DATABASE SYSTEMS

LAB MANUAL

INSTRUCTOR: SIR ANWAR ALI

PREPARED BY:

MARVI SHEIKH (B2433064)

MARINA ZAFAR (B2433063)

SAAMIA (B2433104)

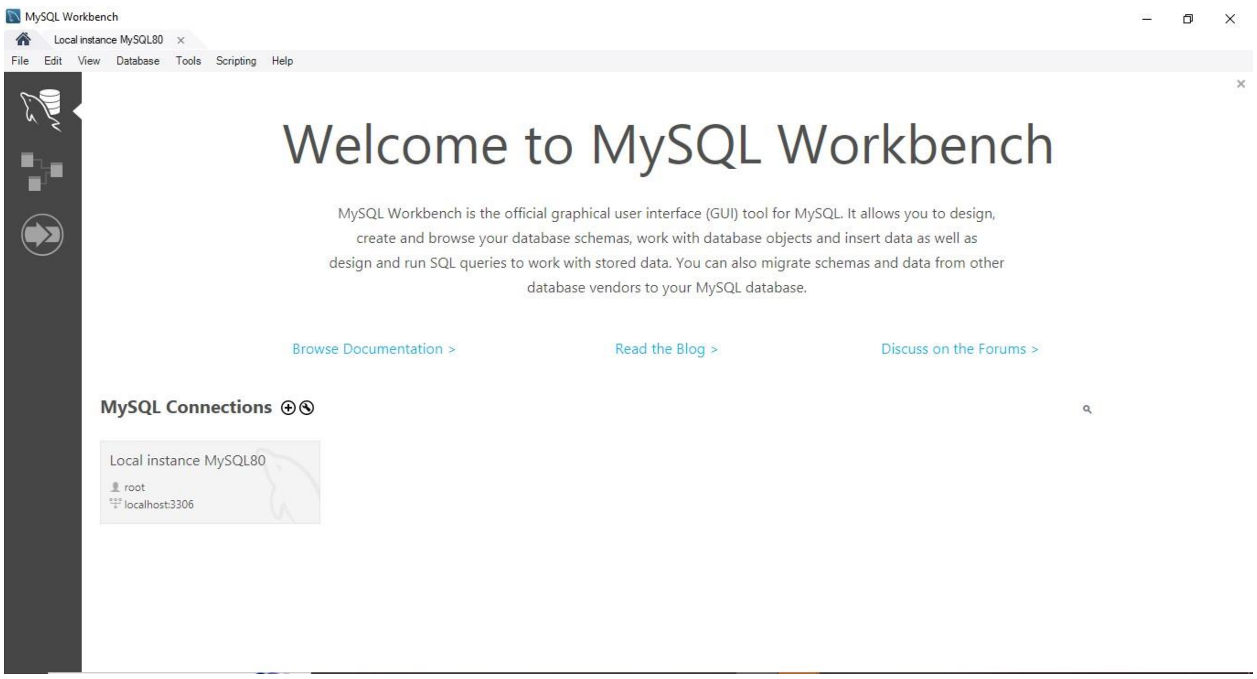
BANKING SYSTEM DATABASE

Step 1: Objective / Aim

The aim of this project is to design and implement a database system for managing a bank's core operations such as customers, accounts, loans, and transactions using MySQL Workbench.

Step 2: Tools Used

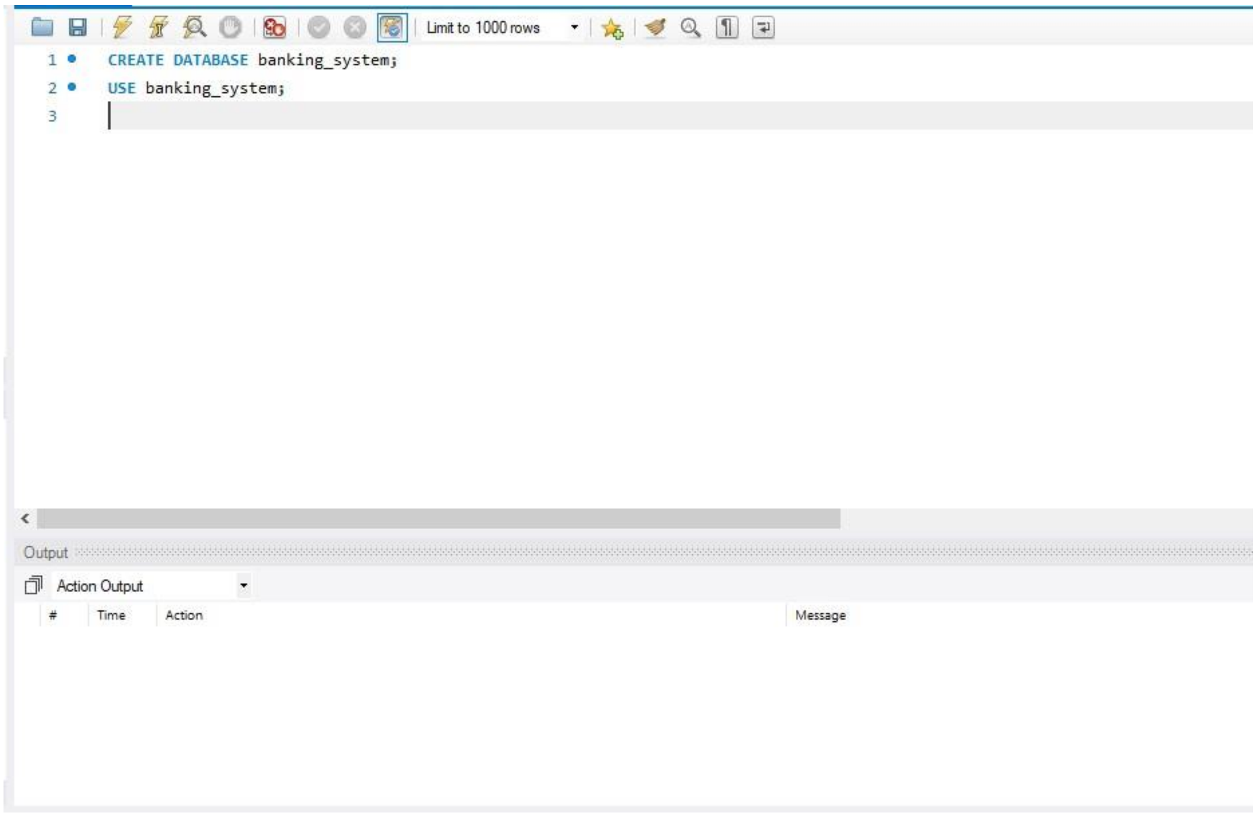
1. Software: MySQL Workbench 8.x
2. Language: SQL (Structured Query Language)
3. Platform: Windows 10 / 11
4. Hardware: Dell Latitude E6440 (Intel i5, 8 GB RAM)



Welcome to MySQL Workbench

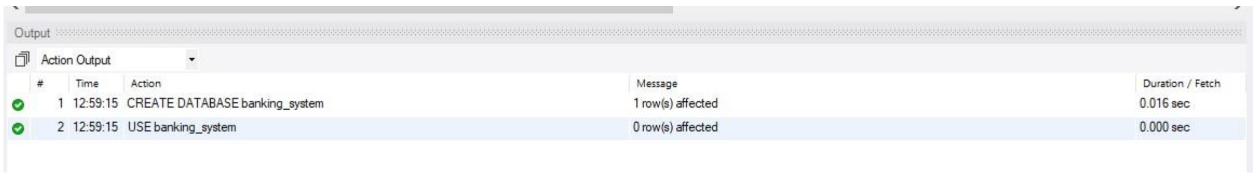
MySQL Workbench is the official graphical user interface (GUI) tool for MySQL. It allows you to design, create and browse your database schemas, work with database objects and insert data as well as design and run SQL queries to work with stored data. You can also migrate schemas and data from other database vendors to your MySQL database.

Step 3: Creating Database



```
CREATE DATABASE banking_system;
USE banking_system;
```

A new database named banking system was created to store all tables related to the banking system.



Output

Action Output

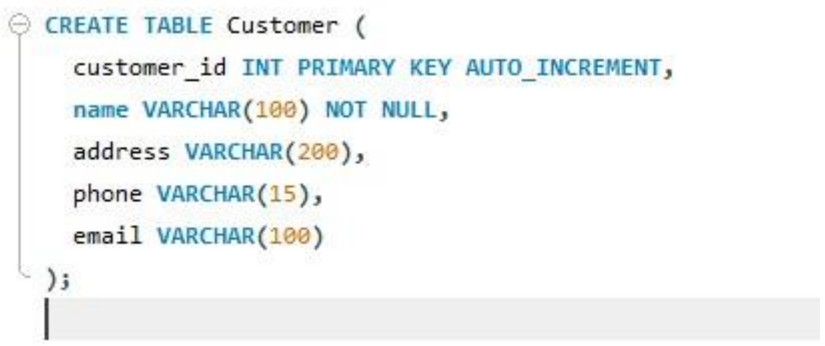
#	Time	Action	Message	Duration / Fetch
1	12:59:15	CREATE DATABASE banking_system	1 row(s) affected	0.016 sec
2	12:59:15	USE banking_system	0 row(s) affected	0.000 sec

Step 4: Creating Tables

Four main tables were created: Customer, Account, Loan, and Transaction.

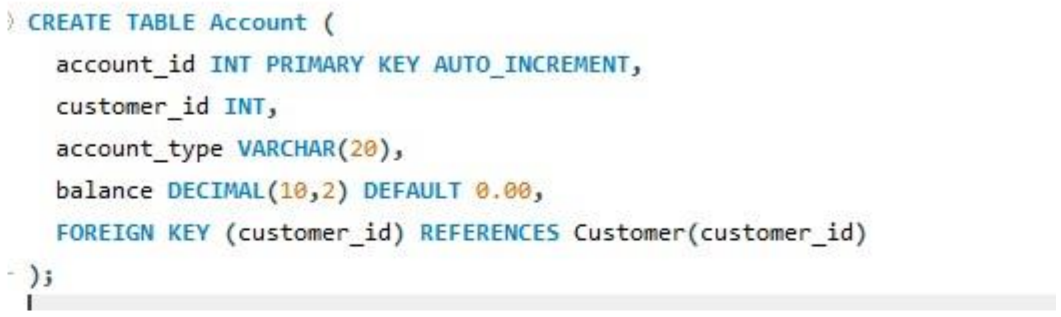
Each table has a Primary Key, and relationships are made using Foreign Keys.

1) CUSTOMER TABLE:



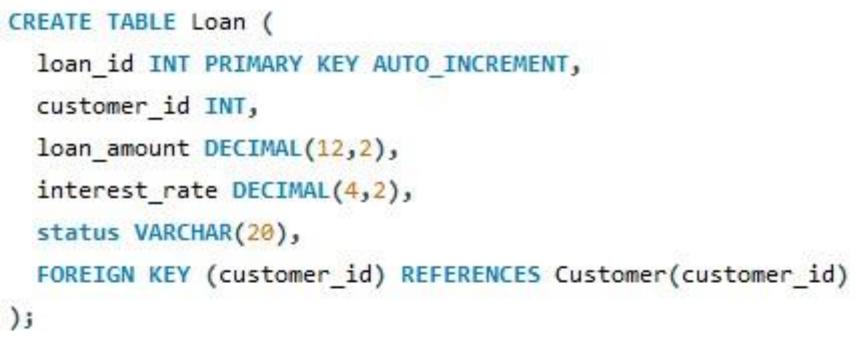
```
CREATE TABLE Customer (
  customer_id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(100) NOT NULL,
  address VARCHAR(200),
  phone VARCHAR(15),
  email VARCHAR(100)
);
```

2) ACCOUNT TABLE:



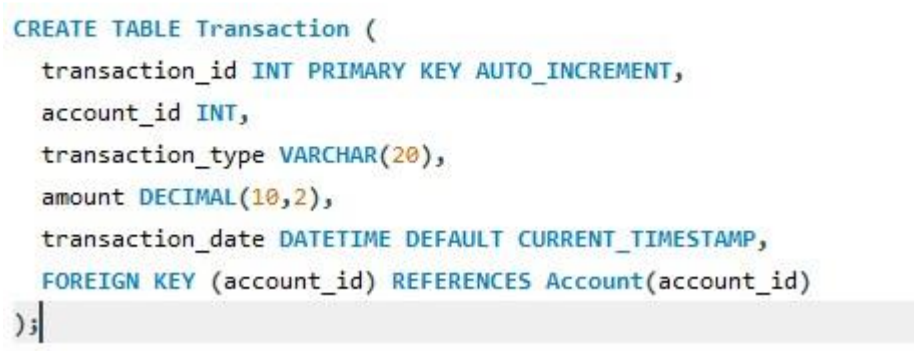
```
CREATE TABLE Account (
  account_id INT PRIMARY KEY AUTO_INCREMENT,
  customer_id INT,
  account_type VARCHAR(20),
  balance DECIMAL(10,2) DEFAULT 0.00,
  FOREIGN KEY (customer_id) REFERENCES Customer(customer_id)
);
```

3) LOAN TABLE:



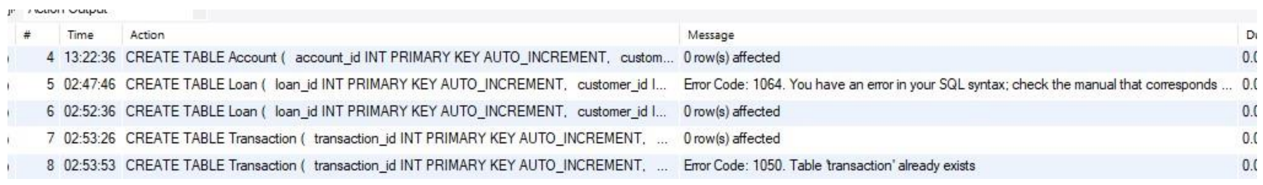
```
CREATE TABLE Loan (
  loan_id INT PRIMARY KEY AUTO_INCREMENT,
  customer_id INT,
  loan_amount DECIMAL(12,2),
  interest_rate DECIMAL(4,2),
  status VARCHAR(20),
  FOREIGN KEY (customer_id) REFERENCES Customer(customer_id)
);
```

4) TRANSACTION TABLE:



```
CREATE TABLE Transaction (
  transaction_id INT PRIMARY KEY AUTO_INCREMENT,
  account_id INT,
  transaction_type VARCHAR(20),
  amount DECIMAL(10,2),
  transaction_date DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (account_id) REFERENCES Account(account_id)
);
```

TABLES ARE CREATED:



#	Time	Action	Message	Du
4	13:22:36	CREATE TABLE Account (account_id INT PRIMARY KEY AUTO_INCREMENT, custom...	0 row(s) affected	0.0
5	02:47:46	CREATE TABLE Loan (loan_id INT PRIMARY KEY AUTO_INCREMENT, customer_id I...	Error Code: 1064. You have an error in your SQL syntax; check the manual that corresponds ...	0.0
6	02:52:36	CREATE TABLE Loan (loan_id INT PRIMARY KEY AUTO_INCREMENT, customer_id I...	0 row(s) affected	0.0
7	02:53:26	CREATE TABLE Transaction (transaction_id INT PRIMARY KEY AUTO_INCREMENT, ...	0 row(s) affected	0.0
8	02:53:53	CREATE TABLE Transaction (transaction_id INT PRIMARY KEY AUTO_INCREMENT, ...	Error Code: 1050. Table 'transaction' already exists	0.0

What is Normalization?

Normalization is the process of organizing data in a database to reduce duplication and ensure data consistency.

It divides large tables into smaller, related ones and connects them using Primary Keys and Foreign Keys.

Types of Normal Forms

- **1NF (First Normal Form)**

Each table has a Primary Key.

All values are atomic (no repeating groups).

Example: In the Customer table, each field (name, phone, email) has a single value only.

- **2NF (Second Normal Form)**

The table is already in 1NF.

All non-key columns depend only on the primary key.

Example: In the Account table, balance depends only on account_id, not on any other field.

- **3NF (Third Normal Form)**

The table is in 2NF.

There are no transitive dependencies (non-key fields don't depend on each other).

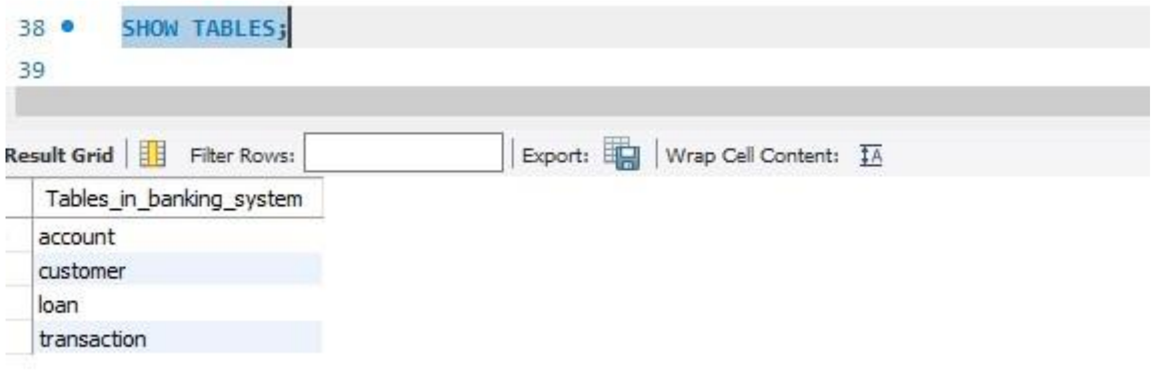
Example: In the Loan table, status depends only on loan_id, not on other fields.

HOW NORMALIZATION WAS APPLIED HERE:

- The Customer, Account, Loan, and Transaction tables were separated to avoid data repetition.
- Each table stores one type of information only.
- They are connected through foreign keys, ensuring data is consistent and easy to update.

STEP 05:

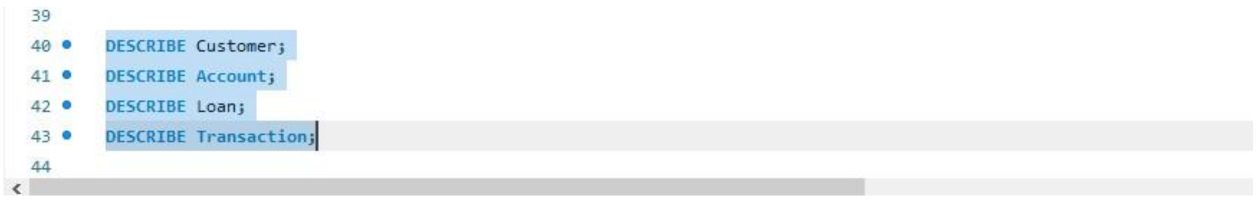
Now, to see if the tables were created, we run:



```
SHOW TABLES;
```

```
Tables_in_banking_system
account
customer
loan
transaction
```

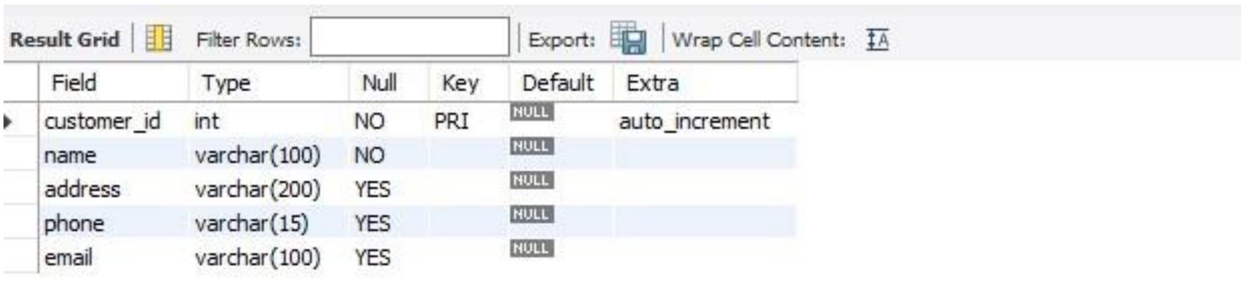
And now, to describe each table:



```
DESCRIBE Customer;
DESCRIBE Account;
DESCRIBE Loan;
DESCRIBE Transaction;
```

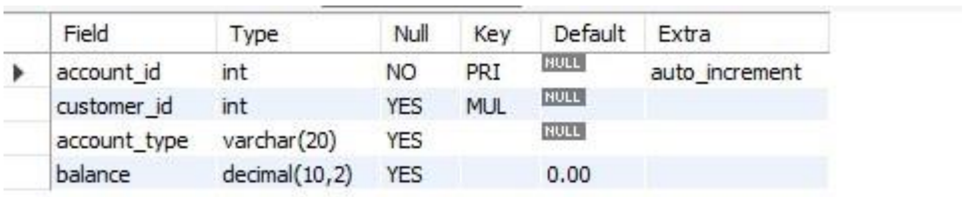
The results will appear like this:

Customer:



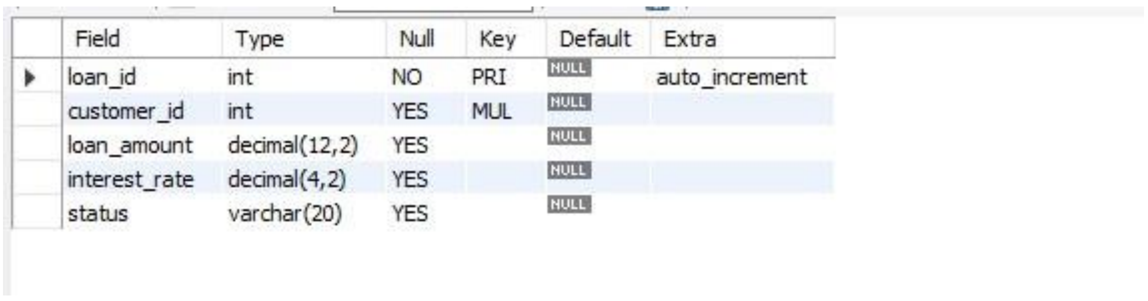
Field	Type	Null	Key	Default	Extra
customer_id	int	NO	PRI	NULL	auto_increment
name	varchar(100)	NO		NULL	
address	varchar(200)	YES		NULL	
phone	varchar(15)	YES		NULL	
email	varchar(100)	YES		NULL	

Account:



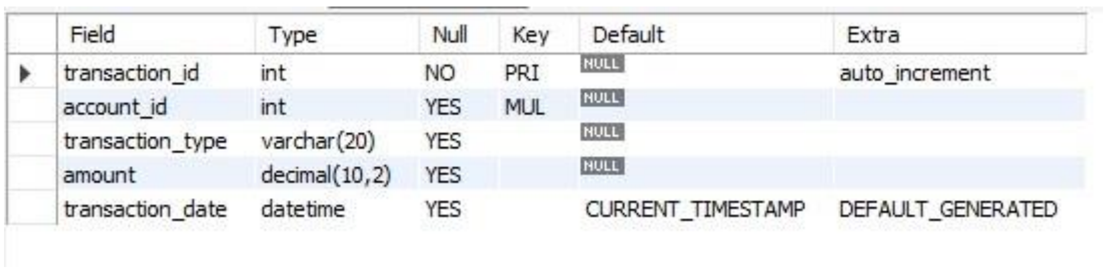
Field	Type	Null	Key	Default	Extra
account_id	int	NO	PRI	NULL	auto_increment
customer_id	int	YES	MUL	NULL	
account_type	varchar(20)	YES		NULL	
balance	decimal(10,2)	YES		0.00	

Loan:



Field	Type	Null	Key	Default	Extra
loan_id	int	NO	PRI	NULL	auto_increment
customer_id	int	YES	MUL	NULL	
loan_amount	decimal(12,2)	YES		NULL	
interest_rate	decimal(4,2)	YES		NULL	
status	varchar(20)	YES		NULL	

Transaction:



Field	Type	Null	Key	Default	Extra
transaction_id	int	NO	PRI	NULL	auto_increment
account_id	int	YES	MUL	NULL	
transaction_type	varchar(20)	YES		NULL	
amount	decimal(10,2)	YES		NULL	
transaction_date	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

DEFINING RELATIONSHIPS:

To define relationships, we need to understand the basic concepts like primary key and foreign key.

-Primary Key: A unique identifier for each record in a table (no duplicates, no nulls).
-Foreign Key: A field that links one table to another using the primary key of that other table.

Following are the possible relations in the database:

The relationships between the tables are established using Foreign Keys.

- Each **Customer** can have multiple **Accounts**. (1:N)
- Each **Customer** can have multiple **Loans**. (1:N)
- Each **Account** can have multiple **Transactions**. (1:N)

ER-DIAGRAM: ENTITY-RELATIONSHIP DIAGRAM:

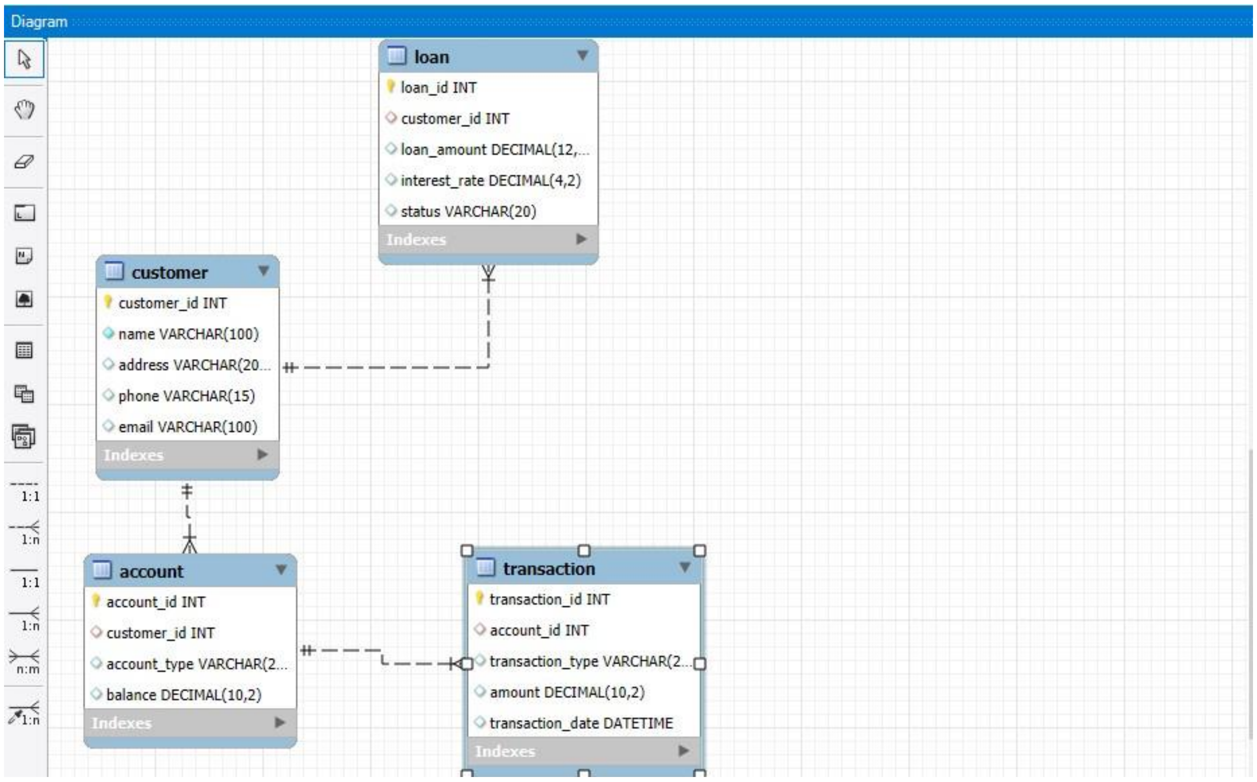
It’s a picture that shows how tables in a database are connected — like how customers, accounts, loans, and transactions are related to each other.

HOW TO VIEW ERD IN MySQL WORKBENCH:

In **MySQL workbench**, go to:
Database → Reverse Engineer → Select your database → Finish
➞ It will automatically generate an **ERD** (Entity Relationship Diagram).

You’ll see all your tables connected by lines showing the relationships.

FIGURE:



Step 06: Insert Sample Data (So Your Database Has Records)

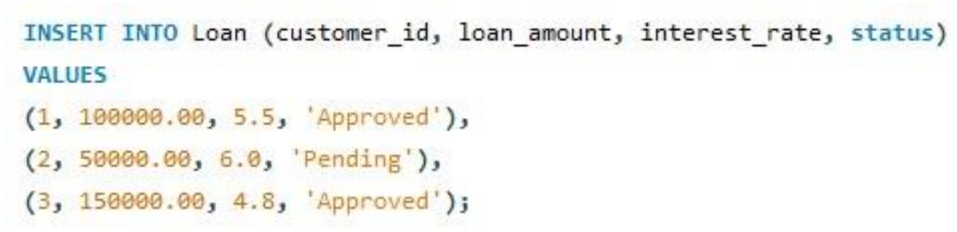
Customer data:

```
INSERT INTO Customer (name, address, phone, email)
VALUES
('Ayesha Khan', 'Karachi', '03001234567', 'ayesha@gmail.com'),
('Ali Raza', 'Lahore', '03214567890', 'ali@gmail.com'),
('Fatima Noor', 'Islamabad', '03331239876', 'fatima@gmail.com');
```

Account data:

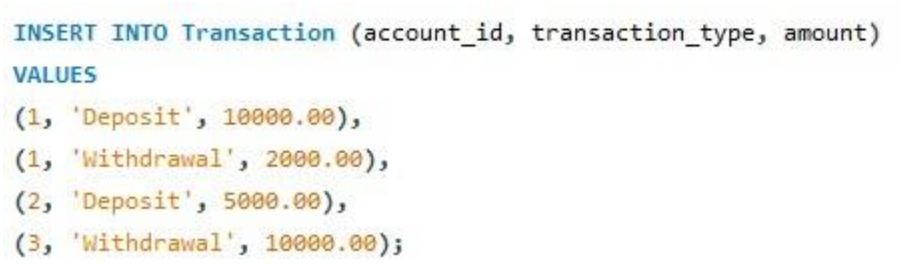
```
INSERT INTO Account (customer_id, account_type, balance)
VALUES
(1, 'Savings', 50000.00),
(2, 'Current', 25000.00),
(3, 'Savings', 80000.00);
```

Loan data:



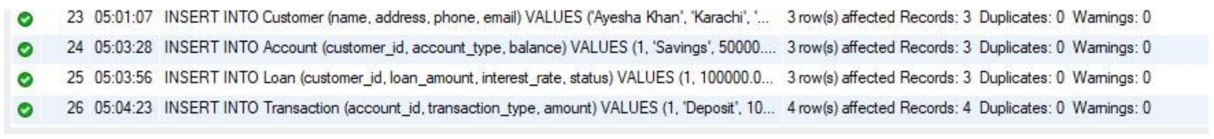
```
INSERT INTO Loan (customer_id, loan_amount, interest_rate, status)
VALUES
(1, 100000.00, 5.5, 'Approved'),
(2, 50000.00, 6.0, 'Pending'),
(3, 150000.00, 4.8, 'Approved');
```

Transaction data:



```
INSERT INTO Transaction (account_id, transaction_type, amount)
VALUES
(1, 'Deposit', 10000.00),
(1, 'Withdrawal', 2000.00),
(2, 'Deposit', 5000.00),
(3, 'Withdrawal', 10000.00);
```

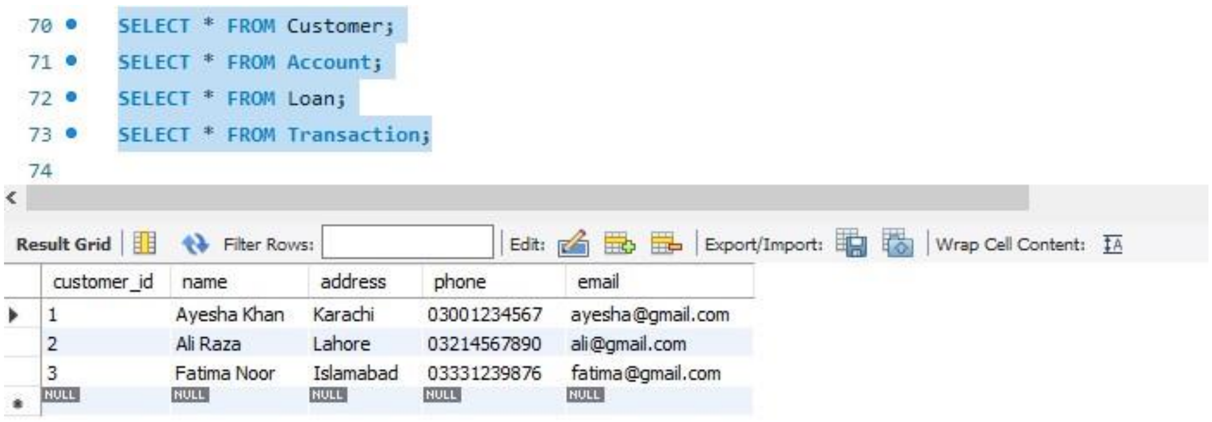
Data has been successfully inserted:



```
23  05:01:07  INSERT INTO Customer (name, address, phone, email) VALUES ('Ayesha Khan', 'Karachi', '...  3 row(s) affected Records: 3  Duplicates: 0  Warnings: 0
24  05:03:28  INSERT INTO Account (customer_id, account_type, balance) VALUES (1, 'Savings', 50000....  3 row(s) affected Records: 3  Duplicates: 0  Warnings: 0
25  05:03:56  INSERT INTO Loan (customer_id, loan_amount, interest_rate, status) VALUES (1, 100000.0...  3 row(s) affected Records: 3  Duplicates: 0  Warnings: 0
26  05:04:23  INSERT INTO Transaction (account_id, transaction_type, amount) VALUES (1, 'Deposit', 10...  4 row(s) affected Records: 4  Duplicates: 0  Warnings: 0
```

Step 07: Check the Data

Customer:

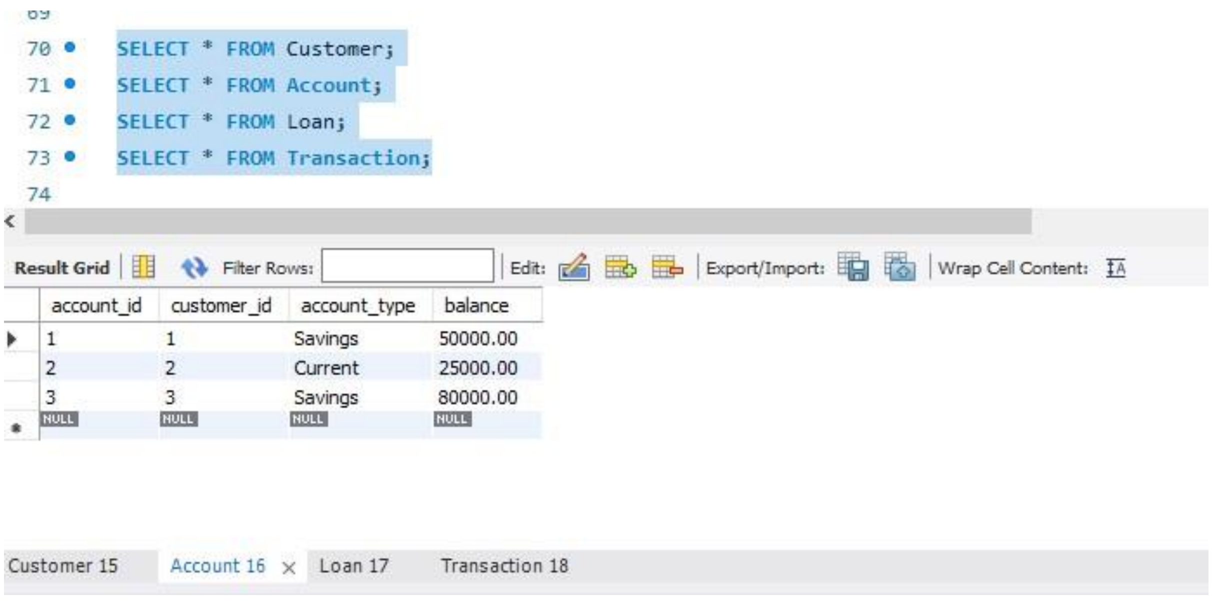


```
SELECT * FROM Customer;
SELECT * FROM Account;
SELECT * FROM Loan;
SELECT * FROM Transaction;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

customer_id	name	address	phone	email
1	Ayesha Khan	Karachi	03001234567	ayesha@gmail.com
2	Ali Raza	Lahore	03214567890	ali@gmail.com
3	Fatima Noor	Islamabad	03331239876	fatima@gmail.com
NULL	NULL	NULL	NULL	NULL

Account:



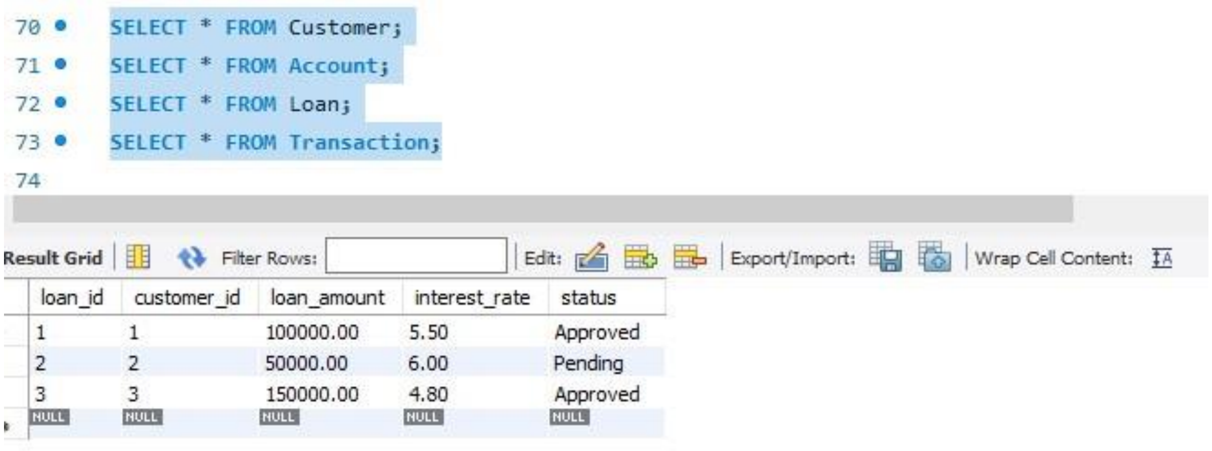
```
SELECT * FROM Customer;
SELECT * FROM Account;
SELECT * FROM Loan;
SELECT * FROM Transaction;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

account_id	customer_id	account_type	balance
1	1	Savings	50000.00
2	2	Current	25000.00
3	3	Savings	80000.00
NULL	NULL	NULL	NULL

Customer 15 | Account 16 | Loan 17 | Transaction 18

Loan:

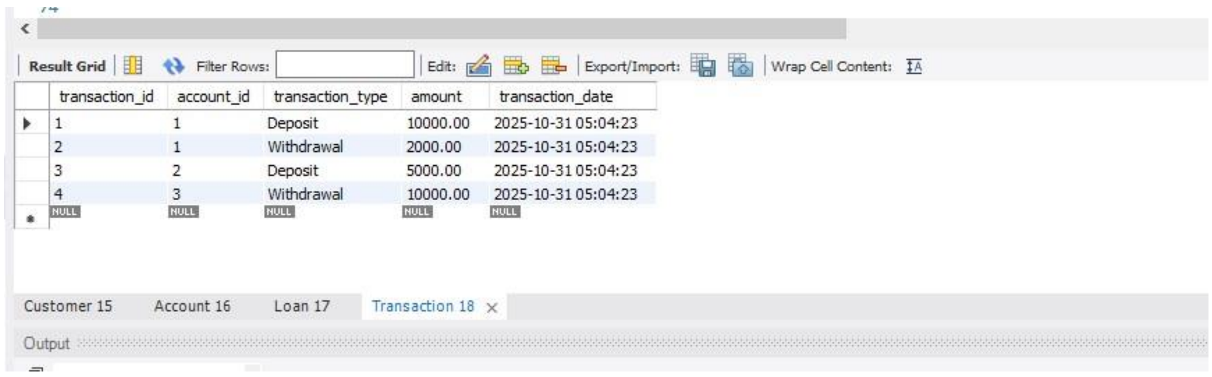


```
SELECT * FROM Customer;
SELECT * FROM Account;
SELECT * FROM Loan;
SELECT * FROM Transaction;
```

loan_id	customer_id	loan_amount	interest_rate	status
1	1	100000.00	5.50	Approved
2	2	50000.00	6.00	Pending
3	3	150000.00	4.80	Approved
NULL	NULL	NULL	NULL	NULL

Customer 15 Account 16 Loan 17 Transaction 18

Transaction:



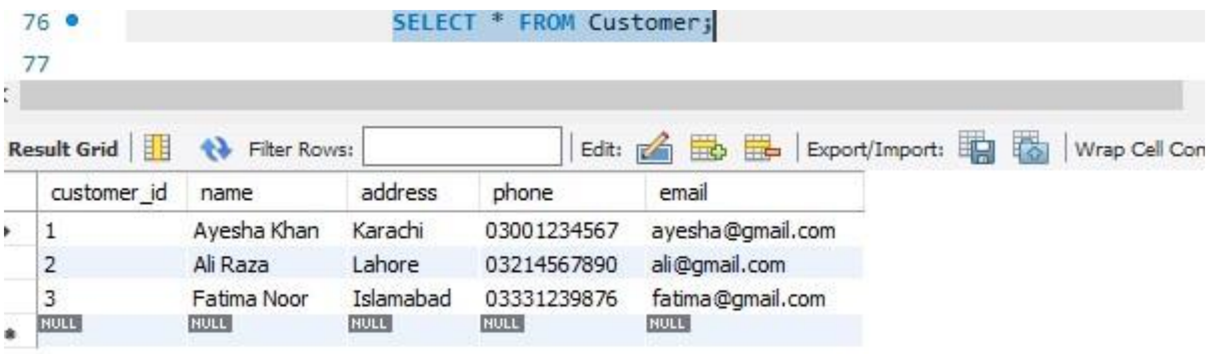
transaction_id	account_id	transaction_type	amount	transaction_date
1	1	Deposit	10000.00	2025-10-31 05:04:23
2	1	Withdrawal	2000.00	2025-10-31 05:04:23
3	2	Deposit	5000.00	2025-10-31 05:04:23
4	3	Withdrawal	10000.00	2025-10-31 05:04:23
NULL	NULL	NULL	NULL	NULL

Customer 15 Account 16 Loan 17 Transaction 18

Output

Step 08: Executing Queries (System Demonstration)

1) Show all customers:

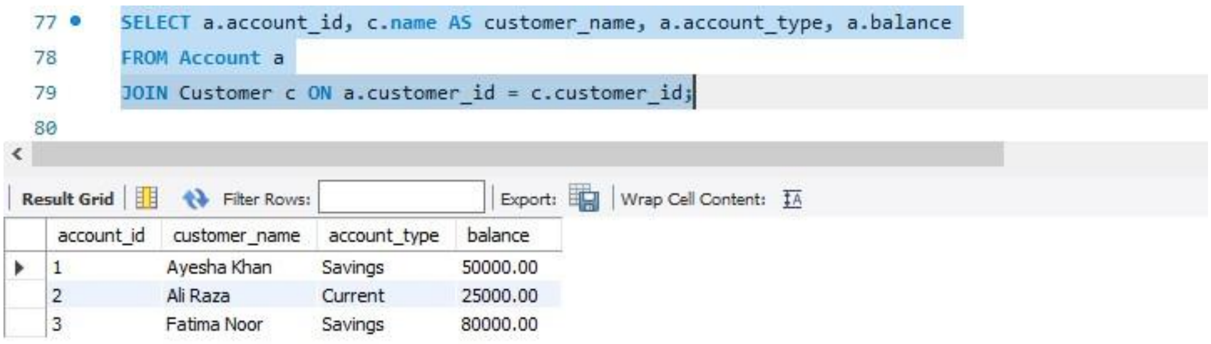


```
SELECT * FROM Customer;
```

customer_id	name	address	phone	email
1	Ayesha Khan	Karachi	03001234567	ayesha@gmail.com
2	Ali Raza	Lahore	03214567890	ali@gmail.com
3	Fatima Noor	Islamabad	03331239876	fatima@gmail.com
NULL	NULL	NULL	NULL	NULL

Description: Displays all customers stored in the database.

2. Show all accounts with customer names



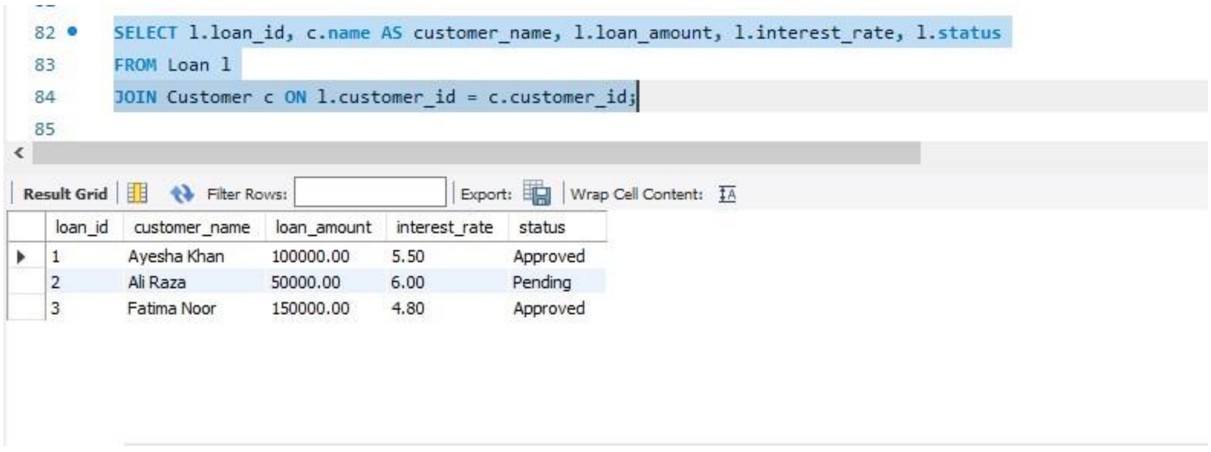
```sql
SELECT a.account_id, c.name AS customer_name, a.account_type, a.balance
FROM Account a
JOIN Customer c ON a.customer_id = c.customer_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

account_id	customer_name	account_type	balance
1	Ayesha Khan	Savings	50000.00
2	Ali Raza	Current	25000.00
3	Fatima Noor	Savings	80000.00

Description: Shows which account belongs to which customer.

3. Show all loans with customer names



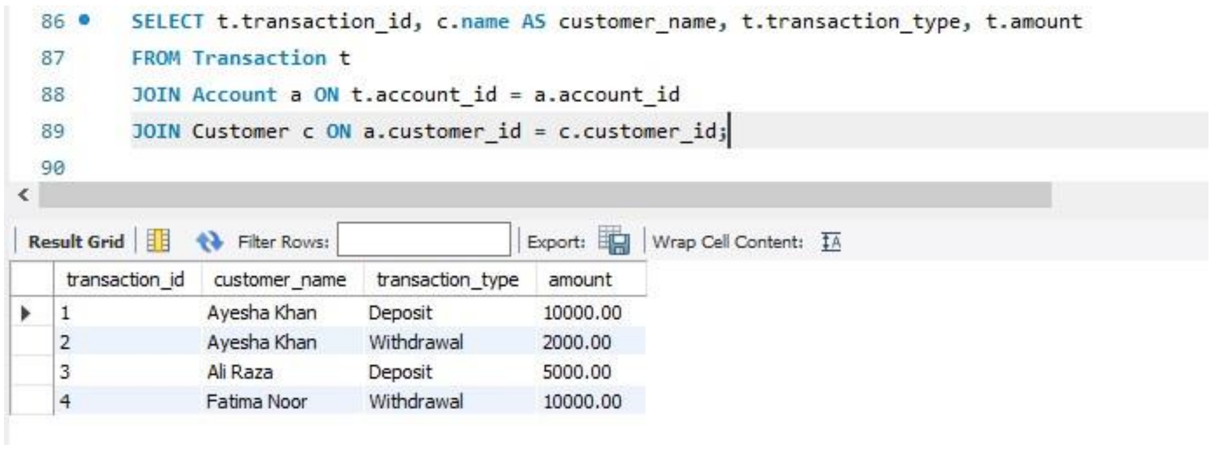
```sql
SELECT l.loan_id, c.name AS customer_name, l.loan_amount, l.interest_rate, l.status
FROM Loan l
JOIN Customer c ON l.customer_id = c.customer_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

loan_id	customer_name	loan_amount	interest_rate	status
1	Ayesha Khan	100000.00	5.50	Approved
2	Ali Raza	50000.00	6.00	Pending
3	Fatima Noor	150000.00	4.80	Approved

Description: Displays each loan along with the name of the customer who took it.

4. Show all transactions with account and customer names

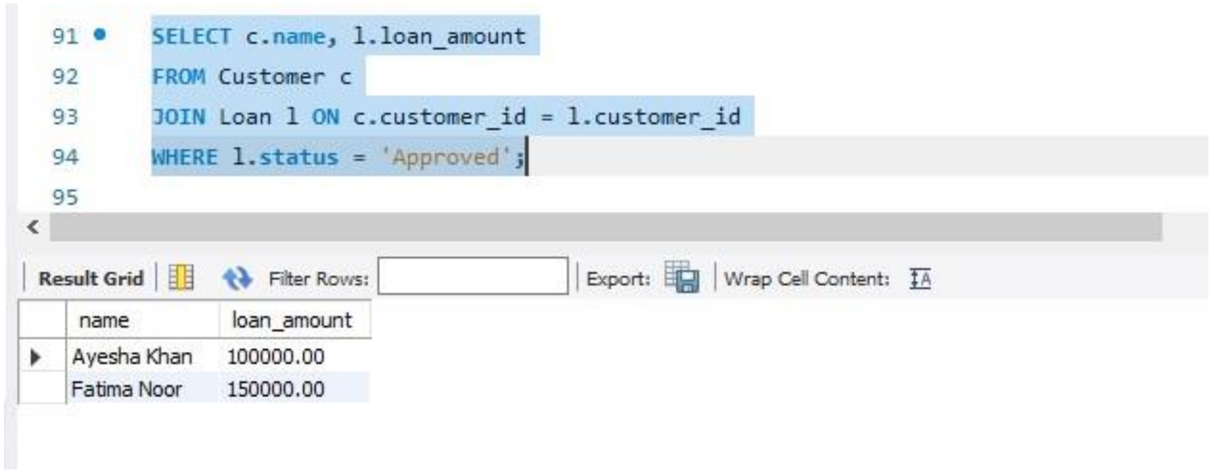


```
SELECT t.transaction_id, c.name AS customer_name, t.transaction_type, t.amount
FROM Transaction t
JOIN Account a ON t.account_id = a.account_id
JOIN Customer c ON a.customer_id = c.customer_id;
```

transaction_id	customer_name	transaction_type	amount
1	Ayesha Khan	Deposit	10000.00
2	Ayesha Khan	Withdrawal	2000.00
3	Ali Raza	Deposit	5000.00
4	Fatima Noor	Withdrawal	10000.00

Description: Combines transaction, account, and customer details.

5. Show customers who have approved loans

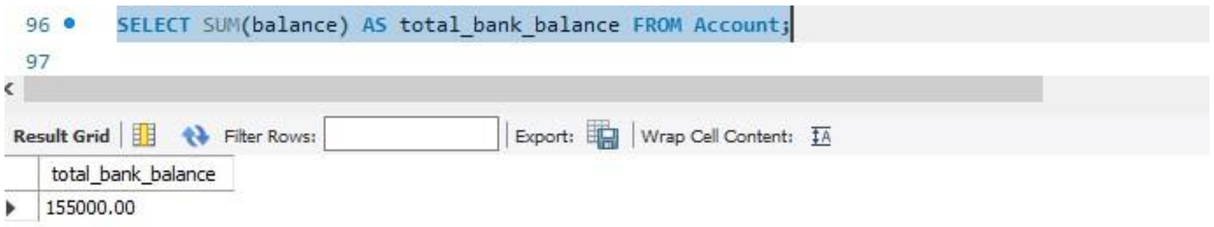


```
SELECT c.name, l.loan_amount
FROM Customer c
JOIN Loan l ON c.customer_id = l.customer_id
WHERE l.status = 'Approved';
```

name	loan_amount
Ayesha Khan	100000.00
Fatima Noor	150000.00

Description: Lists customers whose loan applications are approved.

6. Show total balance in all accounts



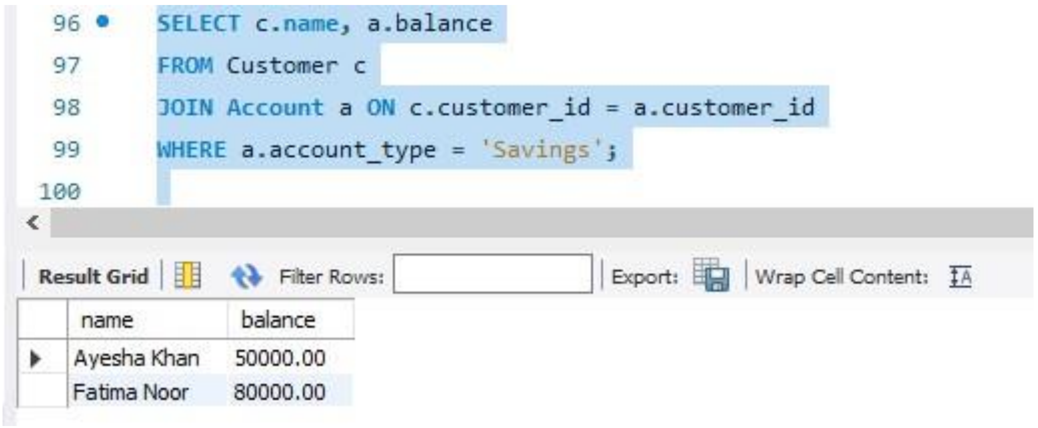
```
SELECT SUM(balance) AS total_bank_balance FROM Account;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

total_bank_balance
155000.00

Description: Shows total money stored in all accounts combined.

7. Show customers having Savings accounts



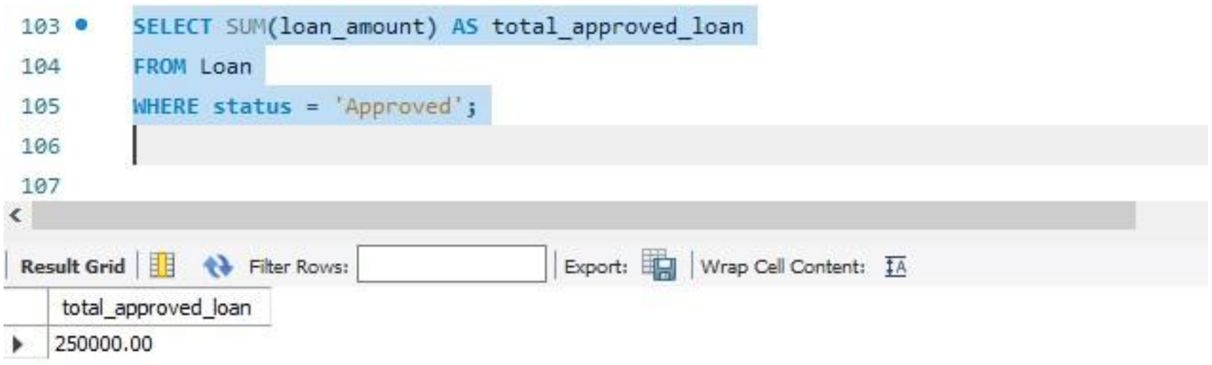
```
SELECT c.name, a.balance
FROM Customer c
JOIN Account a ON c.customer_id = a.customer_id
WHERE a.account_type = 'Savings';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

name	balance
Ayesha Khan	50000.00
Fatima Noor	80000.00

Description: Displays customers who have savings accounts.

8. Show total loan amount approved



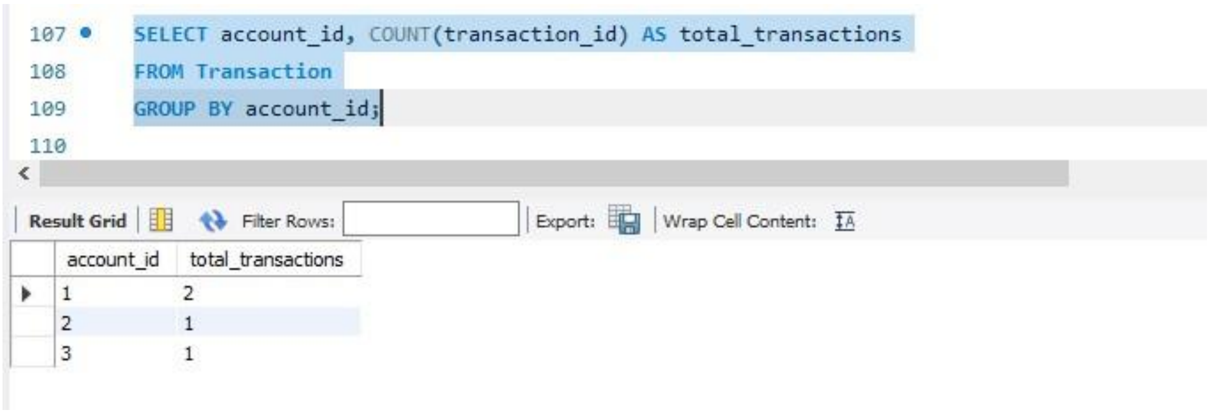
```
SELECT SUM(loan_amount) AS total_approved_loan
FROM Loan
WHERE status = 'Approved';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

total_approved_loan
250000.00

Description: Calculates total approved loan amount from all customers.

9. Count number of transactions per account



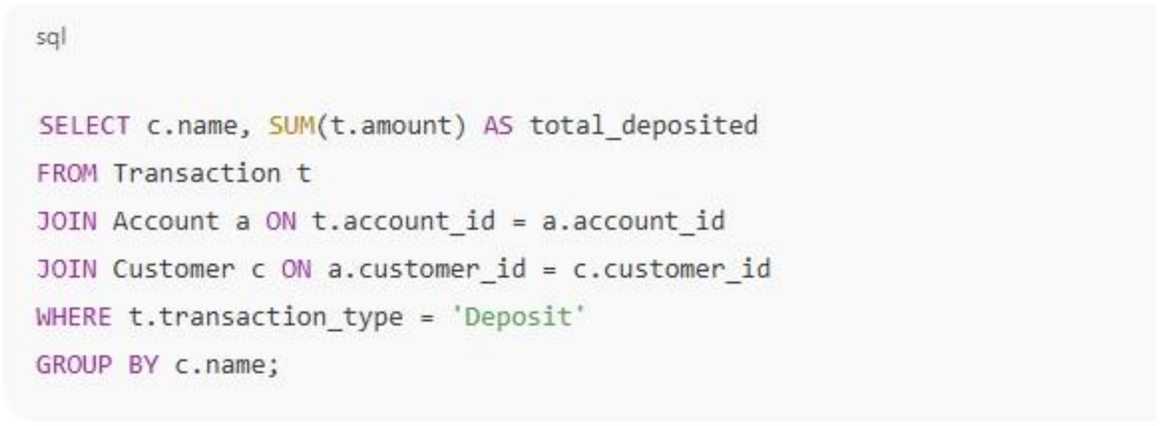
```sql
SELECT account_id, COUNT(transaction_id) AS total_transactions
FROM Transaction
GROUP BY account_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

account_id	total_transactions
1	2
2	1
3	1

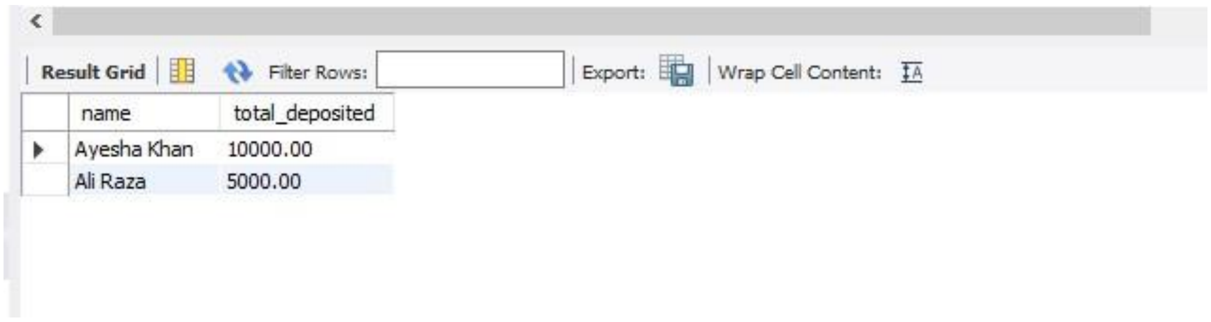
Description: Shows how many transactions happened in each account.

10. Show total money deposited by each customer



sql

```sql
SELECT c.name, SUM(t.amount) AS total_deposited
FROM Transaction t
JOIN Account a ON t.account_id = a.account_id
JOIN Customer c ON a.customer_id = c.customer_id
WHERE t.transaction_type = 'Deposit'
GROUP BY c.name;
```



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

name	total_deposited
Ayesha Khan	10000.00
Ali Raza	5000.00

Description: Calculates total deposited amount by each customer.

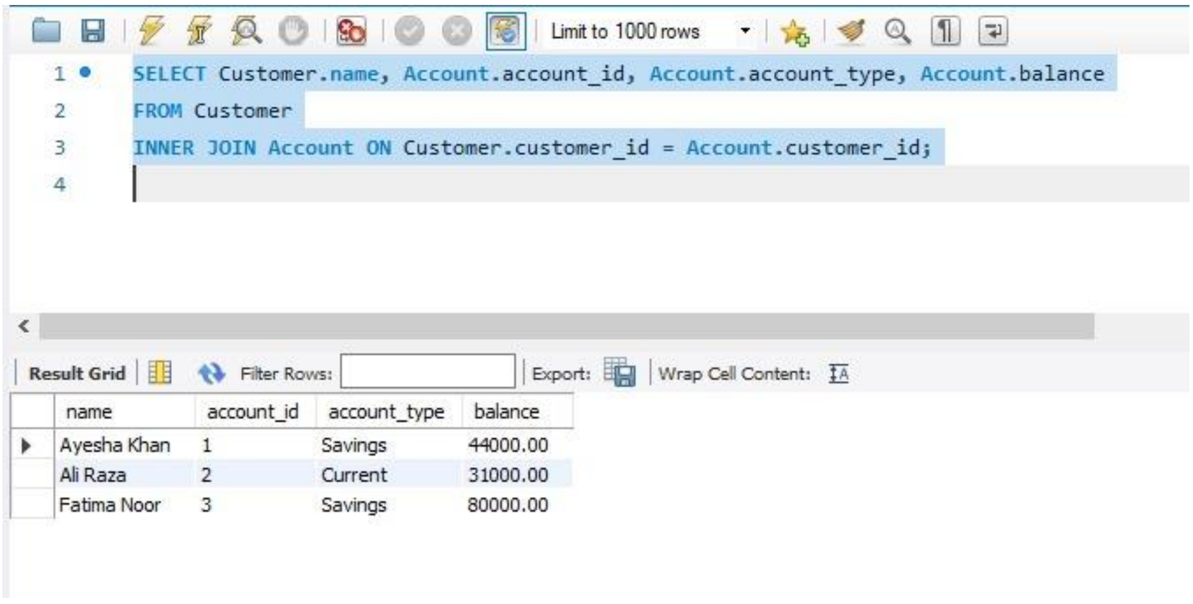
STEP 09: SQL JOINS

WHAT ARE JOINS?
Joins in SQL are used to combine data from two or more tables based on a related column — usually the foreign key.
They help us view connected information, such as which customer owns which account or which transactions belong to which account.

Types of Joins:

INNER JOIN – Returns only matching records from both tables.
LEFT JOIN – Returns all records from the left table, even if there's no match in the right one.
RIGHT JOIN – Returns all records from the right table, even if there's no match in the left one.
FULL JOIN – Combines all records from both tables (MySQL simulates this using UNION).

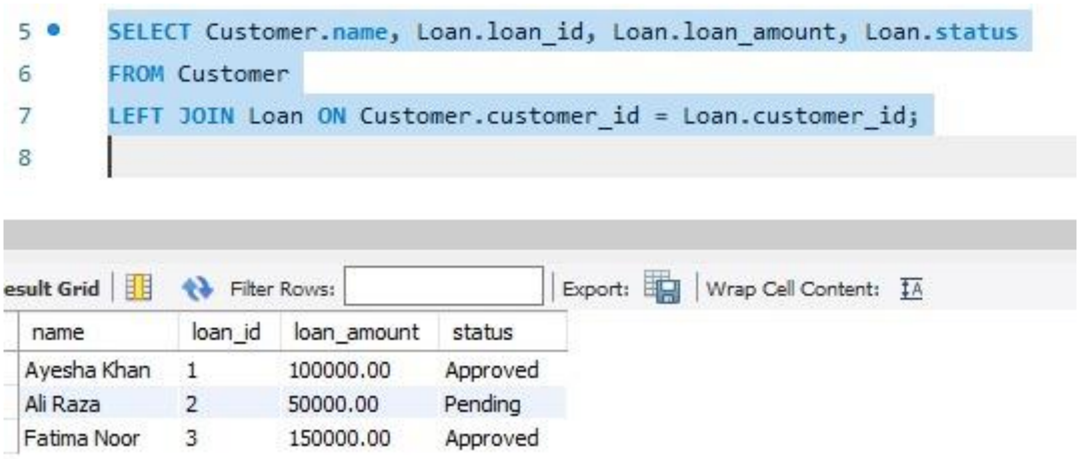
1. **Show All Customers and Their Accounts:**



```
SELECT Customer.name, Account.account_id, Account.account_type, Account.balance
FROM Customer
INNER JOIN Account ON Customer.customer_id = Account.customer_id;
```

name	account_id	account_type	balance
Ayesha Khan	1	Savings	44000.00
Ali Raza	2	Current	31000.00
Fatima Noor	3	Savings	80000.00

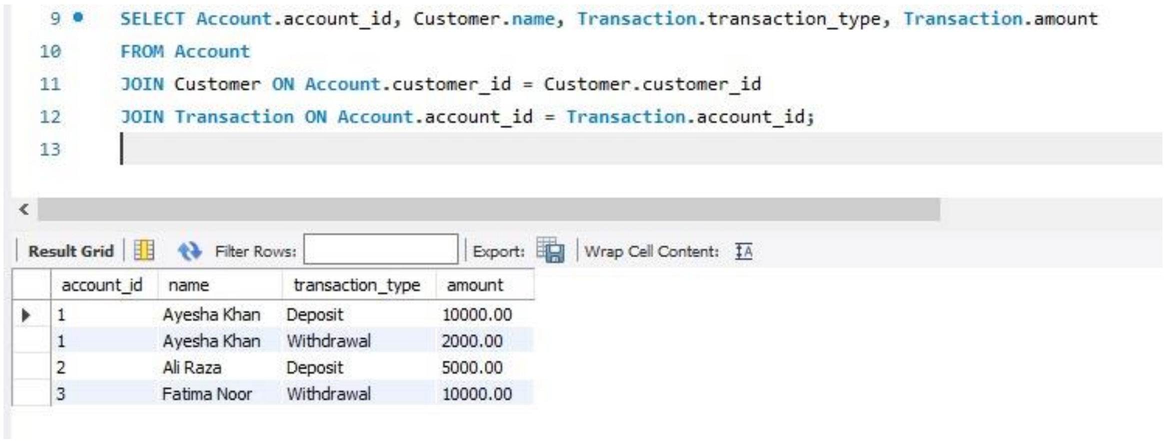
2. Show All Customers and Their Accounts:



```
SELECT Customer.name, Loan.loan_id, Loan.loan_amount, Loan.status
FROM Customer
LEFT JOIN Loan ON Customer.customer_id = Loan.customer_id;
```

name	loan_id	loan_amount	status
Ayesha Khan	1	100000.00	Approved
Ali Raza	2	50000.00	Pending
Fatima Noor	3	150000.00	Approved

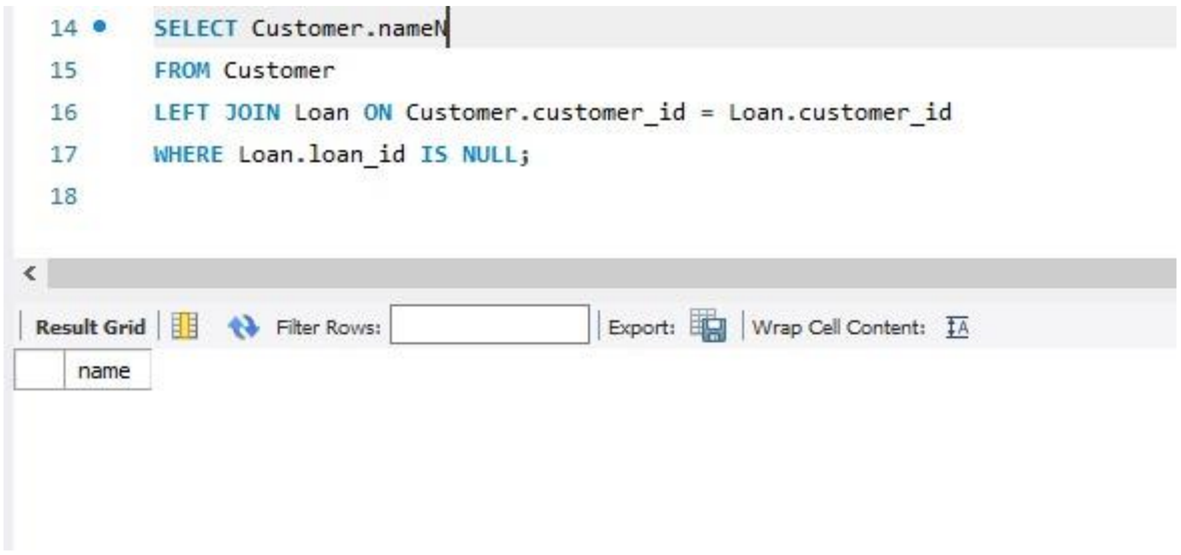
3. Show Account Details with Transactions:



```
SELECT Account.account_id, Customer.name, Transaction.transaction_type, Transaction.amount
FROM Account
JOIN Customer ON Account.customer_id = Customer.customer_id
JOIN Transaction ON Account.account_id = Transaction.account_id;
```

account_id	name	transaction_type	amount
1	Ayesha Khan	Deposit	10000.00
1	Ayesha Khan	Withdrawal	2000.00
2	Ali Raza	Deposit	5000.00
3	Fatima Noor	Withdrawal	10000.00

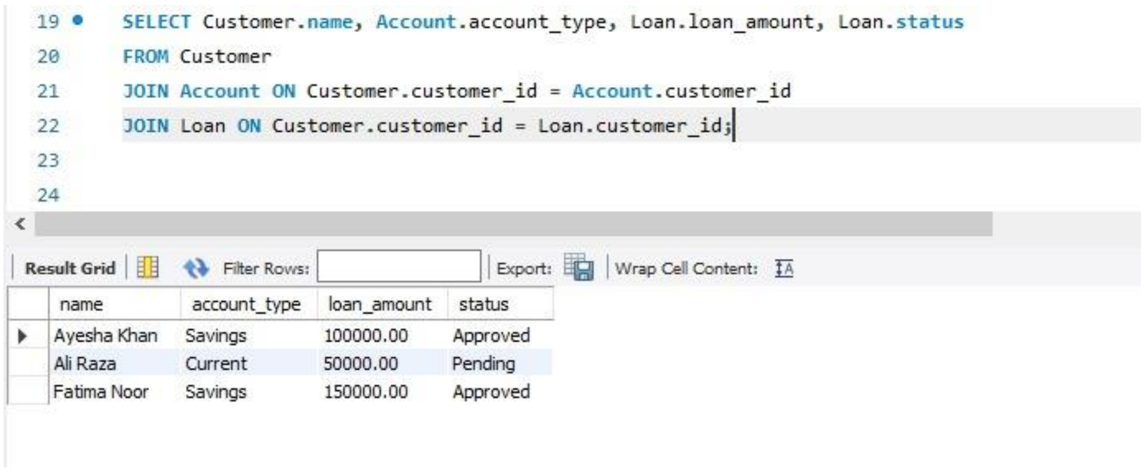
4. Show Customers Who Have No Loans:



```
SELECT Customer.nameN
FROM Customer
LEFT JOIN Loan ON Customer.customer_id = Loan.customer_id
WHERE Loan.loan_id IS NULL;
```

	name

5. Show Loan and Account Info Together:



```sql
SELECT Customer.name, Account.account_type, Loan.loan_amount, Loan.status
FROM Customer
JOIN Account ON Customer.customer_id = Account.customer_id
JOIN Loan ON Customer.customer_id = Loan.customer_id;
```

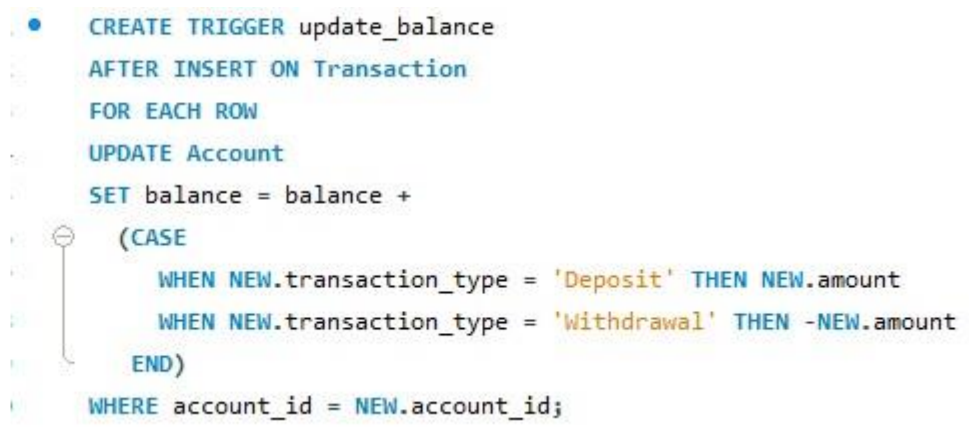
name	account_type	loan_amount	status
Ayesha Khan	Savings	100000.00	Approved
Ali Raza	Current	50000.00	Pending
Fatima Noor	Savings	150000.00	Approved

STEP 10: Database Triggers

A trigger is an automatic action performed by the database when certain events happen — such as inserting, updating, or deleting data in a table.

Triggers help maintain data accuracy and consistency.

This trigger updates account balance automatically when a new transaction occurs.



```sql
CREATE TRIGGER update_balance
AFTER INSERT ON Transaction
FOR EACH ROW
UPDATE Account
SET balance = balance +
  (CASE
     WHEN NEW.transaction_type = 'Deposit' THEN NEW.amount
     WHEN NEW.transaction_type = 'Withdrawal' THEN -NEW.amount
   END)
WHERE account_id = NEW.account_id;
```

This trigger runs automatically after every new transaction.

- If it’s a **Deposit**, the account balance **increases**.
- If it’s a **Withdrawal**, the balance **decreases**.

It helps keep the account balance updated automatically.

PURPOSE OF TRIGGER IN DATABASE:

It runs **automatically** whenever something happens — like inserting, updating, or deleting data.

Example (for banking system):

When a new **transaction** is added:

- The trigger automatically **updates the account balance**, so you don’t need to change it manually each time.

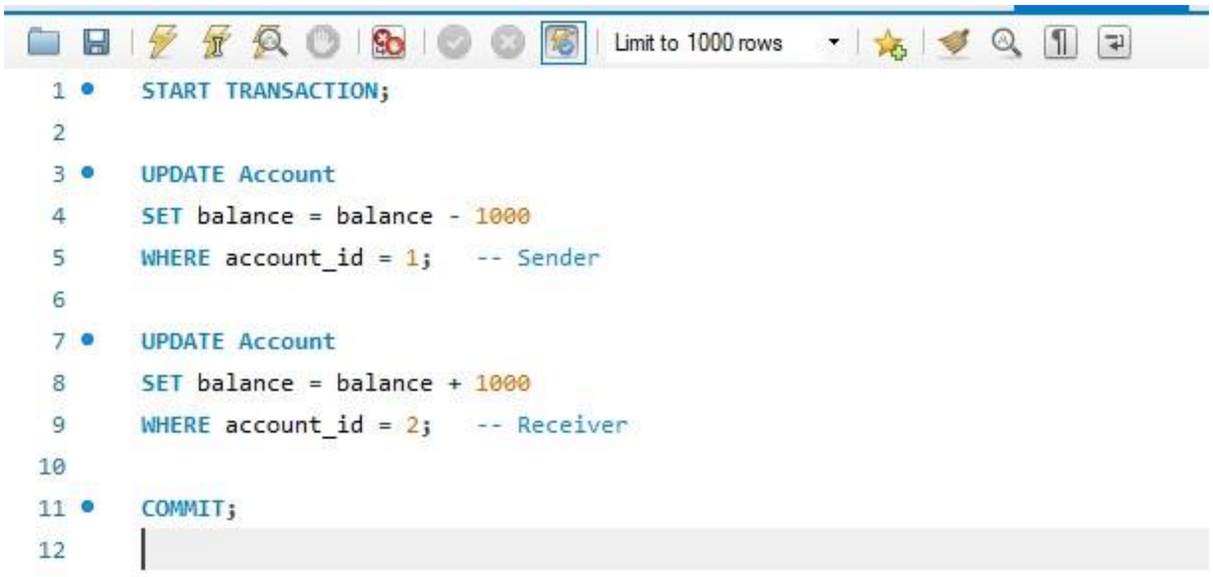
Step 11: ACID Properties:

ACID ensures database transactions are reliable:

- **A – Atomicity:** All steps in a transaction complete or none do.
- **C – Consistency:** Data remains valid before and after the transaction.
- **I – Isolation:** Transactions don’t affect each other.
- **D – Durability:** Once saved, data stays saved even after failures.

When a customer deposits money:

- The system adds money to the account (Atomicity).
- The total balance remains valid (Consistency).
- If another user deposits at the same time, their transaction doesn’t interfere (Isolation).
- After saving, even if the system shuts down, the balance remains updated (Durability).



```sql
START TRANSACTION;

UPDATE Account
SET balance = balance - 1000
WHERE account_id = 1;   -- Sender

UPDATE Account
SET balance = balance + 1000
WHERE account_id = 2;   -- Receiver

COMMIT;
```

This code shows how ACID works:

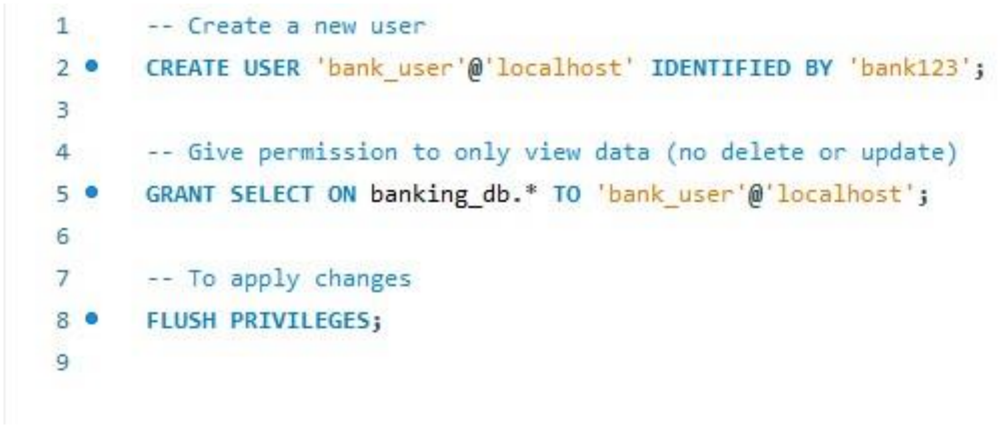
- **Atomicity:** Both updates happen together or not at all.
- **Consistency:** Total money before and after transfer stays same.
- **Isolation:** Other users can’t see changes until commit.
- **Durability:** Once committed, changes stay saved even after shutdown.

Step 12: Security and Recovery:

Security Measures:

- Use user accounts and passwords for access control.
- Restrict database privileges (e.g., only admins can delete data).
- Keep backups regularly to prevent data loss.

Simple Security Example (Using User Privileges)



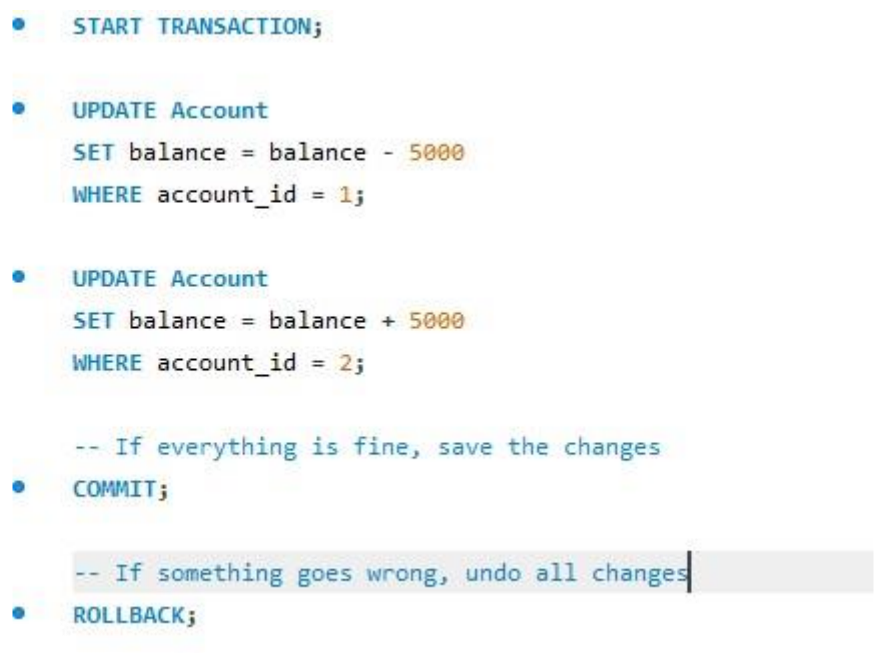
```sql
-- Create a new user
CREATE USER 'bank_user'@'localhost' IDENTIFIED BY 'bank123';

-- Give permission to only view data (no delete or update)
GRANT SELECT ON banking_db.* TO 'bank_user'@'localhost';

-- To apply changes
FLUSH PRIVILEGES;

```

This code adds a new user named bank user who can only view the database records but cannot change or delete them.

This helps protect data by giving limited access to users — keeping important information safe.

Recovery in Database:

Database recovery means getting your data back after a failure — like a crash, power loss, or error.
It ensures the database goes back to its last consistent state using backups or logs.



```
START TRANSACTION;

UPDATE Account
SET balance = balance - 5000
WHERE account_id = 1;

UPDATE Account
SET balance = balance + 5000
WHERE account_id = 2;

-- If everything is fine, save the changes
COMMIT;

-- If something goes wrong, undo all changes
ROLLBACK;
```

This code transfers money between two accounts.
If both updates run successfully, we **COMMIT** (save changes).
If an error happens (like power failure), we **ROLLBACK** — canceling incomplete work.

This ensures **data safety and consistency** — a key part of **database recovery**.

Conclusion:
In this project, we designed a complete banking database system that stores and manages customer, account, loan, and transaction details efficiently. The project demonstrated how database concepts like **relationships, normalization, ACID properties, triggers, security, and recovery** ensure data accuracy and reliability.
Overall, it helped us understand how real-world banking operations can be handled through a well-structured and secure database system.