Day 2: Banking & Finance SQL Practice

Scenario Description

Business Context:

You are part of a data engineering team at a bank. The organization manages thousands of customer accounts and daily transactions such as deposits, withdrawals, and transfers. Analysts rely on SQL queries to identify high-value customers, monitor failed transactions, and analyze account activities.

Why it matters:

- Monitoring customer transactions helps detect fraud or anomalies.
- Calculating balances accurately is vital for compliance and reporting.
- Identifying inactive or high-risk customers supports better business decisions.

Database Schema

Tables

1. **Customers**

```
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    join_date DATE NOT NULL
);
```

1. **Accounts**

```
CREATE TABLE Accounts (
    account_id INT PRIMARY KEY,
    customer_id INT NOT NULL,
    account_type VARCHAR(20) CHECK(account_type IN ('Savings', 'Current',
'Loan')),
    balance DECIMAL(12,2) DEFAULT 0.00,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);
```

1. **Transactions**

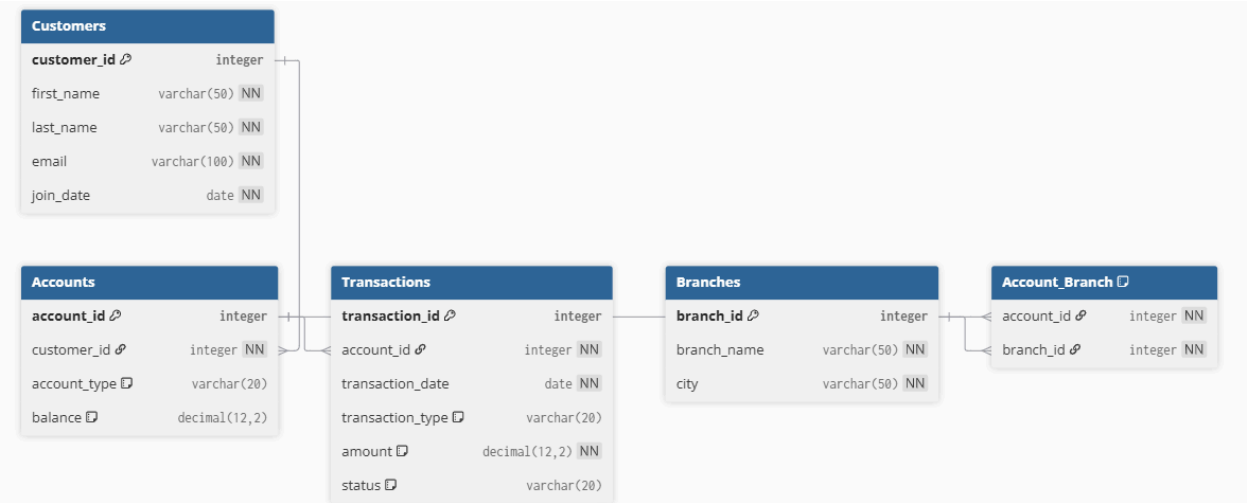
```
CREATE TABLE Transactions (
    transaction_id INT PRIMARY KEY,
    account_id INT NOT NULL,
    transaction_date DATE NOT NULL,
    transaction_type VARCHAR(20) CHECK(transaction_type IN ('Deposit','With
drawal','Transfer')),
    amount DECIMAL(12,2) NOT NULL CHECK(amount > 0),
    status VARCHAR(20) CHECK(status IN ('Success','Failed','Pending')),
    FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);
```

1. **Branches**

```
CREATE TABLE Branches (
    branch_id INT PRIMARY KEY,
    branch_name VARCHAR(50) NOT NULL,
    city VARCHAR(50) NOT NULL
);
```

1. **Account_Branch**

```
CREATE TABLE Account_Branch (
    account_id INT NOT NULL,
    branch_id INT NOT NULL,
    PRIMARY KEY (account_id, branch_id),
    FOREIGN KEY (account_id) REFERENCES Accounts(account_id),
    FOREIGN KEY (branch_id) REFERENCES Branches(branch_id)
);
```



Sample Data

Customers

```
INSERT INTO Customers VALUES
(1,'John','Miller','john.miller@bank.com','2023-01-15'),
(2,'Emma','Watson','emma.watson@bank.com','2023-03-20'),
(3,'Liam','Brown','liam.brown@bank.com','2024-05-10');
```

Accounts

```
INSERT INTO Accounts VALUES
(101,1,'Savings',5000.00),
```

```
(102,2,'Current',20000.00),
(103,3,'Savings',1000.00),
(104,1,'Loan',-3000.00);
```

Transactions

```
INSERT INTO Transactions VALUES
(1001,101,'2025-04-01','Deposit',2000.00,'Success'),
(1002,101,'2025-04-03','Withdrawal',1000.00,'Success'),
(1003,102,'2025-04-05','Deposit',5000.00,'Failed'),
(1004,103,'2025-04-10','Deposit',1500.00,'Success'),
(1005,104,'2025-04-12','Transfer',3000.00,'Pending');
```

Branches

```
INSERT INTO Branches VALUES
(1,'Central Branch','Mumbai'),
(2,'North Branch','Delhi'),
(3,'South Branch','Bangalore');
```

Account_Branch

```
INSERT INTO Account_Branch VALUES
(101,1),
(102,2),
(103,3),
(104,1);
```

ERD (Textual)

```
Customers (1) ──< Accounts (M)
Accounts (1) ──< Transactions (M)
Accounts (M) ──< Account_Branch (M) >── Branches (1)
```

Relationships:

- **Customers → Accounts** : 1:M
- **Accounts → Transactions** : 1:M
- **Accounts ↔ Branches** : M:N (via Account_Branch)

SQL Questions

Easy

1. List all customers and their account types.

Medium

1. Find the total successful transaction amount per account.

Hard

1. Retrieve accounts where the total withdrawals exceed deposits.

Difficult

1. Find customers who have at least one failed transaction.

Expert

1. Identify the top branch by **total transaction value** (successful only), showing branch name, city, and total amount.

Solutions with Explanations

Easy

```
SELECT c.first_name, c.last_name, a.account_type
FROM Customers c
JOIN Accounts a ON c.customer_id = a.customer_id;
```

Explanation:

- Simple join to relate customers with their accounts.
- Useful for viewing all customer-account relationships.

Medium

```
SELECT a.account_id, SUM(t.amount) AS total_successful_amount
FROM Accounts a
JOIN Transactions t ON a.account_id = t.account_id
WHERE t.status = 'Success'
GROUP BY a.account_id;
```

Explanation:

- Aggregates successful transactions per account.
- Uses `SUM()` and `GROUP BY`.
- Filtered to include only `Success` transactions.

Tip: Index `Transactions.status` for faster filtering.

Hard

```
SELECT a.account_id
FROM Accounts a
JOIN Transactions t ON a.account_id = t.account_id
GROUP BY a.account_id
HAVING SUM(CASE WHEN t.transaction_type = 'Withdrawal' AND t.status='S
uccess' THEN t.amount ELSE 0 END) >
```

```
    SUM(CASE WHEN t.transaction_type = 'Deposit' AND t.status='Success'
THEN t.amount ELSE 0 END);
```

Explanation:

- Conditional aggregation compares withdrawals vs. deposits.
- Accounts with higher withdrawals than deposits are flagged.
- `CASE WHEN` used to compute per-transaction-type totals efficiently.

Tip: Use this pattern for financial audits and balance risk detection.

Difficult

```
SELECT DISTINCT c.customer_id, c.first_name, c.last_name
FROM Customers c
JOIN Accounts a ON c.customer_id = a.customer_id
JOIN Transactions t ON a.account_id = t.account_id
WHERE t.status = 'Failed';
```

Explanation:

- Joins customers → accounts → transactions.
- `DISTINCT` ensures each customer appears once even with multiple failed transactions.
- Helps detect clients who may need issue resolution or fraud check.

Expert

```
SELECT b.branch_name, b.city, SUM(t.amount) AS total_successful_value
FROM Branches b
JOIN Account_Branch ab ON b.branch_id = ab.branch_id
JOIN Accounts a ON ab.account_id = a.account_id
JOIN Transactions t ON a.account_id = t.account_id
```

```
WHERE t.status = 'Success'
GROUP BY b.branch_name, b.city
ORDER BY total_successful_value DESC
LIMIT 1;
```

Explanation:

- Multi-table join to aggregate transaction amounts by branch.
- Considers only successful transactions.
- `LIMIT 1` returns the top-performing branch.

Tip:

Add composite index `(status, account_id)` on `Transactions` for performance on large datasets.