**UNIVERSITY INSTITUTE OF COMPUTING**

**CASE STUDY REPORT**

**ON**

**PARTICULAR CASE STUDY**

Program Name: BCA

Subject Name/Code: Database Management System (23CAT-251)

**Submitted by: Submitted to:**

**Name:** Ansh Gupta **Name: Mr. Arvinder Singh**

**UID:23BCA**10724 **Designation: Professor**

**Section:4’a’**

ABSTRACT

* Introduction:

📘 Introduction: Online Banking System – Database Project

In today’s fast-paced digital era, the banking industry heavily relies on robust data management systems to ensure the secure and efficient handling of vast amounts of financial and customer information. This project focuses on the design and development of a relational database management system (RDBMS) for an Online Banking System. The goal is to simulate the core functionalities of a real-world bank, enabling operations such as account management, transactions, loan processing, and customer support.

The system is built using Structured Query Language (SQL) and follows proper normalization principles to minimize redundancy and maintain data integrity. It is designed to support essential database operations like insertion, deletion, updating, searching, and aggregating data. Relationships between entities such as customers, accounts, transactions, and loans are clearly defined through foreign key constraints and visualized using an Entity Relationship (ER) diagram.

By the end of the project, the system demonstrates how a well-designed database can serve as the backbone of a banking application, supporting both operational tasks and strategic decision-making.

* Technique:

Here are the key techniques used in the Online Banking Database System we just created:

**🔹 1. Relational Database Design**

* Technique: Normalization (up to 3NF)
* Purpose: Reduces redundancy and ensures data integrity by organizing data into multiple related tables using primary and foreign keys.

**🔹 2. Entity-Relationship Modelling (ER Diagram)**

* Technique: ER Modelling using entities, attributes, and relationships
* Purpose: Visualizes the data structure, helping define one-to-many and many-to-one relationships clearly.

**🔹 3. Data Integrity & Constraints**

* Techniques Used:
  + PRIMARY KEY
  + FOREIGN KEY
  + UNIQUE
  + NOT NULL
  + CHECK
  + DEFAULT
* Purpose: Maintains data validity, accuracy, and referential integrity between tables.

**🔹 4. Structured Query Language (SQL)**

* Techniques:
  + DDL (Data Definition Language): CREATE, ALTER, DROP
  + DML (Data Manipulation Language): INSERT, SELECT, UPDATE, DELETE
  + DCL (Data Control Language): GRANT, REVOKE (not covered above but used in real apps)
* Purpose: To create structure, manipulate, and control access to data.

**🔹 5. Aggregation & Grouping**

* Technique: SQL aggregation functions (SUM, AVG, COUNT, GROUP BY)
* Purpose: Derives insights from data, such as total balances, average loan amounts, and transaction statistics.

**🔹 6. Joins**

* Techniques:
  + INNER JOIN
  + LEFT JOIN (can be used)
  + SELF JOIN (for hierarchical relationships like account transfers)
* Purpose: Combines data from multiple related tables for meaningful output.

**🔹 7. Cascading Actions**

* Technique: ON DELETE CASCADE, ON DELETE SET NULL
* Purpose: Automatically updates or deletes dependent data, maintaining referential integrity.

**🔹 8. Transaction Management (Advanced, not shown yet)**

* Technique: BEGIN, COMMIT, ROLLBACK
* Purpose: Ensures atomicity and consistency during multi-step operations like fund transfers.

**🔹 9. Indexing (Optional/Advanced)**

* Technique: CREATE INDEX
* Purpose: Improves query performance, especially for searching and joining on large datasets.
* System Configuration:

🌐 System Configuration refers to the hardware, software, database, and networking setup that supports the functioning of the system.

Here’s what’s typically used:

**🔹 1. Database Management System (DBMS)**

* Software: MySQL / PostgreSQL / Oracle / SQL Server
* Configuration Settings:
  + max\_connections (controls concurrent users)
  + innodb\_buffer\_pool\_size (for performance tuning)
  + query\_cache\_size (for read-heavy systems)
* Purpose: To store and manage relational data, enforce constraints, and run queries.

**🔹 2. Server Environment**

* OS: Linux (Ubuntu/CentOS) or Windows Server
* Web Server: Apache / NGINX (if connected to a frontend)
* Host Machine Config:
  + CPU: Quad-Core or higher
  + RAM: 8GB or more (depends on traffic)
  + Storage: SSD with minimum 100GB for production
* Purpose: Host the database and optionally serve APIs or web frontend.

**🔹 3. Application Layer (optional if using full-stack)**

* Language: Python (Flask, Django), PHP (Laravel), Java (Spring Boot), Node.js
* Framework: MVC pattern
* API Server: RESTful or GraphQL endpoints
* Purpose: Middleman between frontend and DB; handles logic, security, and transactions.

**🔹 4. Frontend (for user interface)**

* Language: HTML/CSS/JavaScript
* Framework: React.js, Angular, Vue.js
* Purpose: To display user-friendly forms, dashboards, and analytics.

**🔹 5. Network Configuration**

* IP Configuration: Static IP for server, firewall setup
* Port Setup:
  + 3306 for MySQL
  + 5432 for PostgreSQL
  + 80/443 for web servers
* Secure Protocols: HTTPS, SSL/TLS, VPN
* Purpose: Secure, fast, and reliable communication between users and database.

**🔹 6. Security Configuration**

* Access Control: User roles, GRANT/REVOKE permissions in SQL
* Backups: Daily snapshots using mysqldump or pg\_dump
* Firewall: Allow only specific ports and IPs
* Data Encryption: At-rest and in-transit encryption using SSL
* Monitoring Tools: Fail2ban, UFW, Zabbix, Nagios

**🔹 7. Backup and Recovery**

* Daily Backup Scripts: Automated using corn jobs
* Recovery Plan: Full recovery using dump files or replication
* Redundancy: RAID setup or cloud replication (e.g., AWS RDS Multi-AZ)

**🔹 8. Cloud Configuration (if deployed on cloud)**

* Provider: AWS / Azure / Google Cloud
* Services:
  + Amazon RDS (for DB)
  + EC2 (for backend app)
  + S3 (for backup storage)
* Auto-scaling, load balancing, monitoring via CloudWatch
* INPUT:

In the context of the Online Banking Database System, the INPUT refers to all the data and information that is entered into the system by users, employees, or other services.

Here’s a categorized list of inputs used in the system:

**🔹 1. Customer Inputs**

* Name, Email, Phone number
* Address
* Date of Birth (DOB)
* Login credentials (in a full system)
* Uploaded documents (ID proofs, etc.) – in real-world apps

**🔹 2. Account Creation Inputs**

* Selected account type (Savings, Current)
* Initial deposit amount
* Customer ID (linked to the account)
* Account status (Active, Suspended, Closed)

**🔹 3. Transaction Inputs**

* Source account number
* Transaction type (Deposit, Withdrawal, Transfer)
* Amount
* Receiver’s account number (for transfers)
* Date (auto-generated or input)

**🔹 4. Loan Application Inputs**

* Loan type (Home, Car, Personal)
* Loan amount
* Interest rate (fixed or variable)
* Customer ID
* Loan status (Pending, Approved, Rejected)

**🔹 5. Employee/Admin Inputs**

* Employee name, email, and role
* Assigned support tickets
* Loan approvals or account status changes

**🔹 6. Customer Support Inputs**

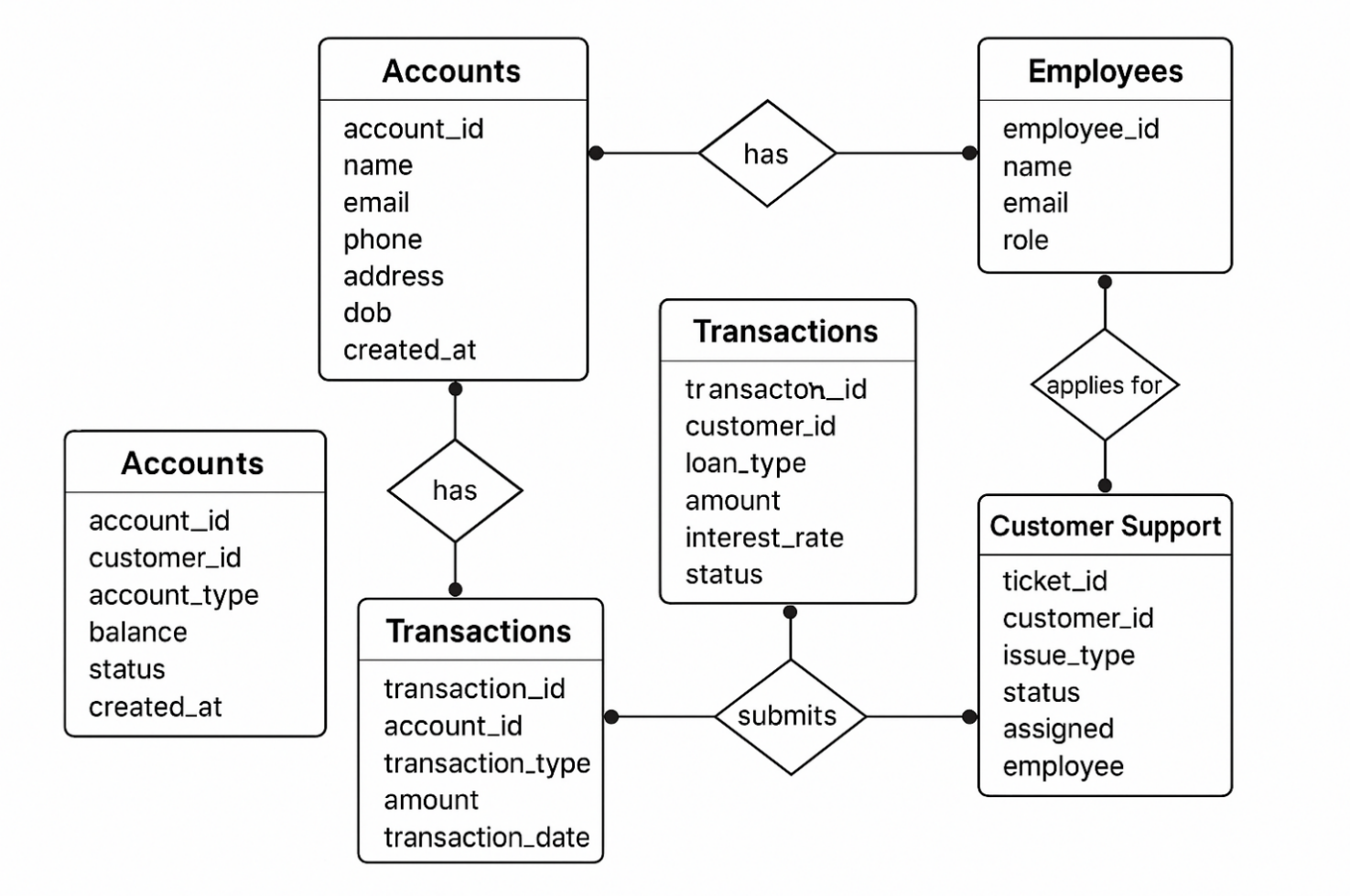
* Customer ID
* Issue type (Technical, Fraud, Inquiry)
* Status of the ticket
* Assigned employee ID

**🔹 7. System/Automated Inputs**

* Timestamps (e.g., created at, transaction date)
* Auto-increment IDs (customer\_id, account\_id, etc.)
* System-generated alerts (low balance, loan status updates)
* Default values (e.g., account balance = 0)

**🔹 8. Frontend UI Inputs (if using a user interface)**

* Registration and login forms
* Buttons (submit, update, delete)
* Dropdowns (for account type, issue type, etc.)
* Search boxes and filters
* ER DIAGRAM:



* TABLE REALTION:

Great! Let’s break down the 📊 Table Relationships used in the Online Banking Database System. These define how different tables are connected through foreign keys and represent how real-world entities interact with each other.

Here are the major table relationships:

**🟩 1. Customers ↔ Accounts**

* Relationship: One-to-Many
* Description: One customer can have multiple accounts.
* Foreign Key: Accounts.customer\_id → Customers.customer\_id

**🟦 2. Accounts ↔ Transactions**

* Relationship: One-to-Many
* Description: One account can be involved in many transactions.
* Foreign Key: Transactions.account\_id → Accounts.account\_id
* Additional: For transfer-type transactions, receiver\_account\_id also refers to Accounts.account\_id (self-join)

**🟨 3. Customers ↔ Loans**

* Relationship: One-to-Many
* Description: One customer can apply for multiple loans.
* Foreign Key: Loans.customer\_id → Customers.customer\_id

**🟥 4. Customers ↔ Customer Support**

* Relationship: One-to-Many
* Description: One customer can raise multiple support tickets.
* Foreign Key: CustomerSupport.customer\_id → Customers.customer\_id

**🟧 5. Employees ↔ Customer Support**

* Relationship: One-to-Many (optional)
* Description: One employee can be assigned to handle many support tickets.
* Foreign Key: CustomerSupport.assigned\_employee → Employees.employee\_id

**🔁 6. Accounts (Self-Join) in Transactions**

* Relationship: One-to-One (in transfer transactions)
* Description: A transfer involves two accounts: sender and receiver.
* Foreign Key: Transactions.receiver\_account\_id → Accounts.account\_id
* TABULAR FORMAT:

1️. Table: Customers

| **Column Name** | **Data Type** | **Constraints** |
| --- | --- | --- |
| customer\_id | INT | PRIMARY KEY, AUTO\_INCREMENT |
| name | VARCHAR (100) | NOT NULL |
| email | VARCHAR (100) | NOT NULL, UNIQUE |
| phone | VARCHAR (15) | NOT NULL, UNIQUE |
| address | TEXT | NOT NULL |
| dob | DATE | NOT NULL |
| created at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP |

* 2️. Table: Accounts

| **Column Name** | **Data Type** | **Constraints** |
| --- | --- | --- |
| account\_id | INT | PRIMARY KEY, AUTO\_INCREMENT |
| customer\_id | INT | FOREIGN KEY → Customers(customer\_id), NOT NULL |
| account type | ENUM ('Savings’, ‘Current') | NOT NULL |
| balance | DECIMAL (15,2) | DEFAULT 0.00 |
| status | ENUM('Active','Suspended','Closed') | NOT NULL |

* 3️. Table: Transactions

| **Column Name** | **Data Type** | **Constraints** |
| --- | --- | --- |
| transaction\_id | INT | PRIMARY KEY, AUTO\_INCREMENT |
| account\_id | INT | FOREIGN KEY → Accounts(account\_id), NOT NULL |
| transaction type | ENUM('Deposit','Withdrawal','Transfer') | NOT NULL |
| amount | DECIMAL (15,2) | CHECK (amount > 0), NOT NULL |
| transaction date | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP |
| receiver\_account\_id | INT | FOREIGN KEY → Accounts(account\_id), NULL (for deposits) |

* 4️. Table: Loans

| **Column Name** | **Data Type** | **Constraints** |
| --- | --- | --- |
| loan\_id | INT | PRIMARY KEY, AUTO\_INCREMENT |
| customer\_id | INT | FOREIGN KEY → Customers(customer\_id), NOT NULL |
| loan type | ENUM('Personal','Home','Car') | NOT NULL |
| amount | DECIMAL (15,2) | CHECK (amount > 0), NOT NULL |
| interest rate | DECIMAL (5,2) | CHECK (interest rate > 0), NOT NULL |
| status | ENUM('Approved','Pending','Rejected') | NOT NULL |

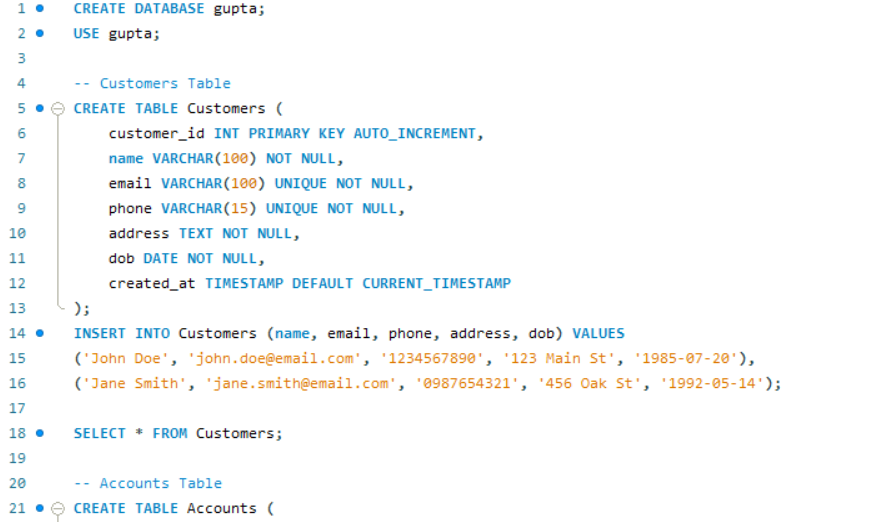
* 5️. Table: Employees

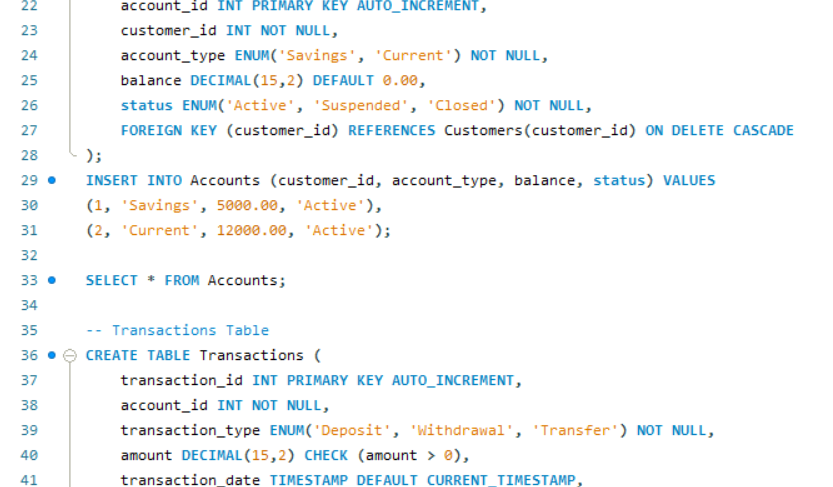
| **Column Name** | **Data Type** | **Constraints** |
| --- | --- | --- |
| employee | INT | PRIMARY KEY, AUTO\_INCREMENT |
| name | VARCHAR (100) | NOT NULL |
| email | VARCHAR (100) | NOT NULL, UNIQUE |
| role | ENUM('Manager','Clerk','Support') | NOT NULL |

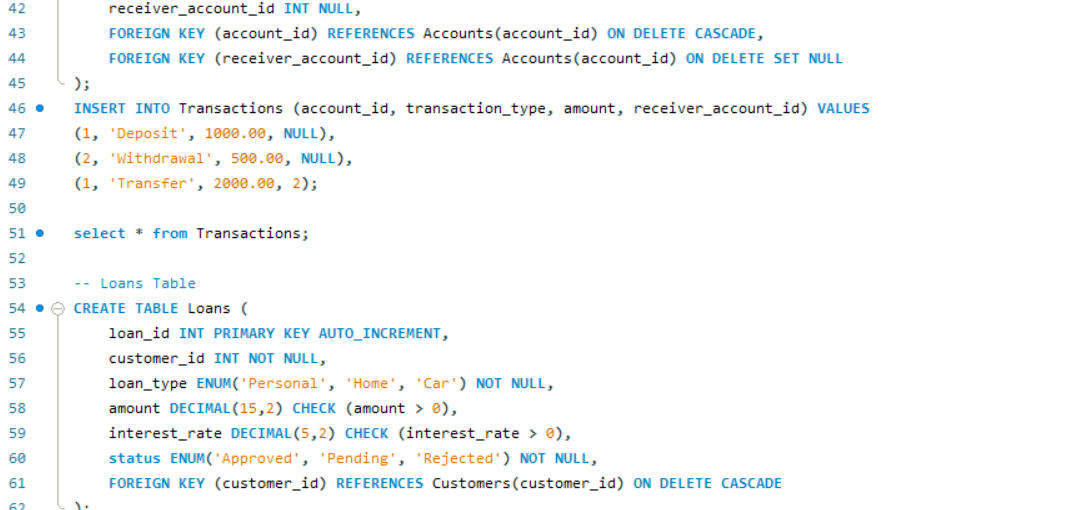
* 6️. Table: Customer Support

| **Column Name** | **Data Type** | **Constraints** |
| --- | --- | --- |
| ticket\_id | INT | PRIMARY KEY, AUTO\_INCREMENT |
| customer\_id | INT | FOREIGN KEY → Customers(customer\_id), NOT NULL |
| issue type | ENUM('Technical','Fraud','Inquiry') | NOT NULL |
| status | ENUM ('Open’, ‘In Progress’, ‘Resolved') | NOT NULL |
| assigned employee | INT | FOREIGN KEY → Employees(employee\_id), NULL (optional assignment) |

* TABLE CREATION:

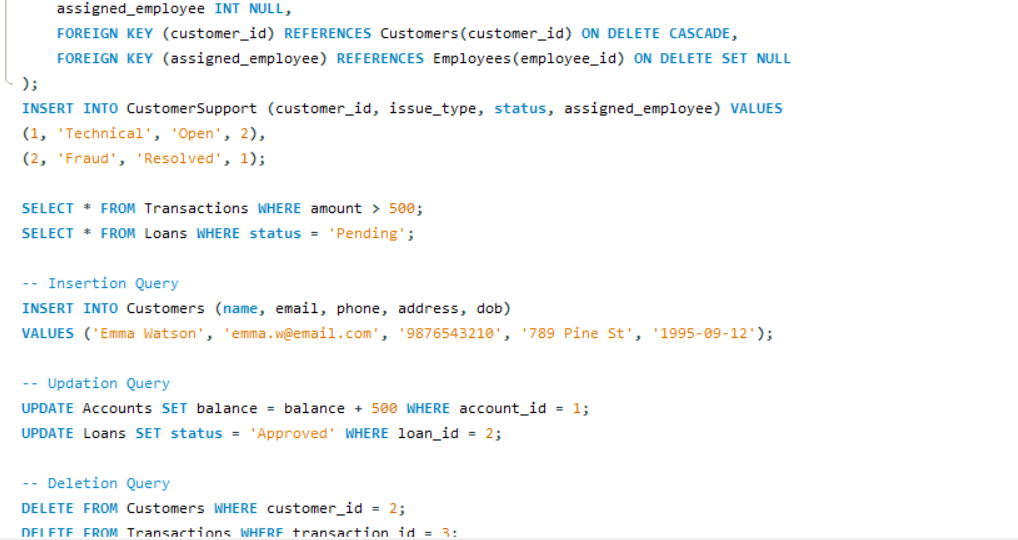


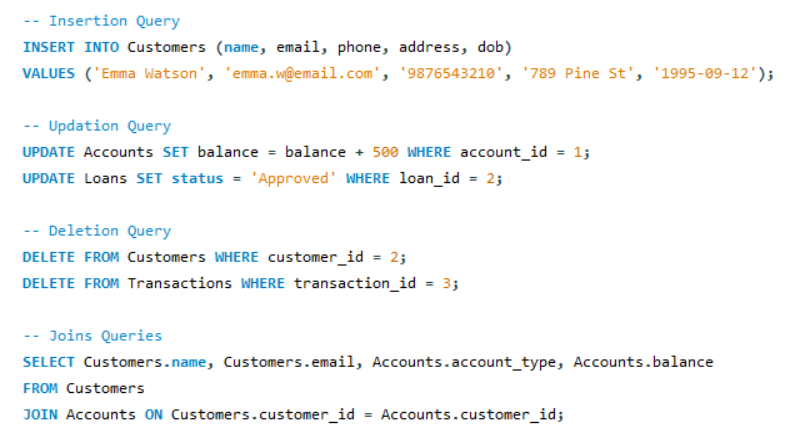


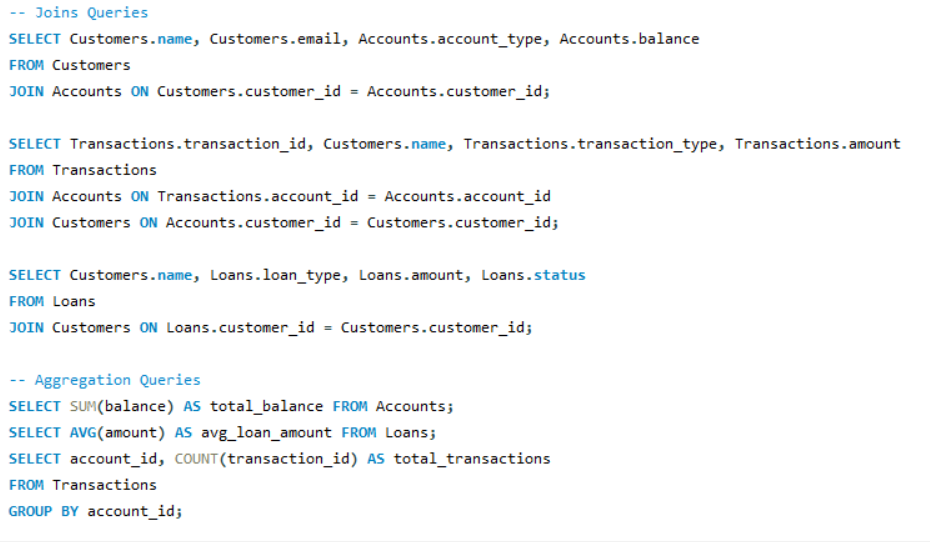


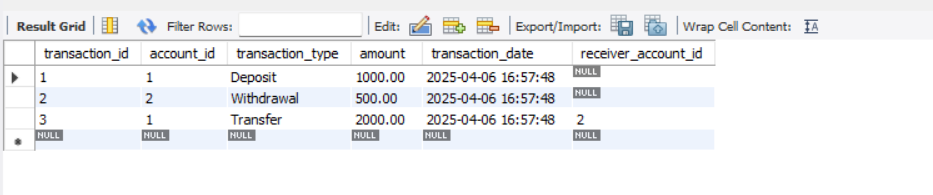


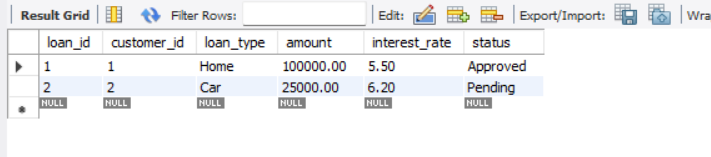
* SQL QUERIES WITH OUTPUT:











* SUMMARY:

Here's a clear and concise 📋 Summary of the Online Banking Database System project:

📌 Project Title: Online Banking Database System

🔹 Objective: To design and implement a relational database system that manages customer accounts, transactions, loans, employees, and support services in a secure and efficient way.

🔹 Real-World Application: Banking system used by financial institutions to manage:

* Customer information
* Account management
* Fund transfers and transactions
* Loan processing
* Customer support

🔹 Key Entities & Tables: 1. Customers – Stores user info (name, email, phone, DOB) 2. Accounts – Tracks account types, balances, and ownership 3. Transactions – Records deposits, withdrawals, and transfers 4. Loans – Manages customer loan applications 5. Employees – Admins or support personnel 6. Customer Support – Tickets raised by customers and handled by staff

🔹 Relationships:

* One customer can have multiple accounts and loans
* One account can have multiple transactions
* One employee can handle many support tickets

🔹 Techniques Used:

* Relational Database Design (Normalization)
* ER Diagram modelling
* SQL with DDL, DML, JOINs, Constraints
* Aggregation queries (SUM, COUNT)
* Data Integrity enforcement (PK, FK, NOT NULL, UNIQUE)

🔹 System Configuration:

* DBMS: MySQL or PostgreSQL
* OS: Linux or Windows
* Frontend: HTML/CSS/JavaScript (if needed)
* App Layer: Optional (Python/Java/.NET)
* Server: Local or cloud (AWS, Azure)

🔹 Inputs:

* Customer registration details
* Account type selection
* Transaction data (amount, type)
* Loan requests
* Support queries

🔹 Output:

* Account balances and summaries
* Transaction history
* Loan status
* Support ticket reports

🔹 SQL Operations Demonstrated:

* Table creation with constraints
* INSERT, SELECT, UPDATE, DELETE
* INNER JOIN for linked data
* GROUP BY and aggregate functions
* CONCLUSION:

The Online Banking Database System is a comprehensive and well-structured solution designed to manage the core operations of a financial institution. Through the implementation of normalized relational tables, defined relationships, and robust SQL operations, the system ensures data integrity, scalability, and reliability.

Key takeaways:

🔹 Efficient Data Management  
The system successfully handles customer profiles, multiple accounts per customer, secure transaction logs, loan applications, and employee-administered customer support—all through well-connected tables using primary and foreign keys.

🔹 Real-World Applicability  
It simulates real-world banking processes like deposits, withdrawals, transfers, and support ticket tracking, making it an excellent prototype for modern financial applications.

🔹 Use of Best Practices  
With normalization, constraints (PK, FK, NOT NULL, UNIQUE, CHECK), indexing options, and ER modelling, the design follows best database design principles for performance and data accuracy.

🔹 Flexible & Extensible  
The database is modular and can be easily extended to include features like credit cards, branch locations, mobile banking, and fraud detection systems in the future.

🔹 Foundation for Full Applications  
This backend database can seamlessly integrate with front-end interfaces and APIs, enabling full-stack banking applications for customers and employees alike.

🎯 Final Thought:  
This project not only demonstrates core database concepts but also lays the foundation for a real-world banking application that prioritizes structure, security, and service delivery.

 https://github.com/anshgupta110805/online-banking