation – 20% Metadata verification – 20% Viva – 20%

Field	Content (For Students)
Practical Number	Practical – 3
CO/PO	CO2, CO5 • PO1, PO2, PO3
Problem Definition (Advanced Level with Compulsory Queries)	In this practical, you will write **advanced SELECT queries with filtering and sorting** on the **Global Trust Bank** database. The objective is to learn how to extract **precise, meaningful, and business-oriented information** using `WHERE`, `IN`, `BETWEEN`, `DISTINCT`, `TOP/LIMIT`, `ORDER BY`, logical operators, and NULL handling.

	You must **execute and submit the following compulsory queries:** **1)** Display all **customers** along with their City and Phone. **2)** List all **Savings accounts** with Balance greater than **₹25,000**. **3)** Retrieve all **employees** whose Salary is **between ₹30,000 and ₹80,000** (inclusive). **4)** Display the list of **distinct cities** where customers reside. **5)** Show the **Top 5 accounts with highest Balance** (use `TOP` / `LIMIT`). **6)** Fetch all **transactions** that occurred **between two given dates** (e.g., '2024-01-01' and '2024-03-31'). **7)** Find all **employees** who are **earning commission** (i.e., `Commission IS NOT NULL` and `Commission > 0`). **8)** Display all **loans** where `LoanType` is either **'Home'**, **'Car'**, or **'Education'** using `IN`. **9)** Fetch all **accounts** whose Balance is **between ₹1,00,000 and ₹5,00,000** and order them by Balance in **descending order**. **10)** Retrieve all **customers from a specific city** (e.g., 'Surat' or 'Ahmedabad') and sort them by name in **alphabetical order**. **11)** List all **transactions of type 'Debit'** where `Amount > ₹50,000` and order them by `TransDate` (latest first). **12)** Find all **customers who do not have an Email** (`Email IS NULL`), if any.
Key Questions / Analysis / Interpretation to be evaluated during/after Implementation	• How does applying different conditions with `WHERE`, `AND`, `OR` change the result set? • When should we use `IN` instead of multiple `OR` conditions? • Why is `BETWEEN` useful for date and amount-based queries in banking? • What is the impact of using `TOP`/`LIMIT` in dashboards and reports? • Why is `DISTINCT` important in removing duplicates in report generation? • How does appropriate ordering (`ORDER BY`) improve readability and decision-making in banking reports?
Supplementary Problems (Advanced Level – Detailed)	1) Display all **customers having accounts in any of the branches** located in cities `'Surat'`, `'Vadodara'`, `'Ahmedabad'` using `IN` and subqueries. 2) List all **loans** where `Amount` is **greater than the average loan amount** (use `BETWEEN` in combination with subquery or direct comparison). 3) Retrieve all **employees** whose salary is **not in the range ₹40,000–₹60,000** and who belong to specific job roles (e.g., `'MANAGER'`, `'CLERK'`). 4) Show **Top 10 highest debit transactions** for a particular account number using `TOP/LIMIT` with `ORDER BY Amount DESC`. 5) Display all **accounts which were opened between two specific dates**, sorted by `OpeningDate` ascending. 6) Find **customers from a given city** whose name **starts with 'A'** and order them by length of name (if supported).
Key Skills to be addressed	• Writing robust data retrieval queries using `SELECT` and `WHERE` • Designing business-oriented filters using `IN`, `BETWEEN`, logical operators • Removing duplication and improving clarity using `DISTINCT` • Implementing sorted and paginated result sets using `ORDER BY` and `TOP/LIMIT` • Understanding and handling `NULL` values in real datasets
Applications	• Generation of **daily customer and account reports** • **Transaction monitoring** and high-value transaction identification • **Regulatory and compliance reporting** (e.g., large value transactions) • Customer segmentation based on **city, balance, or loan type**

Learning Outcome	After completing this practical, you will be able to: (1) Design **precise SQL queries** to extract required information from large banking datasets, (2) Apply multiple types of conditions and filters to answer **real banking questions.** (3) Generate **sorted, clean, and interpretation-ready result sets** for use in reports and dashboards.
Dataset/Test Data (Source and Description If Applicable)	Use the **Global Trust Bank** schema created in Practical 1 with **at least 10 records in each of the tables**: `Branch`, `Job`, `Employee`, `Customer`, `Account`, `Transaction`, `Loan`. Ensure that data covers multiple cities, different loan types, varying balances, and different salary ranges so filters can be meaningfully tested.
Tools/Technology To Be Used	SQL Server / MySQL / PostgreSQL (any one as per lab setup)
Total Hours of Problem Definition Implementation	**2 Hours**
Total Hours of Engagement	**2 Hours** (including implementation, verification, and doubt resolution)
Post Laboratory Work Description (Advanced – Similar Problems)	After the lab session, students must complete and submit the following additional tasks: 1) Design **5 new SELECT queries** that a **Bank Manager** might need for decision-making (e.g., "Top N customers based on balance", "List of customers with both savings and loan accounts in a specific branch"). 2) Write **3 complex filter queries** combining `IN`, `BETWEEN`, `AND`, `OR`, and `ORDER BY` to generate **targeted marketing lists**, such as customers eligible for premium credit cards. 3) Prepare a **short report file** (Word/PDF) containing: (a) Query, (b) Screenshot of execution, (c) One-line interpretation of the result in **business language** (e.g., "This query helps RM identify high-value customers in Surat").
Rubrics (Practical Evaluation/Viva)	Query correctness & execution – **40%** Use of appropriate filters (`IN`, `BETWEEN`, `DISTINCT`, `ORDER BY`, `TOP`) – **25%** Relevance of query to business context – **15%** Post-lab additional queries & explanations – **10%** Viva / Conceptual understanding – **10%**

Field	Content (For Students)
Practical Number	Practical – 4
CO/PO	CO2, CO5 • PO1, PO2, PO3
Problem Definition (Advanced Level with Compulsory Queries)	In this practical, you will perform **two major categories of operations** on the **Global Trust Bank** database: 1)**String Functions, Pattern Matching & Data Cleaning** – to clean, standardize, and validate customer, employee, and branch textual data. 2)**Aggregates, Grouping, HAVING & Set Operations** – to perform analytical reporting on accounts, loans, and transactions. You must **implement and execute at least the following queries:** **Part A – String Functions & Data Cleaning** **1)** Display all **Customer Names** in `UPPER()` and `LOWER()` format along with original names.

	2) Use `TRIM()` / `LTRIM()` / `RTRIM()` to remove extra spaces from `CustName` and show before–after effect (use alias columns like `OriginalName`, `CleanedName`). **3)** Extract the **username** from `Email` of each customer (portion before '@') using `SUBSTRING` and `CHARINDEX` (or equivalent). **4)** Display all customers whose email domain is **'gmail.com'**, **'yahoo.com'**, or **'outlook.com'** using `LIKE`/pattern matching. **5)** Replace all occurrences of city name `'Amdavad'` or `'Barodra'` with standardized forms `'Ahmedabad'` and `'Vadodara'` using `REPLACE`. **6)** Mask the **last 4 digits** of customer `Phone` number for privacy (e.g., `98XXXXXX45`) using string functions. **7)** Find and list all **employees whose name starts with 'A' and ends with 'l'** using `LIKE` pattern (`A%l`). **Part B – Aggregates, Grouping, HAVING & Set Operations** **8)** Display **total number of customers in each City** using `COUNT(*)` and `GROUP BY City`. **9)** Show **total and average Balance per Branch** using `SUM(Balance)` and `AVG(Balance)` grouped by `BranchID` (or BranchName through a join). **10)** For each **LoanType**, display **total loan amount**, **minimum loan**, and **maximum loan** using `SUM`, `MIN`, and `MAX`. **11)** List only those **branches** where **total loan amount exceeds a given threshold** (e.g., ₹50,00,000) using `HAVING`. **12)** Generate a report of **number of accounts per AccountType** (Savings, Current, Salary) and show only those account types having **more than 5 accounts** using `HAVING COUNT(*) > 5`. **13)** Using **set operations**: • Find customers who **have both an Account and a Loan** (use `INTERSECT` or equivalent logic). • Find customers who **have an Account but no Loan** (`EXCEPT` / `MINUS` or equivalent). **14)** Generate a summary of **total debit and total credit transaction amount** per Account using `GROUP BY AccountNo` and a conditional aggregation (if supported) or separate queries.
Key Questions / Analysis / Interpretation to be evaluated during/after Implementation	• Why is **data cleaning** (removing spaces, standardizing text) critical in banking systems? • How does **pattern matching** help in **email and phone validation** and in identifying invalid or suspicious data? • What is the difference between `COUNT(*)` and `COUNT(column)`? When can they differ in result? • Why do we need `GROUP BY` when using aggregate functions like `SUM`, `AVG`, `MAX`? • How does `HAVING` differ from `WHERE` in analytical queries? • How do **set operations** (`UNION`, `INTERSECT`, `EXCEPT`) help in comparing groups of customers such as "with loan" vs "without loan"? • How can these queries be used to support **business decisions**, such as identifying high-performing branches or risk-heavy loan types?
Supplementary Problems (Advanced Level – Detailed)	1) Identify all **potentially invalid email IDs** (e.g., those not containing `'@'` or ending with unsupported domains). 2) Generate a list of **top 3 cities** with the **highest number of customers**, and for each such city, show the **total account balance** as well. 3) Find **branches** where the **average account balance** is **less than the overall average account balance** of the bank.

	4) Generate a **Loan Risk View**: for each `LoanType`, show **avg loan amount**, **max loan amount**, and only display `LoanType` where avg amount is **above overall avg loan amount**. 5) Create a **"Premium Customer List"** of customers having **balance > ₹5,00,000** OR **loan amount > ₹10,00,000** and living in **metro cities** (you can define which cities are metro in your dataset). 6) Show **total number of transactions per day** for a selected month and identify days having **no transactions at all** (use grouping and possibly set logic).
Key Skills to be addressed	• String-based data cleansing and formatting • Pattern matching for validation and search • Design and use of aggregate functions for analytics • Group-based reporting using `GROUP BY` and `HAVING` • Comparative analytics using set operations (`UNION`, `INTERSECT`, `EXCEPT`/`MINUS`)
Applications	• KYC data standardization and cleansing • Email/phone validation for communication systems • Branch, product and city-wise performance reporting • Identification of high-value or high-risk customers • MIS & BI reporting for management
Learning Outcome	After completing this practical, you will be able to: (1) **Clean and standardize text data** using string functions, (2) Use **pattern matching** to validate and search customer/employee data, (3) Build **analytical SQL queries** using aggregates, grouping, and HAVING (4) Apply **set-based thinking** to compare different categories of customers and accounts for business insights.
Dataset/Test Data (Source and Description If Applicable)	Continue using the **Global Trust Bank** schema from previous practicals with **rich, varied data**: multiple branches, different loan types, diverse cities, realistic email/phone variations, and a mix of small and large balances and loan amounts. Ensure presence of some **messy/duplicate/inconsistent entries** (e.g., inconsistent city spellings) to effectively test cleaning operations.
Tools/Technology To Be Used	SQL Server / MySQL / PostgreSQL (as available in the lab)
Total Hours of Problem Definition Implementation	**2 Hours**
Total Hours of Engagement	**2 Hours**
Post Laboratory Work Description (Advanced – Similar Problems)	After completing the in-lab work, students must solve and document the following additional tasks: 1) Design a **"City Quality Report"** showing, for each city, (a) number of customers, (b) total balance, and (c) count of customers with **missing or invalid emails**. Use string and aggregate functions together. 2) Create a **"Branch Health Summary"** that lists each branch with: total number of accounts, total balance, number of loans, and show only those branches where **total balance < total loan amount** (possible risk branches). 3) Prepare a **"Customer Contact Cleansing Report"** that lists all customers with malformed phone numbers (wrong length or invalid pattern) and suggest corrections (you may assume correction rules).

	4) Submit a **short reflection note** (one page) explaining: (a) two examples where data cleaning changed the result of reports, (b) one example where GROUP BY + HAVING directly supports a managerial decision (e.g., closing/merging a weak branch).
Rubrics (Practical Evaluation/Viva)	String & cleaning logic – **25%** Aggregate & grouping correctness – **25%** Use of HAVING and set operations – **20%** Business relevance & interpretation of results – **15%** Viva / Conceptual clarity – **15%**
Field	**Content (For Students)**
Practical Number	Practical – 5
CO/PO	CO3, CO5 • PO1, PO2, PO3
Problem Definition (Advanced Level with Compulsory Queries)	In this practical, you will perform **multi-table relational data extraction** using **INNER JOIN, LEFT JOIN, RIGHT JOIN, SELF JOIN**, and advanced filtering using **EXISTS and NOT EXISTS** on the **Global Trust Bank** database. The objective is to understand how data from multiple related entities such as **Customer, Account, Branch, Employee, Transaction, Job, and Loan** can be combined to generate **real-world business intelligence reports**. You must **execute and submit at least the following compulsory queries:** **1)** Display **Customer Name, Account Number, Account Type, and Branch Name** using JOIN between Customer, Account, and Branch tables. **2)** Display **Employee Name, Job Title, Salary, and Branch Name** using JOIN between Employee, Job, and Branch. **3)** List all **customers who have taken at least one loan** using INNER JOIN between Customer and Loan tables. **4)** Find all **customers who do NOT have any loan** using `NOT EXISTS`. **5)** Display all **branches that have NO employees** using LEFT JOIN with NULL filtering or `NOT EXISTS`. **6)** Display all **employees who have never processed any transaction** using `NOT EXISTS` with Transaction table. **7)** Find all **customers who have more than one account** using JOIN and GROUP BY/HAVING logic. **8)** Display **high-value customers (Balance > ₹5,00,000)** along with their Branch Name and Account Type using JOIN and filtering. **9)** Perform a **SELF JOIN on Customer table** to find pairs of customers who belong to the **same City** (excluding self-matches). **10)** Display **Branch-wise total number of transactions processed** using JOIN between Branch, Account, and Transaction tables. **11)** Find all **loans issued by branches located in a specific city** (e.g., Surat) using multi-table JOIN. **12)** Display **employees who processed transactions for accounts belonging to another branch** (cross-branch transaction handling – advanced analysis).
Key Questions / Analysis / Interpretation to be evaluated during/after Implementation	• What is the difference between **INNER JOIN, LEFT JOIN, and RIGHT JOIN** in terms of result sets? • Why is **SELF JOIN** required when a table has logical relationships within itself? • When should **EXISTS / NOT EXISTS** be preferred over `IN` or JOIN?

	• How does incorrect join condition lead to a **Cartesian Product**, and why is it dangerous in enterprise databases? • Why are JOIN-based queries central to **core banking reports**? • How do JOIN operations affect query performance as data volume increases?
Supplementary Problems (Advanced Level – Detailed)	1) Identify **customers whose total transaction amount exceeds ₹10,00,000** and display their branch information using JOIN + GROUP BY. 2) Detect **dormant accounts** (accounts with no transactions in the last 12 months) using LEFT JOIN and date filtering. 3) Find **employees who have processed transactions for more than 50 distinct accounts** (workload analysis). 4) Generate a **Customer–Loan–Branch 360-degree report** showing Customer Name, Loan Type, Loan Amount, Branch City. 5) Identify **branches having both high deposits and high loans**, based on account balance and loan totals using multiple JOINs and HAVING. 6) Find **customers who have a loan in one branch but an account in another branch** (cross-branch dependency analysis).
Key Skills to be addressed	• Multi-table relational query design • Use of JOINs for enterprise data extraction • Anti-join logic using `NOT EXISTS` • SELF JOIN for intra-entity relationship analysis • Performance-aware SQL design
Applications	• Customer 360° profiling in banks • Cross-branch transaction monitoring • Employee workload and performance analysis • Fraud detection and compliance reporting • Loan and deposit dependency reporting
Learning Outcome	After completing this practical, you will be able to: (1) Combine multiple related tables using different types of JOINs, (2) Apply **EXISTS and NOT EXISTS** for advanced conditional filtering, (3) Perform **cross-entity business analytics** (4) Design **real-world banking reports** that rely on relational integrity and multi-table dependencies.
Dataset/Test Data (Source and Description If Applicable)	Use the **Global Trust Bank** schema with **sufficient data volume**: multiple branches, customers with multiple accounts, varied transaction history, employees across different jobs, and loans across different branches to ensure meaningful JOIN outcomes.
Tools/Technology To Be Used	SQL Server / MySQL / PostgreSQL
Total Hours of Problem Definition Implementation	**2 Hours**
Total Hours of Engagement	**2 Hours**
Post Laboratory Work Description (Advanced – Similar Problems)	1) Design a **"Complete Customer Profile Report"** showing Customer → Account → Transaction → Loan → Branch details using at least **4 JOINs**. 2) Identify **top 5 employees based on total transaction amount processed**. 3) Generate a **Branch Dependency Matrix** showing relationships between branches through shared customers and employees. 4) Prepare a **short analytical note** (1–2 pages) explaining how JOIN-based reports help banks in **risk management and resource allocation**.
Rubrics (Practical Evaluation/Viva)	Join correctness & logic – **30%** Use of EXISTS/NOT EXISTS & anti-join logic – **20%**

	Business relevance of results – **20%** Post-lab analytical work – **10%** Viva / Conceptual clarity – **20%**

Field	Content (For Students)
Practical Number	Practical – 6
CO/PO	CO4, CO5 • PO1, PO2, PO3
Problem Definition (Advanced Level with Compulsory Queries)	In this practical, you will implement **single-row subqueries, multi-row subqueries, correlated subqueries, and Common Table Expressions (CTEs)** on the **Global Trust Bank database**. The objective is to learn how **nested logic and temporary result sets** help in solving **complex real-world banking analytics problems**. You must **execute and submit the following compulsory queries:** **1)** Find the **customer(s) having the highest account balance** using a subquery. **2)** Display all **employees whose salary is greater than the average salary** of all employees using a subquery. **3)** Find all **accounts whose balance is greater than the average balance of their respective branch** using a **correlated subquery**. **4)** Display **customers who live in cities where at least one loan has been issued** using a subquery with `IN`. **5)** Find all **loans whose amount is greater than the average loan amount of the same loan type** using a **correlated subquery**. **6)** Display the **branch(es) that have issued the maximum number of loans** using nested subqueries. **7)** Using a **CTE**, display **branch-wise total account balance** in a structured and readable form. **8)** Using a **CTE**, generate a report of **employee count per branch along with total and average salary**. **9)** Using a **CTE**, identify **high-risk customers** whose total loan amount is **greater than their total account balance**. **10)** Compare the output of a **subquery-based solution vs CTE-based solution** for any one of the above problems and explain the difference in readability and maintenance.
Key Questions / Analysis / Interpretation to be evaluated during/after Implementation	• What is the difference between a **simple subquery and a correlated subquery**? • Why do **correlated subqueries execute row-by-row**, and what is their performance impact? • When should a **CTE be preferred over nested subqueries**? • How does a **derived result set** simplify complex business logic? • Why are subqueries heavily used in **risk assessment and compliance reporting**? • Can every subquery be rewritten as a JOIN? If yes, why do we still use subqueries?
Supplementary Problems (Advanced Level – Detailed)	1) Identify **customers who hold more than one type of account** using subqueries.

	2) Find **employees who earn more than the highest-paid employee of another branch** using nested subqueries. 3) Using a correlated subquery, find **branches where the total loan amount is greater than the total deposit (account balance)**. 4) Using a CTE, generate a **ranked list of branches based on total loan amount**. 5) Identify **top 3 customers per branch based on account balance** using CTE + ranking logic (if supported). 6) Create a **"Customer Financial Strength Index"** using CTE that combines total balance and total loan into a single computed score.
Key Skills to be addressed	• Nested query writing and logical decomposition • Use of correlated subqueries for row-level comparison • Structuring complex queries using CTEs • Analytical thinking for financial risk and performance evaluation
Applications	• Credit risk analysis • Branch and customer performance benchmarking • Salary and payroll analytics • Loan vs deposit dependency studies • Advanced MIS and compliance reporting
Learning Outcome	After completing this practical, you will be able to: (1) Solve **complex multi-layered SQL problems** using subqueries and correlated subqueries. (2) Design **readable and maintainable analytical queries using CTEs.** (3) Apply these techniques for **advanced banking analytics such as risk detection, branch ranking, and customer profiling**.
Dataset/Test Data (Source and Description If Applicable)	Use the **Global Trust Bank** schema with **rich interlinked data**: multiple loans per customer, multiple employees per branch, wide variation in balances, salaries, and loan amounts to ensure meaningful correlated results.
Tools/Technology To Be Used	SQL Server / MySQL / PostgreSQL
Total Hours of Problem Definition Implementation	**2 Hours**
Total Hours of Engagement	**2 Hours**
Post Laboratory Work Description (Advanced – Similar Problems)	1) Using only **subqueries (no JOINs)**, recreate any **two reports** that were earlier solved using JOINs. 2) Using **CTEs**, design a **Monthly Branch Performance Summary** that shows, for each branch: total deposits, total loans, number of transactions, and number of customers. 3) Using **correlated subqueries**, find employees whose salary is **greater than the average salary of their own job role**. 4) Prepare a **comparison note (1–2 pages)** explaining: subquery vs join vs CTE in terms of readability, performance (theoretical), and maintenance.
Rubrics (Practical Evaluation/Viva)	Subquery correctness – **30%** Correlated subquery logic – **20%** CTE structure & readability – **20%** Business interpretation & documentation – **10%** Viva / Conceptual clarity – **20%**

Field	Content (For Students)
Practical Number	Practical – 7
CO/PO	CO4, CO5 • PO1, PO2, PO3

Problem Definition (Advanced Level with Compulsory Queries)	In this practical, you will use **Common Table Expressions (CTEs), especially Recursive CTEs**, to generate **sequences, schedules, and derived analytical views** on the **Global Trust Bank** database. The focus is on solving problems that require **repeated / hierarchical / step-wise computation**, such as EMI schedules, monthly summaries, and cumulative calculations. You must **execute and submit at least the following queries:** **1)** Using a **recursive CTE**, generate a simple **number series from 1 to 12** (representing 12 EMI months). **2)** Using a **recursive CTE**, generate all **months between two given dates** (e.g., from '2024-01-01' to '2024-12-31'), with one row per month. **3)** For a given **LoanID**, generate an **EMI schedule** for 12 months using a recursive CTE: each row should show `MonthNo`, `OutstandingPrincipal`, and `ApproxEMI` (you may assume a simple flat calculation). **4)** Using a recursive CTE, generate a **daily date series for one selected month** and LEFT JOIN it with the **Transaction** table to show **days with zero transactions** for that month. **5)** Using a CTE (non-recursive), compute **branch-wise monthly total transaction amount** (Credit + Debit) and then filter to show only those (Branch, Month) pairs where **total amount exceeds a threshold** (e.g., ₹10,00,000). **6)** Using a recursive CTE, calculate a **running total of transactions (cumulative balance)** for a particular AccountNo ordered by TransDate (if your DB supports window functions, also compare with a window function solution). **7)** Assume you want to model a **simple hierarchy of Job Roles** (e.g., 'Clerk' reports to 'Manager', 'Manager' reports to 'Regional Manager'). Create a small **JobHierarchy** temp table and use a recursive CTE to display the **job hierarchy levels** from top to bottom. **8)** Use a CTE to find **top 3 branches** by **total loan amount**, and then on that result, apply another CTE (or nested CTE) to compute the **average loan per customer** for only those top branches. **9)** Using a recursive CTE, generate **a ladder of interest accumulation**: starting from initial Amount (e.g., 1,00,000) and applying a simple yearly interest rate (e.g., 8%) for 5 years, showing year-wise projected amount (Year1…Year5). **10)** Take any one complex report you built earlier using subqueries and rewrite it using one or more CTEs to improve **readability and modularity**.
Key Questions / Analysis / Interpretation to be evaluated during/after Implementation	• What is the difference between a **normal (non-recursive) CTE** and a **recursive CTE**? • How does a recursive CTE work internally (anchor part + recursive part + termination)? • In what kind of business problems are **recursive CTEs** especially useful? • How do CTEs improve the **readability and maintainability** of large SQL queries compared to deeply nested subqueries? • What are potential **performance risks** of recursive CTEs, and why is it important to have a proper **termination condition**? • How can recursive CTEs be used to **generate sequences and schedules** even when no such data exists physically in the tables?
Supplementary Problems (Advanced Level – Detailed)	1) Using a recursive CTE, generate an **installment schedule for 24 months** for all loans of type `'Home'` where each row indicates `LoanID`, `MonthNo`,

	`ApproxMonthlyAmount`, and `CumulativePaid` (assume simple interest or a flat EMI formula). 2) Create a **recursive calendar** for one full year and join it with both **Account Opening Dates** and **Loan Issue Dates** to generate a **Bank Activity Density View** (how many accounts/loans opened per day or per month). 3) Model a **simple multi-level branch structure** (e.g., Region → City → Branch) using a temporary hierarchy table and use a recursive CTE to print the full **path of each branch** (Region/City/BranchName). 4) Combine a recursive CTE with aggregation to compute **"Customer Lifetime Value approximation"**: starting from current balance and adding projected growth over several periods (hypothetical, but good for thinking). 5) Use a recursive CTE to generate **all possible installment numbers for selected high-value loans** and then join with actual **Transaction** data to detect **missing EMIs** (months where expected EMI did not happen).
Key Skills to be addressed	• Designing and using **CTEs and Recursive CTEs** • Generating **sequences, schedules, and hierarchies** using SQL only • Structuring complex analytical queries in a **modular, readable** way • Applying recursion to **financial and temporal data** (EMIs, months, dates, cumulative values)
Applications	• EMI and repayment schedule generation • Interest projection and financial forecasting • Hierarchical organization / branch / job modeling • Gap analysis (e.g., missing transactions or EMIs) • Time-based analytics without needing a separate calendar table
Learning Outcome	After completing this practical, you will be able to: (1) Implement **recursive CTEs** to generate sequences and hierarchies, (2) Use CTEs to organize complex logic into **clear, layered queries.** (3) Apply these techniques to solve **realistic banking problems** such as EMI schedules, transaction timelines, branch hierarchies, and financial projections.
Dataset/Test Data (Source and Description If Applicable)	Continue using the **Global Trust Bank** schema with previously inserted data. For hierarchy-related tasks, you may create **small helper tables** such as `JobHierarchy` or `BranchHierarchy`. For EMI/schedule tasks, ensure loans exist with **different LoanTypes, Amounts, and InterestRate values**.
Tools/Technology To Be Used	SQL Server / PostgreSQL (preferred for recursive CTEs) / MySQL 8+ (if CTE support is available)
Total Hours of Problem Definition Implementation	**2 Hours**
Total Hours of Engagement	**2 Hours**
Post Laboratory Work Description (Advanced – Similar Problems)	1) Design a **report using recursive CTE** that, for a particular year, lists each month along with: total transactions, total new accounts, and total issued loans (you may approximate or simulate if needed). 2) Prepare a **"What-if EMI Projection"** query that, given a LoanID and different interest rate scenarios (e.g., 7%, 8%, 9%), generates 3 parallel EMI schedules using CTEs (you may use separate runs or param simulation). 3) Write a **short technical note** comparing: (a) Recursive CTE-based running total, (b) Window-function-based running total (if supported), including pros and cons of each in terms of clarity and potential performance.

	4) Submit all queries and outputs in a document/PDF with proper headings and **2–3 business interpretations** explaining where such recursive logic would be critical in a real bank.
Rubrics (Practical Evaluation/Viva)	Correct CTE & recursive structure – **30%** Complexity and relevance of chosen problems – **20%** Clarity and modularity of SQL code – **20%** Post-lab analytical work & documentation – **10%** Viva / Understanding of recursion & use-cases – **20%**

Field	Content (For Students)
Practical Number	Practical – 8
CO/PO	CO5 • PO1, PO2, PO3, PO5
Problem Definition (Advanced Level with Compulsory Tasks/Queries)	In this practical, you will design and implement **database triggers** to automatically enforce **business rules** and maintain **audit trails** in the **Global Trust Bank** database. The goal is to: (1) Ensure that critical rules such as **no negative balance**, **no invalid loan updates**, and **proper logging of changes** are always followed; (2) Keep an **audit history** of important operations for **compliance and forensic analysis**. You must **implement and test at least the following triggers:** **1)** Create an **AFTER INSERT trigger on Transaction** that automatically **updates the Account balance** based on `TransType` (`Credit` increases balance, `Debit` decreases balance). **2)** Modify the trigger (or create a separate BEFORE/INSTEAD OF trigger, depending on DBMS) to **prevent Account balance from going negative**. If a debit transaction would cause negative balance, **raise an error and cancel the transaction**. **3)** Create a **Transaction_Audit** table and an **AFTER INSERT trigger on Transaction** that **inserts a log entry** into Transaction_Audit with fields like `TransID`, `AccountNo`, `TransType`, `Amount`, `TransDate`, `PerformedByEmpID`, `InsertedAt` (current timestamp). **4)** Create a **Loan_Audit** table and an **AFTER UPDATE trigger on Loan** that logs any change in `Amount` or `InterestRate` along with fields: `LoanID`, `OldAmount`, `NewAmount`, `OldInterest`, `NewInterest`, `ChangedByUser` (you may simulate), and `ChangedAt`. **5)** Test all triggers with **sample INSERT/UPDATE/DELETE statements** and observe behavior, including error messages and audit logs.
Key Questions / Analysis / Interpretation to be evaluated during/after Implementation	• What is the difference between **BEFORE, AFTER, and INSTEAD OF** triggers (as supported by your DBMS)? • How do triggers work with **multi-row INSERT/UPDATE/DELETE** operations (i.e., when more than one row is affected)? • How do `INSERTED` and `DELETED` pseudo-tables (or equivalents) help in reading old and new values inside triggers? • Why are triggers important in **banking systems** for enforcing rules and maintaining audit trails? • What are the **risks** of overusing triggers (e.g., hidden logic, performance overhead, debugging difficulty)? • Why is it important that triggers are **idempotent** and handle unexpected data gracefully?

Supplementary Problems (Advanced Level – Detailed)	1) Design a trigger that **prevents very large debit transactions** (e.g., `Amount > 2,00,000`) unless a special flag or approval code is present (you may simulate using an extra column in Transaction or a configuration table). 2) Create a trigger that **logs all failed attempts** to perform a transaction which violates the negative balance rule into a table **FailedTransaction_Log** with reason and timestamp. 3) Implement a trigger that prevents **deletion of a Branch** if there are any active Accounts or Loans mapped to it, raising a proper error message to the user. 4) Develop a trigger that automatically **increases InterestRate slightly** (e.g., +0.5%) for Loans marked as `LoanType = 'Personal'` when their `IssueDate` is older than a certain threshold (you may simulate this as a business scenario). 5) Create a trigger that maintains a **Branch_Stats** table, where each new Account or Loan automatically updates `TotalAccounts` or `TotalLoans` counts for that Branch.
Key Skills to be addressed	• Designing triggers for **automatic business rule enforcement** • Implementing **audit trails** for critical banking operations • Understanding **pseudo-tables** (`INSERTED`, `DELETED` or equivalents) • Ensuring **data consistency and preventing invalid operations** without changing all application code • Thinking about **side-effects and performance** of triggers in production systems
Applications	• Enforcement of **no overdraft** and **limit checks** • **Regulatory compliance & auditing** (logging all critical updates) • Preventing **unauthorized or risky changes** in loans and accounts • HR and employee-related change tracking • Automated maintenance of **summary and statistics tables**
Learning Outcome	After completing this practical, you will be able to: (1) Design and implement **robust database triggers** that protect critical business rules, (2) Create and maintain **audit tables** for tracking historical changes. (3) Analyze the **impact and limitations of triggers** in enterprise banking systems.
Dataset/Test Data (Source and Description If Applicable)	Use the **Global Trust Bank** schema with meaningful sample data: multiple accounts with different balances, employees across branches, loans with varying amounts and interest rates, and enough transactions to test different trigger conditions (high vs low amounts, valid vs invalid values, etc.). Create additional audit tables: `Transaction_Audit`, `Loan_Audit`, `Employee_Audit`, `FailedTransaction_Log`, `Branch_Stats` as needed.
Tools/Technology To Be Used	SQL Server / PostgreSQL / MySQL (trigger syntax may vary slightly—follow the syntax of the DBMS used in your lab)
Total Hours of Problem Definition Implementation	**2 Hours**
Total Hours of Engagement	**2 Hours**
Post Laboratory Work Description (Advanced – Similar Problems)	1) Design a **Security_Escalation_Log** table and corresponding triggers to log any **update to high-risk fields** such as `InterestRate`, `Loan Amount`, or `Account Status`, including old and new values and timestamp.

	2) Prepare a **scenario document** (1–2 pages) describing at least **three real-life bank incidents** that could be prevented or easily investigated using well-designed triggers and audit tables. 3) Implement and test an **"emergency off switch"**: a configuration table (e.g., `SystemFlags`) with a flag `AllowHighValueTransactions`, and a trigger that **blocks high-value transactions if this flag is OFF**. 4) Submit screenshots or outputs showing **normal operations vs blocked operations**, along with a brief **business explanation** of what each trigger is protecting.
Rubrics (Practical Evaluation/Viva)	Correctness of trigger logic – **30%** Audit table design & correctness – **20%** Coverage of important business rules – **20%** Post-lab scenarios & documentation – **10%** Viva / Understanding of triggers & risks – **20%**

Field	Content (For Students)
Practical Number	Practical – 9
CO/PO	CO5 • PO1, PO2, PO3, PO5
Problem Definition (Advanced Level with Compulsory Tasks/Queries)	In this practical, you will design and implement **User-Defined Functions (UDFs)** to encapsulate **reusable business logic** for the **Global Trust Bank** database. You will create **scalar functions** and **table-valued functions** to compute interest, EMI, and financial summaries that can be reused in multiple queries and reports. You must **implement and test at least the following functions:** **1)** Create a **scalar function** `fn_GetAccountAgeInYears(@AccountNo)` that returns the **age of an account in years** based on `OpeningDate` and current date. Use it to display AccountNo, OpeningDate, and AccountAge for all accounts. **2)** Create a **scalar function** `fn_GetCustomerTotalBalance(@CustomerID)` that returns the **total account balance** of a given customer by summing `Balance` from the `Account` table. Use it in a query to list `CustomerID, CustName, City, TotalBalance` for all customers. **3)** Create a **scalar function** `fn_CalculateSimpleInterest(@Principal, @Rate, @Years)` and use it with the `Loan` table to display `LoanID, CustomerID, LoanType, Amount, InterestRate, IssueDate, CalculatedSimpleInterest` (assume Years = 1 or derive from some business rule). **4)** Create a **table-valued function** `fn_GetAccountsByBranch(@BranchID)` that returns a table with columns: `AccountNo, CustomerID, AccountType, Balance`. Use it to show all accounts of a given branch and then filter accounts with `Balance > 100000`. **5)** Create a **table-valued function** `fn_GetCustomerFinancialSummary(@CustomerID)` that returns a single-row table with columns like: `CustomerID, TotalBalance, TotalLoan,`

	`NetPosition = TotalBalance - TotalLoan`. Use it in a query to list all customers whose **NetPosition is negative** (more loan than deposits).
Key Questions / Analysis / Interpretation to be evaluated during/after Implementation	• What is the difference between a **scalar UDF** and a **table-valued UDF**? • How do UDFs promote **reusability and consistency** in business logic? • In what kinds of calculations (e.g., interest, risk score, EMI) are functions especially helpful? • What are possible **downsides** of overusing UDFs (e.g., performance impact)?
Supplementary Problems (Advanced Level – Fewer but Deeper)	1) Create a **scalar function** `fn_GetBranchTotalBusiness(@BranchID)` that returns: `TotalBusiness = SumOfAllAccountBalances + SumOfAllLoanAmounts` for that branch, and use it to compare branches. 2) Create a **table-valued function** `fn_GetHighValueCustomers(@MinNetPosition)` that returns customers whose `NetPosition` (Balance – Loan) is **greater than @MinNetPosition**, including `CustName` and `City` (you may reuse `fn_GetCustomerFinancialSummary` internally or re-implement logic). 3) Extend `fn_CalculateSimpleInterest` (or create a new function) to accept **months instead of years**, and discuss how that changes calculations for short-term loans.
Key Skills to be addressed	• Designing scalar and table-valued UDFs • Encapsulating formulas and business rules • Reusing function logic across multiple queries • Thinking in terms of **modular database logic** rather than repetitive code
Applications	• EMI and interest calculators • Customer and branch summary endpoints/APIs • Risk and exposure checks based on net financial position • Standard calculations used by reporting and dashboard systems
Learning Outcome	After completing this practical, you will be able to: (1) Write **scalar and table-valued functions** for banking computations (2) Integrate these functions within SELECT statements to build **rich financial reports.** (3) Appreciate how UDFs improve **reusability and maintainability** of business logic in SQL.
Dataset/Test Data (Source and Description If Applicable)	Use the **Global Trust Bank** schema with realistic data: multiple accounts per customer, multiple loans, different interest rates and amounts, multiple branches. Ensure some customers have **only deposits**, some **only loans**, and some **both**, so that `NetPosition` and `TotalBalance` results vary.
Tools/Technology To Be Used	SQL Server / PostgreSQL / MySQL (with UDF support, as per lab setup)
Total Hours of Problem Definition Implementation	**2 Hours**
Total Hours of Engagement	**2 Hours**
Post Laboratory Work Description (Advanced – Similar but Limited Set)	1) Design a **Customer Snapshot Report** using your functions that shows: `CustomerID, CustName, City, TotalBalance, TotalLoan, NetPosition` and categorize each customer as `'Safe'`, `'Neutral'`, or `'Risky'` based on NetPosition thresholds (you can do categorization in SQL or propose a new scalar function).

	2) Write a **one-page documentation** table listing each UDF you created, with columns: `FunctionName, Type (Scalar/TVF), Parameters, Return Type, Short Business Description`, and submit it along with SQL scripts.
Rubrics (Practical Evaluation/Viva)	Correct function creation & syntax – **30%** Correctness of business calculation – **25%** Effective usage in queries – **20%** Post-lab documentation – **10%** Viva / Conceptual clarity – **15%**

Field	Content (For Students)
Practical Number	Practical – 10
CO/PO	CO4, CO5 • PO1, PO2, PO3, PO5
Problem Definition (Advanced Level with Compulsory Procedures)	In this practical, you will design and implement **Stored Procedures (SPs)** to handle **core banking operations** in the **Global Trust Bank** database. The aim is to encapsulate frequently used operations such as **fund transfer, account opening, loan disbursement, and reporting** inside stored procedures, using **parameters and transactions (COMMIT/ROLLBACK)** to ensure **ACID properties**. You must **implement and test at least the following stored procedures:** **1) `sp_OpenNewAccount`** – A stored procedure to **open a new account** for an existing customer.• Input parameters: `@CustomerID`, `@BranchID`, `@AccountType`, `@OpeningBalance`.• Tasks: (a) Insert a new row into `Account` with given details and current date as `OpeningDate`, (b) Optionally insert an initial **Credit transaction** into `Transaction` table for the opening balance, using a transaction block to ensure either both succeed or both fail. **2) `sp_TransferFunds`** – A stored procedure to **transfer funds** from one account to another.• Input parameters: `@FromAccountNo`, `@ToAccountNo`, `@Amount`, `@PerformedByEmpID`.• Tasks: (a) Check if `@FromAccountNo` has **sufficient balance**, (b) If yes, begin a transaction, insert a **Debit transaction** for `FromAccountNo` and a **Credit transaction** for `ToAccountNo`, update both balances, and then `COMMIT`. (c) If balance is insufficient, **ROLLBACK** and return an error/message. **3) `sp_RecordLoanDisbursement`** – A stored procedure to **disburse a loan** to a customer.• Input parameters: `@CustomerID`, `@BranchID`, `@LoanType`, `@Amount`, `@InterestRate`.• Tasks: (a) Insert a new row in `Loan` table with `IssueDate = GETDATE()` (or equivalent), (b) Optionally credit the customer's selected account with the loan amount by inserting a transaction and updating the account balance inside a transaction block. **4) `sp_GetCustomerStatement`** – A stored procedure to **generate a mini-statement** for an account.• Input parameters: `@AccountNo`, `@FromDate`, `@ToDate`.• Tasks: (a) Return all rows from `Transaction` table for that `AccountNo` between the given dates, ordered by `TransDate`, (b) Optionally include running balance if you wish (can be basic). **5) `sp_GetBranchSummary`** – A stored procedure to return **branch-level summary report**.• Input parameters: `@BranchID` (optional – if NULL, show

	summary for all branches).• Tasks: For each relevant branch: (a) total number of customers, (b) total number of accounts, (c) total account balance, (d) total loan amount. Return this as a result set.
Key Questions / Analysis / Interpretation to be Evaluated During/After Implementation	• How do **stored procedures** differ from plain SQL scripts and from functions? • Why are stored procedures useful for **banking workflows** (fund transfer, loan disbursement, etc.)? • How do **transactions (BEGIN TRAN / COMMIT / ROLLBACK)** ensure atomicity in fund transfers? • Why are **input parameters** essential for reusability of stored procedures? • In what situations should we **return result sets** vs **output parameters** from a stored procedure?
Supplementary Problems (Advanced Level – Limited but Deep)	1) Extend `sp_TransferFunds` to also log an entry into a **Transfer_Audit** table with details such as `FromAccountNo, ToAccountNo, Amount, PerformedByEmpID, TransferTime`. 2) Create `sp_CloseAccount(@AccountNo)` that **closes an account** only if its balance is zero and there are **no pending loans** linked to that customer; otherwise return an appropriate error message. 3) Create `sp_UpdateInterestRate(@LoanType, @NewRate)` to **bulk-update InterestRate** for all loans of a given type, and ensure this operation is wrapped in a transaction.
Key Skills to be addressed	• Writing stored procedures with **parameters** • Implementing **transaction control (COMMIT/ROLLBACK)** in SPs • Building **modular business operations** at the database level • Handling **error/validation conditions** inside stored procedures
Applications	• Core banking operations like **fund transfer, account opening, loan posting** • Backend APIs for mobile/internet banking systems • Batch updates and maintenance operations (e.g., interest rate changes) • Centralizing business logic in the database for consistency
Learning Outcome	After completing this practical, you will be able to: (1) Create stored procedures that perform **complex multi-step banking operations**, (2) Use **transactions within stored procedures** to ensure data consistency in critical operations like fund transfer (3) Design database-side logic that can be called from applications in a **secure and reusable** way.
Dataset/Test Data (Source and Description If Applicable)	Use the **Global Trust Bank** schema with: multiple customers, accounts, loans, and transactions. Ensure some accounts have **high balances**, some **low balances**, and **multiple branches**, so that fund transfers, account opening, and loan disbursement can be meaningfully tested.
Tools/Technology To Be Used	SQL Server / PostgreSQL / MySQL (use the stored procedure syntax specific to your DBMS)
Total Hours of Problem Definition Implementation	**2 Hours**
Total Hours of Engagement	**2 Hours**
Post Laboratory Work Description (Advanced – Similar but Limited Set)	1) Design and implement one additional stored procedure of your choice, such as `sp_GetCustomer360View(@CustomerID)` that returns basic details,

	accounts, loans, and recent transactions for a customer (you may return multiple result sets or a combined view). 2) Write a **short note (½–1 page)** explaining how your `sp_TransferFunds` procedure satisfies the **ACID properties** (Atomicity, Consistency, Isolation, Durability). Include at least one example scenario where `ROLLBACK` prevents inconsistent state.
Rubrics (Practical Evaluation/Viva)	Correct creation and execution of SPs – **35%** Proper use of transactions – **25%** Business logic accuracy & validations – **20%** Post-lab extra SP & ACID explanation – **10%** Viva / Conceptual clarity – **10%**

Field	Content (For Students)
Practical Number	Practical – 11
CO/PO	CO4, CO5 • PO1, PO2, PO3, PO4, PO5
Problem Definition (Advanced Scenario)	In this practical, you will work with the **AdventureWorks** sample database to design a **Supply Chain Risk Assessment Model** using SQL. AdventureWorks is experiencing delays and cost anomalies in its supply chain. You are acting as a **Lead Data Analyst**, and your job is to identify **risky vendors, delayed products, and cost anomalies** using real data. You will use tables such as `PurchaseOrderHeader`, `PurchaseOrderDetail`, `Vendor`, `Product`, etc. to build metrics like **Vendor Reliability Score, Delay Severity, and Cost Anomaly Indicators**. Your analysis should help management decide **which vendors to trust, which to monitor, and which to potentially drop**.
Key Questions / Analysis / Interpretation	• Which delays have **highest business impact** (by value, product importance, or frequency)? • How can we **quantify vendor reliability** using delivery history? • Are **high order quantities** correlated with more delays? • Are observed **cost fluctuations normal** (market-driven) or potential anomalies? • If vendor count is reduced by 40%, what **SQL-backed criteria** justify which vendors stay?
Tasks / Problems to be Implemented (Core Set)	**Task 1 – Vendor Delivery Delay Score** • Using `PurchaseOrderHeader`, `PurchaseOrderDetail`, and `Vendor`:– Compute **expected vs actual delivery time** per purchase order. – Calculate **delay (in days)** per order and per product. – Derive a **Vendor Reliability Score** = On-TimeDeliveries / TotalDeliveries. – List **Top 10 most delayed products** and associated vendors. **Task 2 – Cost Anomaly Detection** • Compare `StandardCost` vs `ActualCost` (or related cost fields) for products:– Use aggregates or window functions (e.g. moving average of cost). – Flag products where **cost has increased > 20% above recent norms**. – Produce a list of **products + vendors** with potential cost anomalies and add your interpretation as comments in your submission. **Task 3 – Vendor Reduction Strategy Simulation** • Assume management wants to **reduce the number of vendors by 40%**:

	– Propose a **composite score** for each vendor using factors such as: delivery reliability, average delay, cost stability, and total order volume. – Write queries to compute this score and **rank vendors**. – Identify which vendors would be **retained** vs **removed** if only the top 60% by score remain. **Task 4 – Risk Dashboard Query Set** • Design a **set of summary queries** that could feed a dashboard: – List **high-risk vendors** (low reliability / high anomaly count). – Identify **high-risk products** (often delayed or cost-unstable). – Show **monthly delay trends** (e.g. avg delay per month for last 12 months).
Supplementary Problems (Advanced)	1) Build a query to identify **vendors that are "strategic but risky"** – high sales volume but also high delay or anomaly score. 2) Compute a **"Supply Chain Risk Index"** per product combining delays and cost anomalies; classify products into **Low, Medium, High Risk** buckets. 3) Design a query to simulate the **impact on delays** if the bottom 20% vendors (by reliability) are removed (e.g. how many orders would need reallocation).
Key Skills to be addressed	Analytical SQL on large schema, multi-table joins, window functions / advanced aggregates, KPI design, risk scoring, business interpretation.
Applications	Vendor evaluation, supply chain optimization, operational risk management, performance benchmarking.
Learning Outcome	After completing this practical, you will be able to: (1) Use SQL to evaluate **supply chain performance**; (2) Design **custom risk metrics** such as vendor reliability and cost anomaly scores; (3) Interpret multi-table relationships to support **management decisions** like vendor reduction or renegotiation.
Dataset/Test Data (Source and Description If Applicable)	**AdventureWorks2019 / AdventureWorks2022** sample database (MSSQL). Use **Purchasing**, **Production**, and **Vendor** related tables as required.
Tools/Technology To Be Used	Microsoft SQL Server, SSMS.
Total Hours of Problem Definition Implementation	**2 Hours**
Total Hours of Engagement	**2 Hours**
Post Laboratory Work Description	• Prepare a **1–2 page brief** summarizing: key risky vendors, risky products, and your proposed vendor reduction strategy (with SQL results as evidence). • Suggest **two management actions** (e.g., renegotiate with certain vendors, diversify suppliers for certain products) based on your analysis.
Rubrics (Practical Evaluation/Viva)	Accuracy of computed metrics – **20%** · Creativity in risk scoring – **20%** · Depth of analysis & justification – **30%** · SQL complexity & efficiency – **20%** · Report clarity (post-lab) – **10%**

Field	Content (For Students)
Practical Number	Practical – 12

CO/PO	CO4, CO5 • PO1, PO2, PO3, PO4
Problem Definition (Advanced Scenario)	In this practical, you will use the **Northwind** database to design a **Customer Profitability and Churn Detection Model** using SQL. Northwind Traders wants to classify customers into **High-Value, Medium-Value, At-Risk, and Churned** based on order history, frequency, recency, and profitability. You will act as a **Data Analyst** and propose your own **formulas and segmentation rules** using tables like `Customers`, `Orders`, `Order Details`, `Products`, and `Categories`.
Key Questions / Analysis / Interpretation	• How do you **mathematically define** a "loyal" or "high-value" customer? • Which is more important for churn: **recency, frequency, or monetary value**? • How do **seasonal patterns** (month/quarter) affect churn analysis? • How do **bulk orders** distort average profitability measures? • How should the business **prioritize customers** for retention campaigns?
Tasks / Problems to be Implemented (Core Set)	**Task 1 – Define and Compute Customer Lifetime Value (CLV)** • Design your own **CLV formula** using components like: – Total order value (sum of `UnitPrice * Quantity * (1-Discount)` per customer). – Order frequency (number of orders). – Recency (days since last order). – Optional: product/category-level margins if you approximate them. • Implement SQL queries to compute a **CLV score per customer** and generate a **ranked list of customers** by CLV. **Task 2 – Detect Declining / At-Risk Customers** • Compare customer activity across **time windows** (e.g. last year vs previous years). • Identify customers whose order **frequency dropped by more than 40%** or whose monetary value significantly declined. • Output a list of **"At-Risk" customers** with key metrics (previous frequency, current frequency, % drop). **Task 3 – Market Basket Insights (without ML)** • Using `Orders` and `Order Details`, identify **frequently co-purchased products** (simple market-basket style analysis): – Example: find **top product pairs** that appear together in many orders. – Suggest **bundle/discount strategies** based on your findings. **Task 4 – Profitability Segmentation** • Based on CLV and/or other metrics, define clear **segments** (e.g. `Premium`, `Standard`, `Low`, `Churned`). • Implement SQL queries that assign each customer to a **segment**. • Provide a brief **rationale for your boundaries** (e.g. top 10% CLV = Premium, bottom 20% inactive = Churned).
Supplementary Problems (Advanced)	1) Incorporate **category-level analysis**: which **product categories** contribute the most to high-value customers? 2) Identify **seasonal top customers** (e.g. best customers each quarter) and see if they remain active across seasons.

	3) Design a basic **retention strategy suggestion table** (CustomerID + RecommendedAction: "Call", "Email Offer", "Win-back Campaign") using your segment classification.
Key Skills to be addressed	KPI and formula design (CLV, churn), advanced joins, time-window comparisons, market-basket pattern analysis, customer segmentation.
Applications	Customer retention strategies, targeted marketing, loyalty program design, revenue-focused segmentation.
Learning Outcome	After this practical, you will be able to: (1) Define and compute **custom profitability metrics (CLV)** in SQL, (2) Detect **declining or at-risk customers** using behavioral patterns. (3) Segment customers into meaningful groups to support **data-driven marketing and retention decisions**.
Dataset/Test Data (Source and Description If Applicable)	**Northwind (SQL Server version)** – use `Customers`, `Orders`, `Order Details`, `Products`, `Categories`, `Employees` if needed.
Tools/Technology To Be Used	Microsoft SQL Server, SSMS.
Total Hours of Problem Definition Implementation	**2 Hours**
Total Hours of Engagement	**2 Hours**
Post Laboratory Work Description	• Prepare a **short report** (1–2 pages) summarizing: – Your CLV formula and reasoning. – Top 10 high-value customers and your retention suggestions. – Example list of 10 at-risk customers with suggested actions. • Optionally, suggest **simple loyalty tiers** (e.g. Silver, Gold, Platinum) based on your segments.
Rubrics (Practical Evaluation/Viva)	Validity of custom formulas (CLV, segments) – **30%** · Accuracy of segmentation – **25%** · Insightfulness of churn detection – **25%** · SQL complexity – **10%** · Business justification & report – **10%**

Field	Content (For Students)
Practical Number	Practical – 13
CO/PO	CO4, CO5 • PO1, PO2, PO3, PO4, PO5
Problem Definition (Advanced Scenario)	In this practical, you will use the **Chinook** database (digital music store similar to iTunes/Spotify) to design an **SQL-based Fraud Detection & Anomaly Analysis Model**. The company suspects fraudulent behavior such as **abnormally frequent purchases, geographic inconsistencies, and synthetic (bot) accounts**. You will act as a **Fraud Analyst** and design SQL queries and scoring models that flag **high-risk customers and transactions**.
Key Questions / Analysis / Interpretation	• What behavioral patterns strongly suggest **fraud** vs just **high usage**? • How do **frequent purchases in short time intervals** indicate possible bots/scripts? • Should all **country changes** be treated as fraud, or can they be legit (e.g. VPN, travel)?

	• How can we use simple **heuristics** (rules) and scores instead of complex ML models? • What is an acceptable **false positive rate** for fraud alerts?
Tasks / Problems to be Implemented (Core Set)	Use key Chinook tables: `Customer`, `Invoice`, `InvoiceLine`, `Track`, `Album`, `Employee` etc. **Task 1 – Abnormal Purchase Frequency Detection** • Compute **average purchase frequency** per customer and compare to overall averages: – Find customers whose **number of invoices** or **number of tracks purchased** is **> 3 times the global average**. – Detect customers with **very short intervals** between their invoices (e.g. multiple invoices within minutes or within a very short time window). – Produce a list of **potential high-frequency anomalies** with metrics (total invoices, avg time gap, max burst). **Task 2 – Geographical Anomaly Detection** • Using `Customer` and `Invoice` (with `BillingCountry`): – Detect customers who have invoices from **multiple countries within a 24–48 hour period**. – List such customers with the sequence of countries and timestamps, interpreted as **geo anomalies**. **Task 3 – Synthetic/Bot Customer Detection (Heuristic Rules)** • Analyze patterns in `Customer` data to find **possible synthetic accounts**: – Identify multiple customers sharing the **same address** or suspiciously similar emails (e.g. `user1234@test.com`, `user1235@test.com`). – Design SQL heuristics that group customers by **email pattern / domain / address** and flag clusters that look like bots. **Task 4 – Fraud Severity Score** • Propose a **Fraud Severity Scoring Model** combining: – Frequency Score (based on volume & time gaps) .– Geography Score (number of distinct countries and rapid switches). – Pattern Score (email/address pattern suspiciousness) • Implement SQL queries that compute a **composite fraud score per customer** and rank customers into **Low / Medium / High Risk**.
Supplementary Problems (Advanced)	1) Extend your fraud model to consider **refunds or chargebacks** if your Chinook version has such info (or simulate with flags). 2) Identify **"False Positive Candidates"** – customers flagged as high-frequency but with **long customer lifetime** and **steady behavior** (likely loyal power users, not fraud). 3) Prepare a small **Fraud Casebook** table with 5–10 example customers: their metrics, fraud score, your final verdict (Fraud / Suspicious / Clean) and a brief reason.
Key Skills to be addressed	Pattern detection via SQL, anomaly detection using thresholds, multi-criteria scoring, customer behavior analysis, fraud reasoning.
Applications	Fraud detection in digital commerce, anomaly alerts, risk-based authentication, investigation queue prioritization.
Learning Outcome	After this practical, you will be able to:

	(1) Detect **suspicious purchase patterns** using SQL alone, (2) Design a simple **fraud scoring model** combining multiple behavioral signals (3) Interpret anomaly results to distinguish between **fraud, heavy usage, and normal variation**.
Dataset/Test Data (Source and Description If Applicable)	**Chinook (MSSQL Edition)** – use `Customer`, `Invoice`, `InvoiceLine`, `Track`, `Album` tables primarily; optionally use `Employee` or other tables for extra context.
Tools/Technology To Be Used	Microsoft SQL Server, SSMS.
Total Hours of Problem Definition Implementation	**2 Hours**
Total Hours of Engagement	**2 Hours**
Post Laboratory Work Description	• Prepare a **Fraud Risk Report** summarizing: – Top 10 highest-risk customers with scores. – At least 2–3 example "suspicious but likely genuine" customers. – Short recommendations on **what actions** to take (e.g. manual review, OTP verification, block, monitor).
Rubrics (Practical Evaluation/Viva)	Quality of anomaly logic – **30%** · Design of fraud scoring model – **25%** · Depth of insights & examples – **20%** · SQL implementation & correctness – **15%** · Viva / Explanation of reasoning – **10%**