Course Code: 24MCA130 **Course Instructor: CHAITHRA C S**

Course Name: RDBMS

Week 1: Types of DBMS and SQL Overview

Types of DBMS (Database Management Systems)

A Database Management System (DBMS) is software that manages databases and provides an interface for users to interact with data. The four primary types of DBMS are:

1. **Hierarchical DBMS (HDBMS):**
    - **Structure:** Data is organized in a tree-like structure where each record has a single parent and potentially many children (like a hierarchy). It uses a parent-child relationship.
    - **Example:** IBM's Information Management System (IMS).
    - **Advantages:** Fast access to data and easy to navigate when data is organized hierarchically.
    - **Disadvantages:** Limited flexibility and difficult to manage when relationships between data are complex.
2. **Network DBMS (NDBMS):**
    - **Structure:** Data is represented using a graph structure where nodes (records) are connected through links (relationships). Each node can have multiple parents, allowing for more complex relationships than hierarchical systems.
    - **Example:** Integrated Data Store (IDS).
    - **Advantages:** More flexible than hierarchical DBMS with many-to-many relationships.
    - **Disadvantages:** Complex to design and navigate and can be difficult to manage.
3. **Relational DBMS (RDBMS):**
    - **Structure:** Data is stored in tables (relations), where each table consists of rows and columns. Tables can be related to one another using primary and foreign keys. It uses SQL for querying data.
    - **Example:** MySQL, PostgreSQL, Oracle, Microsoft SQL Server.
    - **Advantages:** Simple to use, flexible, supports complex queries, and is the most widely used type today.
    - **Disadvantages:** Can have performance issues with very large databases or extremely complex relationships.
4. **Object-Oriented DBMS (OODBMS):**

- **Structure:** Data is stored as objects, similar to how it is done in object-oriented programming. Objects encapsulate both data and the operations that can be performed on that data.
- **Example:** db4o, ObjectDB.
- **Advantages:** Best suited for applications requiring complex data representations and relationships, like CAD or multimedia.
- **Disadvantages:** Not as widely adopted, and the learning curve can be steep for those familiar with relational models.

DBMS Languages

A DBMS supports various languages to interact with the database. The main types of DBMS languages are:

1. **Data Definition Language (DDL):**
   - **Purpose:** Defines the structure of the database, including creating, altering, and deleting tables, schemas, and other database objects.
   - **Common Commands:**
     - **CREATE:** Defines a new table or database object.
     - **ALTER:** Modifies an existing database object.
     - **DROP:** Deletes a database object.
     - **TRUNCATE:** Removes all records from a table, retaining the table structure.
2. **Data Manipulation Language (DML):**
   - **Purpose:** Manages and manipulates data in the database (insert, update, delete, retrieve).
   - **Common Commands:**
     - **INSERT:** Adds new data into a table.
     - **UPDATE:** Modifies existing data in a table.
     - **DELETE:** Removes data from a table.
     - **SELECT:** Retrieves data from a table.
3. **Data Control Language (DCL):**
   - **Purpose:** Controls access to the database and manages permissions.
   - **Common Commands:**

- **GRANT:** Gives specific privileges to users or roles.
- **REVOKE:** Removes previously granted privileges.

4. **Transaction Control Language (TCL):**
   - **Purpose:** Manages transactions in a database (group of SQL statements executed together as a unit).
   - **Common Commands:**
     - **COMMIT:** Saves all changes made during the transaction.
     - **ROLLBACK:** Undoes changes made during the transaction.
     - **SAVEPOINT:** Creates a point within a transaction to which you can roll back.
     - **SET TRANSACTION:** Modifies transaction properties.

SQL (Structured Query Language)

SQL (Structured Query Language) is the standard language used to interact with relational databases. SQL enables users to query, update, and manage data, as well as define database structures.

Key SQL Commands:

1. **Data Definition Language (DDL):**
   - **CREATE:**

```
CREATE TABLE Students (
    StudentID INT,
    Name VARCHAR(50),
    Age INT );
```

   - **ALTER:**

```
ALTER TABLE Students ADD Address VARCHAR(100);
```

   - **DROP:**

```
DROP TABLE Students;
```

2. **Data Manipulation Language (DML):**
   - **SELECT:**

```
SELECT Name, Age FROM Students WHERE Age > 18;
```

   - **INSERT:**

```
INSERT INTO Students (StudentID, Name, Age) VALUES (1, 'Alice', 22);
```

- **UPDATE:**

```
UPDATE Students SET Age = 23 WHERE StudentID = 1;
```

- **DELETE:**

```
DELETE FROM Students WHERE Age < 18;
```

3. **Data Control Language (DCL):**
   - **GRANT:**

```
GRANT SELECT, INSERT ON Students TO UserName;
```

   - **REVOKE:**

```
REVOKE SELECT ON Students FROM UserName;
```

4. **Transaction Control Language (TCL):**
   - **COMMIT:**

```
COMMIT;
```

   - **ROLLBACK:**

```
ROLLBACK;
```

Banking Database

Week-2

This lab focuses on creating a database schema for a Banking Database and performing operations using SQL. The database schema includes six tables: BRANCH, ACCOUNT, CUSTOMER, DEPOSITOR, LOAN, and BORROWER.

Objectives

1. Learn to create relational database tables with primary and foreign keys.
2. Insert data into the tables.
3. Perform SQL queries to retrieve data.
4. Understand relationships between tables (Primary Key and Foreign Key).

- The relationships ensure data consistency and integrity.
- Primary Keys uniquely identify records in a table.
- Foreign Keys link tables, maintaining referential integrity.

1. Creating Tables

a. BRANCH Table

```
CREATE TABLE BRANCH (
    branch_name VARCHAR(50) PRIMARY KEY,
    city VARCHAR(50),
    asset VARCHAR(50)
);
```

- **Primary Key:** branch_name

b. ACCOUNT Table

```
CREATE TABLE ACCOUNT (
    account_number VARCHAR(50) PRIMARY KEY,
    branch_name VARCHAR(50),
    balance FLOAT,
```

```
    FOREIGN KEY (branch_name) REFERENCES BRANCH(branch_name)
);
```

- **Primary Key:** account_number
- **Foreign Key:** branch_name references BRANCH(branch_name)

c. CUSTOMER Table

```
CREATE TABLE CUSTOMER (
    customer_name VARCHAR(50) PRIMARY KEY,
    street VARCHAR(100),
    city VARCHAR(50)
);
```

- **Primary Key:** customer_name

d. DEPOSITOR Table

```
CREATE TABLE DEPOSITOR (
    customer_name VARCHAR(50),
    account_number VARCHAR(50),
    PRIMARY KEY (customer_name, account_number),
    FOREIGN KEY (customer_name) REFERENCES CUSTOMER(customer_name),
    FOREIGN KEY (account_number) REFERENCES ACCOUNT(account_number)
);
```

- **Primary Key:** Combination of customer_name and account_number
- **Foreign Keys:**
    - customer_name references CUSTOMER(customer_name)
    - account_number references ACCOUNT(account_number)

e. LOAN Table

```
CREATE TABLE LOAN (
    loan_number VARCHAR(50) PRIMARY KEY,
```

```
    branch_name VARCHAR(50),
    amount INTEGER,
    FOREIGN KEY (branch_name) REFERENCES BRANCH(branch_name)
);
```

- **Primary Key:** loan_number
- **Foreign Key:** branch_name references BRANCH(branch_name)

f. BORROWER Table

```
CREATE TABLE BORROWER (
    customer_name VARCHAR(50),
    loan_number VARCHAR(50),
    PRIMARY KEY (customer_name, loan_number),
    FOREIGN KEY (customer_name) REFERENCES CUSTOMER(customer_name),
    FOREIGN KEY (loan_number) REFERENCES LOAN(loan_number)
);
```

- **Primary Key:** Combination of customer_name and loan_number
- **Foreign Keys:**
  - customer_name references CUSTOMER(customer_name)
  - loan_number references LOAN(loan_number)

---

2. Inserting Data

a. Insert into BRANCH Table

```
INSERT INTO BRANCH (branch_name, city, asset)
VALUES ('Downtown', 'New York', '1000000'),
       ('Uptown', 'Chicago', '800000');
```

b. Insert into ACCOUNT Table

```
INSERT INTO ACCOUNT (account_number, branch_name, balance)
```

```
VALUES ('ACC001', 'Downtown', 5000.00),
       ('ACC002', 'Uptown', 3000.00);
```

c. Insert into CUSTOMER Table

```
INSERT INTO CUSTOMER (customer_name, street, city)
VALUES ('Alice', '123 Main St', 'New York'),
       ('Bob', '456 Oak St', 'Chicago');
```

d. Insert into DEPOSITOR Table

```
INSERT INTO DEPOSITOR (customer_name, account_number)
VALUES ('Alice', 'ACC001'),
       ('Bob', 'ACC002');
```

e. Insert into LOAN Table

```
INSERT INTO LOAN (loan_number, branch_name, amount)
VALUES ('LN001', 'Downtown', 10000),
       ('LN002', 'Uptown', 5000);
```

f. Insert into BORROWER Table

```
INSERT INTO BORROWER (customer_name, loan_number)
VALUES ('Alice', 'LN001'),
       ('Bob', 'LN002');
```

3. Selecting Data

a. Select All Branches

```
SELECT * FROM BRANCH;
```

b. Select All Accounts and Their Balance

```
SELECT account_number, balance FROM ACCOUNT;
```

c. Select All Customers Depositing in a Specific Account

```
SELECT customer_name FROM DEPOSITOR
WHERE account_number = 'ACC001';
```

d. Select All Loans for a Specific Branch

```
SELECT loan_number, amount FROM LOAN
WHERE branch_name = 'Downtown';
```

e. Select All Borrowers and Their Loans

```
SELECT customer_name, loan_number FROM BORROWER;
```

f. Select All Customers and Their Account Balances

```
SELECT c.customer_name, a.balance
FROM CUSTOMER c
JOIN DEPOSITOR d ON c.customer_name = d.customer_name
JOIN ACCOUNT a ON d.account_number = a.account_number;
```

---

4. Primary Key and Foreign Key Relationships

Primary Keys:

- BRANCH(branch_name)
- ACCOUNT(account_number)
- CUSTOMER(customer_name)
- LOAN(loan_number)

Foreign Keys:

- ACCOUNT(branch_name) → BRANCH(branch_name)
- DEPOSITOR(customer_name) → CUSTOMER(customer_name)
- DEPOSITOR(account_number) → ACCOUNT(account_number)
- LOAN(branch_name) → BRANCH(branch_name)
- BORROWER(customer_name) → CUSTOMER(customer_name)
- BORROWER(loan_number) → LOAN(loan_number)

Week-3

Let's continue from the Banking Database and generate the SQL queries for the additional tasks. We will focus on specific queries involving **aggregate functions**, **comparison operators**, and **logical operators**. Below are the SQL queries for the required tasks.

1. List the loan number from LOAN having amount 10000 with a specific branch name.

For this query, we will filter the LOAN table for rows where the amount is 10000 and the branch_name matches a specific branch.

```
SELECT loan_number

FROM LOAN

WHERE amount = 10000 AND branch_name = 'Downtown';
```

2. List the loan number with an amount between 1000 and 10000.

This query uses the BETWEEN operator to filter the LOAN table for loans where the amount is between 1000 and 10000.

```
SELECT loan_number

FROM LOAN

WHERE amount BETWEEN 1000 AND 10000;
```

3. List the customer name (cname) with a substring.

If we want to filter customer names containing a specific substring (e.g., names containing "Ali"), we can use the LIKE operator with wildcard characters (%).

```
SELECT customer_name

FROM CUSTOMER

WHERE customer_name LIKE '%Ali%';
```

4. List the number of tuples in the CUSTOMER table.

To get the count of records (tuples) in the CUSTOMER table, we use the COUNT() aggregate function.

```
SELECT COUNT(*) AS number_of_customers

FROM CUSTOMER;
```

5. List customer name, loan number, and amount with a specific branch name.

This query involves joining the BORROWER, LOAN, and CUSTOMER tables to list the customer name, loan number, and loan amount for a specific branch.

```
SELECT c.customer_name, l.loan_number, l.amount

FROM BORROWER b

JOIN CUSTOMER c ON b.customer_name = c.customer_name

JOIN LOAN l ON b.loan_number = l.loan_number

WHERE l.branch_name = 'Downtown';
```

---

6. Various Aggregate Functions:

a. Get the total sum of loan amounts:

The SUM() aggregate function is used to calculate the total sum of all loan amounts.

```
SELECT SUM(amount) AS total_loan_amount

FROM LOAN;
```

b. Get the average balance of all accounts:

The AVG() function calculates the average balance from the ACCOUNT table.

```
SELECT AVG(balance) AS average_balance

FROM ACCOUNT;
```

c. Get the maximum loan amount:

The MAX() function returns the maximum loan amount.

```
SELECT MAX(amount) AS max_loan_amount

FROM LOAN;
```

d. Get the minimum balance in accounts:

The MIN() function retrieves the minimum account balance.

```
SELECT MIN(balance) AS min_balance

FROM ACCOUNT;
```

e. Get the number of loans (count of loans):

The COUNT() function is used to count the total number of loans.

```
SELECT COUNT(*) AS number_of_loans
FROM LOAN;
```

---

7. Various Comparison Operators:

a. Find customers with a balance greater than 5000:

Here we use the > operator to find accounts with a balance greater than 5000.

```
SELECT c.customer_name, a.balance
FROM CUSTOMER c
JOIN DEPOSITOR d ON c.customer_name = d.customer_name
JOIN ACCOUNT a ON d.account_number = a.account_number
WHERE a.balance > 5000;
```

b. Find customers who borrowed loans greater than 5000:

We use the > comparison operator to find loan amounts greater than 5000.

```
SELECT b.customer_name, l.loan_number, l.amount
FROM BORROWER b
JOIN LOAN l ON b.loan_number = l.loan_number
WHERE l.amount > 5000;
```

c. Find accounts with a balance less than or equal to 1000:

Using the <= operator, this query retrieves accounts with a balance less than or equal to 1000.

```
SELECT a.account_number, a.balance
FROM ACCOUNT a
WHERE a.balance <= 1000;
```

d. Find customers with loans less than 5000 and belonging to a specific branch:

```
SELECT c.customer_name, l.loan_number, l.amount
FROM BORROWER b
```

```
JOIN CUSTOMER c ON b.customer_name = c.customer_name

JOIN LOAN l ON b.loan_number = l.loan_number

WHERE l.amount < 5000 AND l.branch_name = 'Uptown';
```

---

8. Various Logical Operators:

a. Find customers who have a balance greater than 1000 and have a loan of more than 5000:

This query uses AND to combine the two conditions: balance greater than 1000 and loan amount greater than 5000.

```
SELECT c.customer_name, a.balance, l.loan_number, l.amount

FROM CUSTOMER c

JOIN DEPOSITOR d ON c.customer_name = d.customer_name

JOIN ACCOUNT a ON d.account_number = a.account_number

JOIN BORROWER b ON c.customer_name = b.customer_name

JOIN LOAN l ON b.loan_number = l.loan_number

WHERE a.balance > 1000 AND l.amount > 5000;
```

b. Find customers who either have a loan or an account with a balance greater than 5000:

This query uses the OR operator to find customers who meet at least one of the conditions.

```
SELECT c.customer_name

FROM CUSTOMER c

JOIN DEPOSITOR d ON c.customer_name = d.customer_name

JOIN ACCOUNT a ON d.account_number = a.account_number

LEFT JOIN BORROWER b ON c.customer_name = b.customer_name

LEFT JOIN LOAN l ON b.loan_number = l.loan_number

WHERE a.balance > 5000 OR l.amount > 5000;
```

c. Find customers with a loan amount between 1000 and 5000, and from a specific branch:

Here, we combine AND and BETWEEN to find customers who have a loan between 1000 and 5000 and are from a specific branch.

```
SELECT c.customer_name, l.loan_number, l.amount

FROM BORROWER b

JOIN CUSTOMER c ON b.customer_name = c.customer_name

JOIN LOAN l ON b.loan_number = l.loan_number

WHERE l.amount BETWEEN 1000 AND 5000 AND l.branch_name = 'Uptown';  -
- Replace 'Uptown' with the desired branch
```

d. Find customers with names starting with "A" and having an account balance greater than 2000:

This query uses both LIKE and the AND logical operator.

```
SELECT c.customer_name, a.balance

FROM CUSTOMER c

JOIN DEPOSITOR d ON c.customer_name = d.customer_name

JOIN ACCOUNT a ON d.account_number = a.account_number

WHERE c.customer_name LIKE 'A%' AND a.balance > 2000;
```

Summary of SQL Queries:

- **List loan numbers based on conditions**: Using WHERE, BETWEEN, and comparison operators.
- **Substring search**: Using the LIKE operator with %.
- **Count tuples**: Using COUNT() aggregate function.
- **Join tables for specific information**: Using JOIN to combine data from multiple tables.
- **Aggregate functions**: Using SUM(), AVG(), MAX(), MIN(), and COUNT() for calculations.
- **Comparison operators**: Using =, <, >, <=, >=, and BETWEEN to filter data.
- **Logical operators**: Using AND, OR, and NOT to combine multiple conditions.

These queries address the tasks for retrieving data from the Banking Database, including using aggregate functions, comparison operators, and logical operators.

Week-4

Let's design and generate SQL queries for the **Accessories Database**, which includes the PRODUCT, PC, LAPTOP, and PRINTER tables, based on the requirements you provided.

1. Creating the Tables

a. PRODUCT Table

```
CREATE TABLE PRODUCT (
    maker VARCHAR(50),
    model VARCHAR(50),
    type VARCHAR(50),
    PRIMARY KEY (maker, model)
);
```

- Primary Key: The combination of maker and model will be the primary key to uniquely identify each product.

b. PC Table

```
CREATE TABLE PC (
    model VARCHAR(50) PRIMARY KEY,
    speed VARCHAR(50),
    ram VARCHAR(50),
    hdd VARCHAR(50),
    removable_disk INT,
    price INT
);
```

- Primary Key: model is the primary key for the PC table.

c. LAPTOP Table

```
CREATE TABLE LAPTOP (
    model VARCHAR(50) PRIMARY KEY,
```

```
    speed VARCHAR(50),
    ram VARCHAR(50),
    hdd VARCHAR(50),
    screen VARCHAR(50),
    price INT
);
```

- Primary Key: model is the primary key for the LAPTOP table.

d. PRINTER Table

```
CREATE TABLE PRINTER (
    model VARCHAR(50) PRIMARY KEY,
    color BLOB,
    type VARCHAR(50),
    price INT
);
```

- Primary Key: model is the primary key for the PRINTER table.

---

2. Inserting Data

a. Insert into PRODUCT Table

```
INSERT INTO PRODUCT (maker, model, type)
VALUES ('HP', 'Pavilion X360', 'Laptop'),
       ('Dell', 'XPS 15', 'Laptop'),
       ('Canon', 'PIXMA', 'Printer');
```

b. Insert into PC Table

```
INSERT INTO PC (model, speed, ram, hdd, removable_disk, price)
VALUES ('PC001', '3.5 GHz', '8 GB', '1 TB', 1, 30000),
       ('PC002', '2.8 GHz', '4 GB', '500 GB', 0, 25000);
```

c. Insert into LAPTOP Table

```
INSERT INTO LAPTOP (model, speed, ram, hdd, screen, price)
VALUES ('Laptop001', '2.6 GHz', '8 GB', '512 GB', '15.6 inches', 45000),
       ('Laptop002', '3.1 GHz', '16 GB', '1 TB', '14 inches', 70000);
```

d. Insert into PRINTER Table

```
INSERT INTO PRINTER (model, color, type, price)
VALUES ('PIXMA123', 'Black', 'Inkjet', 15000),
       ('PIXMA124', 'Color', 'Inkjet', 20000);
```

---

3. SQL Queries

a. Find the model, speed, RAM, and HDD for all PCs whose price is under 35000.

```
SELECT model, speed, ram, hdd
FROM PC
WHERE price < 35000;
```

- This query filters the PC table to find PCs with a price less than 35000 and retrieves the model, speed, ram, and hdd attributes.

b. Rename the speed column to mhz and hdd column to GB in the PC table.

In SQL, renaming columns depends on the database system (e.g., MySQL, PostgreSQL, SQL Server). For **MySQL**, the syntax is as follows:

```
ALTER TABLE PC
CHANGE COLUMN speed mhz VARCHAR(50),
CHANGE COLUMN hdd GB VARCHAR(50);
```

- This renames speed to mhz and hdd to GB.

c. Find all manufacturers (makers) of laptops.

```
SELECT DISTINCT maker
FROM PRODUCT
```

```
WHERE type = 'Laptop';
```

- This query selects distinct manufacturers (maker) from the PRODUCT table where the type is 'Laptop'.

d. Find all tuples in PRINTER for color.

```
SELECT model, color

FROM PRINTER;
```

- This query retrieves all model and color from the PRINTER table.

---

4. Various Aggregate Functions:

a. Get the total price of all PCs.

```
SELECT SUM(price) AS total_pc_price

FROM PC;
```

- SUM() calculates the total price of all the PCs in the PC table.

b. Get the average price of all laptops.

```
SELECT AVG(price) AS avg_laptop_price

FROM LAPTOP;
```

- AVG() calculates the average price of all laptops in the LAPTOP table.

c. Get the maximum price of a printer.

```
SELECT MAX(price) AS max_printer_price

FROM PRINTER;
```

- MAX() retrieves the highest price of any printer.

d. Get the minimum price of a PC.

```
SELECT MIN(price) AS min_pc_price

FROM PC;
```

- MIN() retrieves the lowest price of any PC.

e. Get the count of laptops.

```
SELECT COUNT(*) AS num_laptops

FROM LAPTOP;
```

- COUNT() counts the total number of laptops in the LAPTOP table.

---

5. Various Comparison Operators:

a. Find all PCs with a price greater than 30000.

```
SELECT model, price
FROM PC
WHERE price > 30000;
```

- This query filters PCs with a price greater than 30000.

b. Find all laptops with a screen size smaller than 15 inches.

```
SELECT model, screen
FROM LAPTOP
WHERE screen < '15 inches';
```

- This query filters laptops where the screen size is smaller than '15 inches'.

c. Find all printers with a price less than or equal to 20000.

```
SELECT model, price
FROM PRINTER
WHERE price <= 20000;
```

- This query filters printers with a price less than or equal to 20000.

d. Find all products (laptops) from a specific maker, e.g., 'HP'.

```
SELECT model
FROM PRODUCT
WHERE maker = 'HP' AND type = 'Laptop';
```

- This query finds all laptops from the manufacturer 'HP'.

---

6. Various Logical Operators:

a. Find all PCs with a price under 30000 or a removable disk of 1.

```
SELECT model, price, removable_disk
```

```
FROM PC

WHERE price < 30000 OR removable_disk = 1;
```

- This query uses the OR logical operator to filter PCs that have a price under 30000 or a removable disk (removable_disk = 1).

b. Find all laptops with a price greater than 50000 and a screen size larger than 15 inches.

```
SELECT model, price, screen

FROM LAPTOP

WHERE price > 50000 AND screen > '15 inches';
```

- This query uses the AND logical operator to find laptops that are priced above 50000 and have a screen larger than 15 inches.

c. Find all printers where the price is either 15000 or the color is 'Black'.

```
SELECT model, color, price

FROM PRINTER

WHERE price = 15000 OR color = 'Black';
```

- This query uses the OR operator to filter printers that either cost 15000 or have a 'Black' color.

d. Find all laptops where the screen size is not '15.6 inches'.

```
SELECT model, screen

FROM LAPTOP

WHERE screen != '15.6 inches';
```

- This query uses the != operator to find laptops where the screen size is not '15.6 inches'.

Summary of SQL Queries:

- **Find models, speed, RAM, HDD for PCs under a certain price**: Using WHERE and comparison operators.
- **Rename columns in a table**: Using ALTER TABLE with CHANGE COLUMN.
- **Find all manufacturers of laptops**: Using DISTINCT with WHERE.
- **Retrieve specific columns**: Using SELECT queries with relevant filters.
- **Aggregate functions**: SUM(), AVG(), MAX(), MIN(), COUNT().

- **Comparison operators**: =, <, >, <=, >=, !=, BETWEEN.
- **Logical operators**: AND, OR, NOT to combine conditions.

Week -5

Creating Tables

```
1 CREATE TABLE CUSTOMER (
    Cust_id VARCHAR(50) PRIMARY KEY,
    Cust_name VARCHAR(100),
    City VARCHAR(50)
);

2 CREATE TABLE ORDER_TABLE (
    Order_num VARCHAR(50) PRIMARY KEY,
    Order_date DATE,
    Cust_id VARCHAR(50),
    Order_amount INT,
    FOREIGN KEY (Cust_id) REFERENCES CUSTOMER(Cust_id)
);

3 CREATE TABLE ITEM (
    Item_id VARCHAR(50) PRIMARY KEY,
    Unit_price INT
);

4 CREATE TABLE ORDERITEM (
    Order_num VARCHAR(50),
    Item_id VARCHAR(50),
    Quantity INT,
```

```
    PRIMARY KEY (Order_num, Item_id),
    FOREIGN KEY (Order_num) REFERENCES ORDER_TABLE(Order_num),
    FOREIGN KEY (Item_id) REFERENCES ITEM(Item_id)
);

5 CREATE TABLE WAREHOUSE (
    Warehouse_id VARCHAR(50) PRIMARY KEY,
    City VARCHAR(50)
);

6 CREATE TABLE SHIPMENT (
    Order_num VARCHAR(50),
    Warehouse_id VARCHAR(50),
    Shipdate DATE,
    PRIMARY KEY (Order_num, Warehouse_id),
    FOREIGN KEY (Order_num) REFERENCES ORDER_TABLE(Order_num),
    FOREIGN KEY (Warehouse_id) REFERENCES WAREHOUSE(Warehouse_id)
);
```

Inserting Data

```
1 INSERT INTO CUSTOMER (Cust_id, Cust_name, City) VALUES
('C001', 'Alice', 'New York'),
('C002', 'Bob', 'Los Angeles');

2 INSERT INTO ORDER_TABLE (Order_num, Order_date, Cust_id,
Order_amount) VALUES
('O001', '2024-01-10', 'C001', 500),
```

```
3 ('O002', '2024-01-15', 'C002', 700);


INSERT INTO ITEM (Item_id, Unit_price) VALUES
('I001', 100),
('I002', 200);


4 INSERT INTO ORDERITEM (Order_num, Item_id, Quantity) VALUES
('O001', 'I001', 2),
('O002', 'I002', 3);


5 INSERT INTO WAREHOUSE (Warehouse_id, City) VALUES
('W001', 'New York'),
('W002', 'Los Angeles');


6 INSERT INTO SHIPMENT (Order_num, Warehouse_id, Shipdate) VALUES
('O001', 'W001', '2024-01-12'),
('O002', 'W002', '2024-01-17');
```

Queries

1. List customer name, number of orders, and average order amount.

```
SELECT C.Cust_name, COUNT(O.Order_num) AS Num_Orders,
AVG(O.Order_amount) AS Avg_Order_Amount
FROM CUSTOMER C
JOIN ORDER_TABLE O ON C.Cust_id = O.Cust_id
GROUP BY C.Cust_name;
```

2. List order numbers for orders shipped from all warehouses in a specific city.

```
SELECT DISTINCT S.Order_num
FROM SHIPMENT S
JOIN WAREHOUSE W ON S.Warehouse_id = W.Warehouse_id
WHERE W.City = 'New York';
```

3. Use aggregate functions

```
SELECT
    COUNT(*) AS Total_Orders,
    SUM(Order_amount) AS Total_Revenue,
    MAX(Order_amount) AS Max_Order_Amount,
    MIN(Order_amount) AS Min_Order_Amount
FROM ORDER_TABLE;
```

4. Use comparison operators (Find orders with amount greater than 500)

```
SELECT * FROM ORDER_TABLE WHERE Order_amount > 500;
```

5. Use logical operators (Find customers from New York with orders above 600)

```
SELECT C.Cust_name, O.Order_amount
FROM CUSTOMER C
JOIN ORDER_TABLE O ON C.Cust_id = O.Cust_id
WHERE C.City = 'New York' AND O.Order_amount > 600;
```

SQL Queries for Order Processing Database

1. Additional Queries Using Aggregate Functions

a) Find the total order amount for each customer

Query:

```
SELECT cust_id, SUM(order_amount) AS total_spent

FROM ORDER

GROUP BY cust_id;
```

Explanation: This query calculates the total amount spent by each customer.

b) Find the average order amount for all orders placed in the last 6 months

Query:

```
SELECT AVG(order_amount) AS avg_order_amount

FROM ORDER

WHERE order_date >= DATE_SUB(CURDATE(), INTERVAL 6 MONTH);
```

Explanation: Retrieves the average order amount for orders placed in the last 6 months.

c) Find the highest and lowest unit price from the ITEM table

Query:

```
SELECT MAX(unit_price) AS highest_price, MIN(unit_price) AS
lowest_price

FROM ITEM;
```

Explanation: Retrieves the highest and lowest unit prices from the ITEM table.

d) Count the total number of shipments from each warehouse

Query:

```
SELECT warehouse_id, COUNT(order_num) AS total_shipments

FROM SHIPMENT

GROUP BY warehouse_id;
```

Explanation: Counts the number of shipments made from each warehouse.

e) Find the total quantity of items sold for each item

Query:

```
SELECT item_id, SUM(quantity) AS total_quantity_sold

FROM ORDERITEM
```

```
GROUP BY item_id;
```

Explanation: Calculates the total quantity sold for each item.

2. Additional Queries Using Comparison Operators

a) Find all orders where the order amount is greater than 10,000

Query:

```
SELECT order_num, cust_id, order_amount
FROM ORDER
WHERE order_amount > 10000;
```

Explanation: Retrieves all orders where the order amount is greater than 10,000.

b) Find all customers from a specific city, e.g., 'New York'

Query:

```
SELECT cust_id, cust_name
FROM CUSTOMER
WHERE city = 'New York';
```

Explanation: Retrieves all customers from 'New York'.

c) Find items with unit price between 500 and 2000

Query:

```
SELECT item_id, unit_price
FROM ITEM
WHERE unit_price BETWEEN 500 AND 2000;
```

Explanation: Retrieves items priced between 500 and 2000.

d) Find all orders placed on or after January 1, 2024

Query:

```
SELECT order_num, order_date, cust_id
FROM ORDER
```

```
WHERE order_date >= '2024-01-01';
```

Explanation: Retrieves orders placed on or after January 1, 2024.

e) Find customers who have placed more than 5 orders

Query:

```
SELECT cust_id, COUNT(order_num) AS total_orders
FROM ORDER
GROUP BY cust_id
HAVING COUNT(order_num) > 5;
```

Explanation: Filters customers who have placed more than 5 orders.

---

3. Additional Queries Using Logical Operators

a) Find all customers who are either from 'New York' or 'Los Angeles'

Query:

```
SELECT cust_id, cust_name, city
FROM CUSTOMER
WHERE city = 'New York' OR city = 'Los Angeles';
```

Explanation: Retrieves customers from either 'New York' or 'Los Angeles'.

b) Find all orders with an amount greater than 5000 and placed after '2023-06-01'

Query:

```
SELECT order_num, order_amount, order_date
FROM ORDER
WHERE order_amount > 5000 AND order_date > '2023-06-01';
```

Explanation: Retrieves orders that satisfy both conditions.

c) Find all shipments made from warehouses in 'Chicago' but not from 'New York'

Query:

```
SELECT S.order_num, W.city
```

```
FROM SHIPMENT S
JOIN WAREHOUSE W ON S.warehouse_id = W.warehouse_id
WHERE W.city = 'Chicago' AND W.city != 'New York';
```

Explanation: Retrieves shipments made from 'Chicago' while excluding 'New York'.

d) Find orders that contain a specific item, e.g., item_id = 'I001'

Query:

```
SELECT order_num
FROM ORDERITEM
WHERE item_id = 'I001';
```

Explanation: Retrieves all orders containing the specified item.

e) Find customers who have not placed any orders

Query:

```
SELECT C.cust_id, C.cust_name
FROM CUSTOMER C
LEFT JOIN ORDER O ON C.cust_id = O.cust_id
WHERE O.order_num IS NULL;
```

Explanation: Identifies customers who have not placed any orders.

Replace certain value based on your inserted tuple values wherever it is applicable in queries.