SQL Coding Challenge

E-Commerce

Instructions : Coding Challenge submissions should be done through the partcipants' Github repository, and the link should be shared with trainers and Hexavarsity.

This is an **SQL coding challenge** where we solve various database-related tasks for an **e-commerce system**. Each task includes a problem statement, SQL query, explanation, and expected output. The goal is to efficiently manage and retrieve data related to products, customers, orders, and transactions.

---

1. Create Database - ecommerce

To begin, I'm creating a database named ecommerce to store all necessary e-commerce-related data, such as customer details, product information, orders, and payments.

CREATE DATABASE ecommerce;

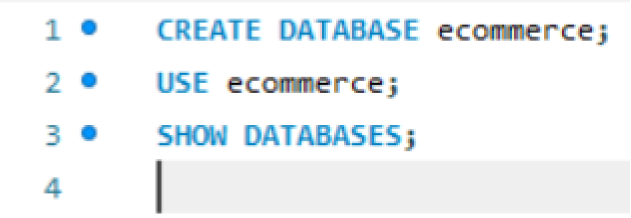
USE ecommerce;

Explanation:

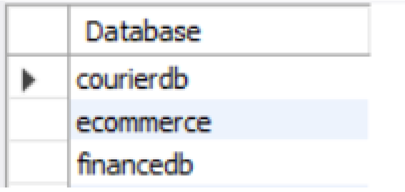
- First, I use CREATE DATABASE ecommerce; to create a new database named ecommerce.
- Then, I use USE ecommerce; to select this database so I can start working on tables.

Output:

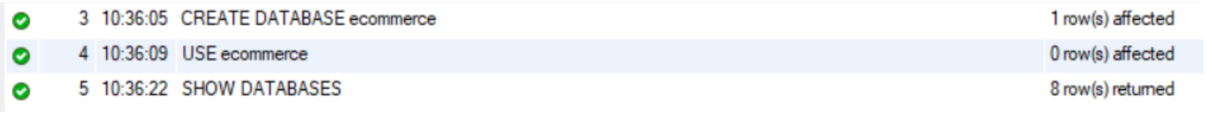
- The ecommerce database is successfully created.
- To confirm, I run: **SHOW DATABASES;**



```
CREATE DATABASE ecommerce;
USE ecommerce;
SHOW DATABASES;
```



	Database
▶	courierdb
	ecommerce
	financedb



```
✓  3  10:36:05  CREATE DATABASE ecommerce    1 row(s) affected
✓  4  10:36:09  USE ecommerce                0 row(s) affected
✓  5  10:36:22  SHOW DATABASES               8 row(s) returned
```

---

2. Creating Tables

1. Creating the customers Table

This table stores customer details like ID, name, email, and password.

SQL Query:

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL
);
```

Explanation:

- customer_id is the Primary Key and auto-increments.
- email is unique to prevent duplicate accounts.
- password is stored as a string (hashed for security in real applications).

Output:

- The customers table is successfully created.

2. Creating the products Table

This table stores product information, including name, price, description, and stock quantity.

SQL Query:

```
CREATE TABLE products (
    product_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    description TEXT,
    stockQuantity INT NOT NULL
);
```

Explanation:

- product_id is the Primary Key and auto-increments.
- price is stored as DECIMAL(10,2) to handle monetary values.
- description allows for a text-based product description.

- stockQuantity keeps track of available stock.

Output:

- The products table is successfully created.

3. Creating the cart Table

This table keeps track of items added to a customer's shopping cart.

SQL Query:

```
CREATE TABLE cart (
  cart_id INT PRIMARY KEY AUTO_INCREMENT,
  customer_id INT,
  product_id INT,
  quantity INT NOT NULL,
  FOREIGN KEY (customer_id) REFERENCES customers(customer_id) ON DELETE CASCADE,
  FOREIGN KEY (product_id) REFERENCES products(product_id) ON DELETE CASCADE
);
```

Explanation:

- cart_id is the Primary Key.
- customer_id and product_id are Foreign Keys referencing customers and products.
- ON DELETE CASCADE ensures that if a customer or product is deleted, related cart entries are also removed.

Output:

- The cart table is successfully created.

4. Creating the orders Table

This table stores customer orders, including order date, total price, and shipping address.

SQL Query:

```
CREATE TABLE orders (
  order_id INT PRIMARY KEY AUTO_INCREMENT,
  customer_id INT,
  order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  total_price DECIMAL(10,2) NOT NULL,
  shipping_address VARCHAR(255) NOT NULL,
  FOREIGN KEY (customer_id) REFERENCES customers(customer_id) ON DELETE CASCADE
);
```

Explanation:

- order_id is the Primary Key and auto-increments.
- customer_id is a Foreign Key linking to customers.
- order_date defaults to the current timestamp.
- total_price is stored as DECIMAL(10,2) to track total order cost.
- shipping_address stores the delivery location.

Output:

- The orders table is successfully created.

5. Creating the order_items Table

This table stores individual products associated with each order.

SQL Query:

```
CREATE TABLE order_items (
   order_item_id INT PRIMARY KEY AUTO_INCREMENT,
   order_id INT,
   product_id INT,
   quantity INT NOT NULL,
   FOREIGN KEY (order_id) REFERENCES orders(order_id) ON DELETE CASCADE,
   FOREIGN KEY (product_id) REFERENCES products(product_id) ON DELETE CASCADE
);
```

Explanation:

- order_item_id is the **Primary Key**.
- order_id and product_id are **Foreign Keys**, linking to orders and products.
- quantity stores the number of units purchased.
- ON DELETE CASCADE ensures that if an order or product is deleted, related order_items are also removed.

Output:

- The order_items table is successfully created.

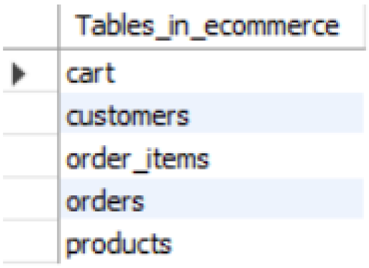
6. Checking All Created Tables

Now that I have created all the necessary tables, I want to verify that they exist in the ecommerce database.

```
SHOW TABLES;
```

Explanation:

- The SHOW TABLES; command displays a list of all tables in the currently selected database (ecommerce).
- This confirms that the tables were successfully created.



Tables_in_ecommerce
cart
customers
order_items
orders
products

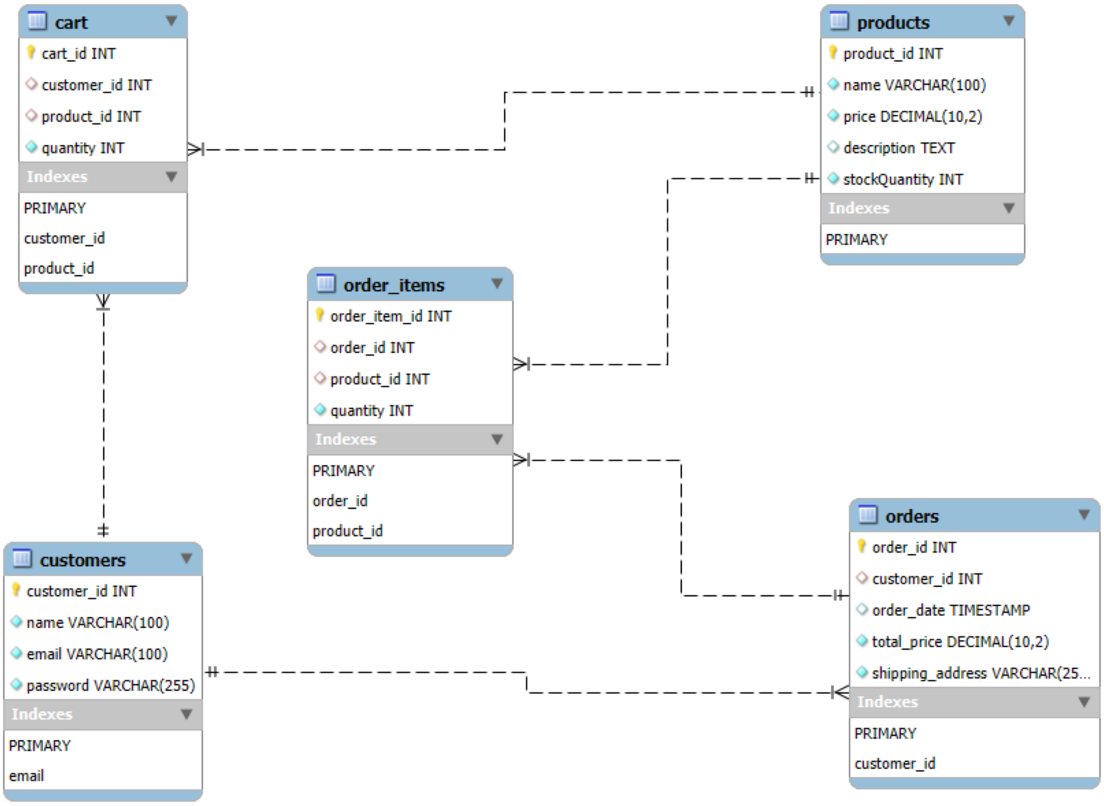
---

3. ER Diagram

Entities and Relationships:

1. **Customers** (customer_id as Primary Key)
   - One customer can place multiple orders (**one-to-many** with orders).
   - One customer can have multiple products in their cart (**one-to-many** with cart).
2. **Products** (product_id as Primary Key)
   - A product can be in multiple carts (**one-to-many** with cart).
   - A product can be part of multiple orders (**one-to-many** with order_items).
3. **Cart** (cart_id as Primary Key)
   - Each cart entry links a customer to a product (**many-to-one** with customers and products).
4. **Orders** (order_id as Primary Key)
   - Each order belongs to a single customer (**many-to-one** with customers).
   - An order consists of multiple products (**one-to-many** with order_items).
5. **Order Items** (order_item_id as Primary Key)
   - Each order item links an order to a product (**many-to-one** with orders and products).

ER Diagram



- **Primary Keys:** customer_id, product_id, cart_id, order_id, order_item_id.
- **Foreign Keys:**
  - customer_id in cart and orders references customers.
  - product_id in cart and order_items references products.
  - order_id in order_items references orders.

---

4. Inserting Values on Tables

1. Inserting Data into the products Table

Now, I'm inserting product data into the products table.

SQL Query:

```
INSERT INTO products (name, description, price, stockQuantity) VALUES
('Laptop', 'High-performance laptop', 800.00, 10),
('Smartphone', 'Latest smartphone', 600.00, 15),
('Tablet', 'Portable tablet', 300.00, 20),
('Headphones', 'Noise-canceling', 150.00, 30),
('TV', '4K Smart TV', 900.00, 5),
('Coffee Maker', 'Automatic coffee maker', 50.00, 25),
('Refrigerator', 'Energy-efficient', 700.00, 10),
('Microwave Oven', 'Countertop microwave', 80.00, 15),
('Blender', 'High-speed blender', 70.00, 20),
('Vacuum Cleaner', 'Bagless vacuum cleaner', 120.00, 10);
```

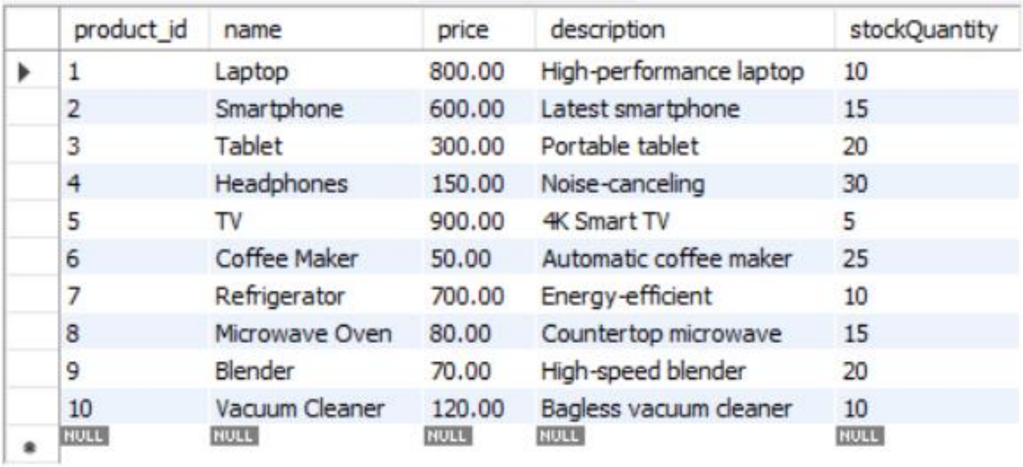
Explanation:

- Since product_id is AUTO_INCREMENT, we omit it in the INSERT statement.
- The database automatically assigns product_id values starting from 1 (or the next available number).
- The INSERT INTO command adds data into the products table.
- The columns specified: product_id, name, description, price, and stockQuantity.
- Values are inserted for each product.

Output:

To verify data insertion, I will run:

```
SELECT * FROM products;
```



	product_id	name	price	description	stockQuantity
▶	1	Laptop	800.00	High-performance laptop	10
	2	Smartphone	600.00	Latest smartphone	15
	3	Tablet	300.00	Portable tablet	20
	4	Headphones	150.00	Noise-canceling	30
	5	TV	900.00	4K Smart TV	5
	6	Coffee Maker	50.00	Automatic coffee maker	25
	7	Refrigerator	700.00	Energy-efficient	10
	8	Microwave Oven	80.00	Countertop microwave	15
	9	Blender	70.00	High-speed blender	20
	10	Vacuum Cleaner	120.00	Bagless vacuum cleaner	10
*	NULL	NULL	NULL	NULL	NULL

2. Inserting Data into the customers Table

Now, I’m inserting customer data into the customers table.

Since we need to store firstName, lastName, and address separately, we have to **alter** the customers table.

Alter Table - Add Columns

```
ALTER TABLE customers
ADD COLUMN firstName VARCHAR(50),
ADD COLUMN lastName VARCHAR(50),

ADD COLUMN address VARCHAR(255);
```

Now that we have the correct columns, I will insert the values.

```
INSERT INTO customers (firstName, lastName, name, email, password, address) VALUES

('John', 'Doe', 'John Doe', 'johndoe@example.com', 'pass123', '123 Main St, City'),

('Jane', 'Smith', 'Jane Smith', 'janesmith@example.com', 'secure456', '456 Elm St, Town'),

('Robert', 'Johnson', 'Robert Johnson', 'robert@example.com', 'robert789', '789 Oak St, Village'),

('Sarah', 'Brown', 'Sarah Brown', 'sarah@example.com', 'sarah101', '101 Pine St, Suburb'),

('David', 'Lee', 'David Lee', 'david@example.com', 'david234', '234 Cedar St, District'),

('Laura', 'Hall', 'Laura Hall', 'laura@example.com', 'laura567', '567 Birch St, County'),

('Michael', 'Davis', 'Michael Davis', 'michael@example.com', 'michael890', '890 Maple St, State'),

('Emma', 'Wilson', 'Emma Wilson', 'emma@example.com', 'emma321', '321 Redwood St, Country'),

('William', 'Taylor', 'William Taylor', 'william@example.com', 'william432', '432 Spruce St, Province'),

('Olivia', 'Adams', 'Olivia Adams', 'olivia@example.com', 'olivia765', '765 Fir St, Territory');
```

Explanation:

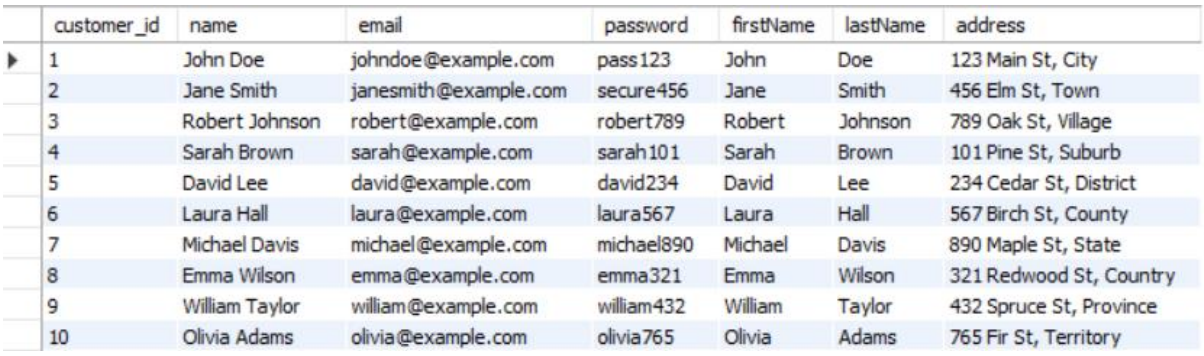
ALTER TABLE

- Adds firstName, lastName, and address columns to the customers table.

INSERT INTO customers

- Stores firstName and lastName separately while keeping name as a full name.
- Includes the email, password, and address for each customer.

Output :



customer_id	name	email	password	firstName	lastName	address
1	John Doe	johndoe@example.com	pass123	John	Doe	123 Main St, City
2	Jane Smith	janesmith@example.com	secure456	Jane	Smith	456 Elm St, Town
3	Robert Johnson	robert@example.com	robert789	Robert	Johnson	789 Oak St, Village
4	Sarah Brown	sarah@example.com	sarah101	Sarah	Brown	101 Pine St, Suburb
5	David Lee	david@example.com	david234	David	Lee	234 Cedar St, District
6	Laura Hall	laura@example.com	laura567	Laura	Hall	567 Birch St, County
7	Michael Davis	michael@example.com	michael890	Michael	Davis	890 Maple St, State
8	Emma Wilson	emma@example.com	emma321	Emma	Wilson	321 Redwood St, Country
9	William Taylor	william@example.com	william432	William	Taylor	432 Spruce St, Province
10	Olivia Adams	olivia@example.com	olivia765	Olivia	Adams	765 Fir St, Territory

3. Inserting Data into the orders Table

Now, I'm inserting customer data into the orders table.

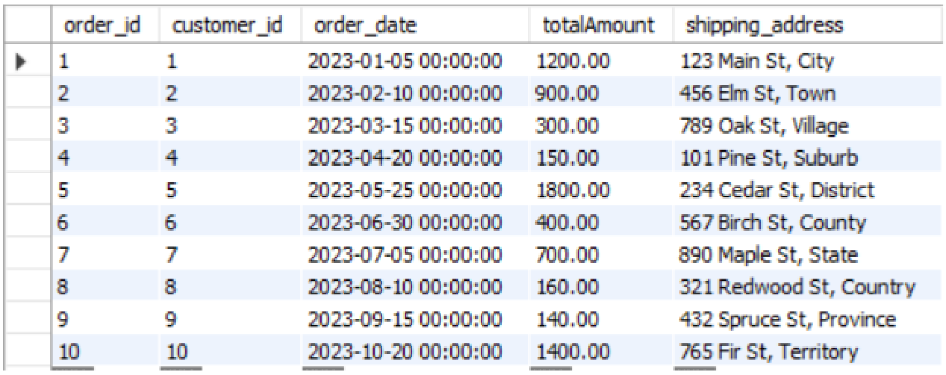
```
INSERT INTO orders (customer_id, order_date, totalAmount, shipping_address)
VALUES

(1, '2023-01-05', 1200.00, '123 Main St, City'),

(2, '2023-02-10', 900.00, '456 Elm St, Town'),

(3, '2023-03-15', 300.00, '789 Oak St, Village'),

(4, '2023-04-20', 150.00, '101 Pine St, Suburb'),

(5, '2023-05-25', 1800.00, '234 Cedar St, District'),

(6, '2023-06-30', 400.00, '567 Birch St, County'),

(7, '2023-07-05', 700.00, '890 Maple St, State'),

(8, '2023-08-10', 160.00, '321 Redwood St, Country'),

(9, '2023-09-15', 140.00, '432 Spruce St, Province'),

(10, '2023-10-20', 1400.00, '765 Fir St, Territory');
```

Explanation:

INSERT INTO orders

Auto-generates order_id since it's AUTO_INCREMENT. Links each order to a customer_id. Stores order date and total amount.

Output :



order_id	customer_id	order_date	totalAmount	shipping_address
1	1	2023-01-05 00:00:00	1200.00	123 Main St, City
2	2	2023-02-10 00:00:00	900.00	456 Elm St, Town
3	3	2023-03-15 00:00:00	300.00	789 Oak St, Village
4	4	2023-04-20 00:00:00	150.00	101 Pine St, Suburb
5	5	2023-05-25 00:00:00	1800.00	234 Cedar St, District
6	6	2023-06-30 00:00:00	400.00	567 Birch St, County
7	7	2023-07-05 00:00:00	700.00	890 Maple St, State
8	8	2023-08-10 00:00:00	160.00	321 Redwood St, Country
9	9	2023-09-15 00:00:00	140.00	432 Spruce St, Province
10	10	2023-10-20 00:00:00	1400.00	765 Fir St, Territory

4. Inserting Data into the order_item Table

Since the order_items table does not have an itemAmount column, we need to **alter** it first:

```
ALTER TABLE order_items
ADD COLUMN itemAmount DECIMAL(10,2);
```

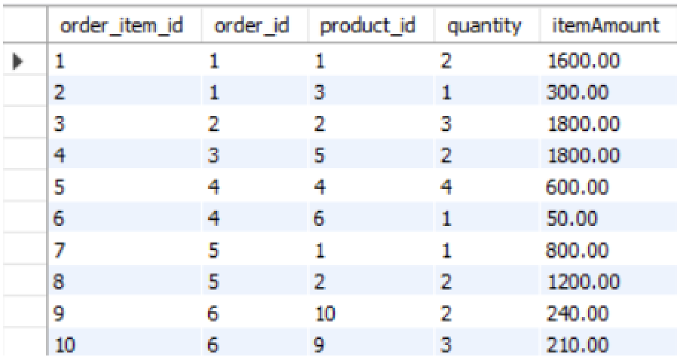
Now that the itemAmount column has been added, we can insert the given data:

```
INSERT INTO order_items (order_id, product_id, quantity, itemAmount) VALUES
(1, 1, 2, 1600.00),
(1, 3, 1, 300.00),
(2, 2, 3, 1800.00),
(3, 5, 2, 1800.00),
(4, 4, 4, 600.00),
(4, 6, 1, 50.00),
(5, 1, 1, 800.00),
(5, 2, 2, 1200.00),
(6, 10, 2, 240.00),
(6, 9, 3, 210.00);
```

Explanation:

- We insert records into order_items, where each row represents an item purchased in an order.
- The order_id references the orders table.
- The product_id references the products table.
- quantity denotes the number of units purchased.
- itemAmount is the total cost for that item (price * quantity).

Output :



order_item_id	order_id	product_id	quantity	itemAmount
1	1	1	2	1600.00
2	1	3	1	300.00
3	2	2	3	1800.00
4	3	5	2	1800.00
5	4	4	4	600.00
6	4	6	1	50.00
7	5	1	1	800.00
8	5	2	2	1200.00
9	6	10	2	240.00
10	6	9	3	210.00

5. Inserting Data into the cart Table

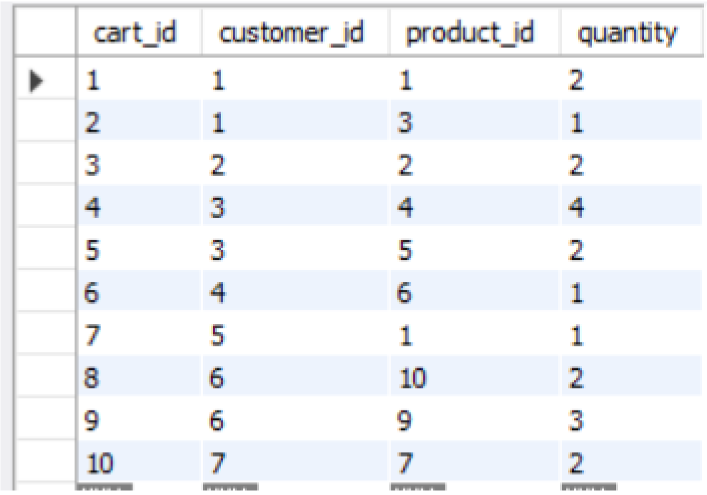
Now, I’m inserting cart data into the cart table.

```
INSERT INTO cart (customer_id, product_id, quantity) VALUES
(1, 1, 2),
(1, 3, 1),
(2, 2, 2),
(3, 4, 4),
(3, 5, 2),
(4, 6, 1),
(5, 1, 1),
(6, 10, 2),
(6, 9, 3),
(7, 7, 2);
```

Explanation:

- The INSERT INTO statement adds new records into the cart table.

Output :



cart_id	customer_id	product_id	quantity
1	1	1	2
2	1	3	1
3	2	2	2
4	3	4	4
5	3	5	2
6	4	6	1
7	5	1	1
8	6	10	2
9	6	9	3
10	7	7	2

5. Problems

1. Update Refrigerator Product Price to 800

I am updating the price of the product named **"Refrigerator"** in the products table to **800**.

Query:

UPDATE products SET price = 800

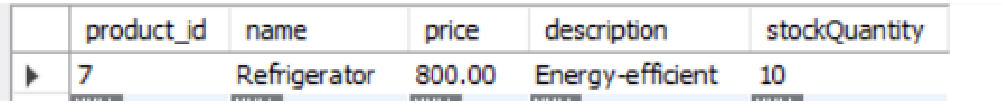
WHERE name = 'Refrigerator';

Explanation:

- UPDATE products specifies the table to modify.
- SET price = 800 updates the price column to **800**.
- WHERE name = 'Refrigerator' ensures only the refrigerator’s price is updated.

Output:

- The price of the Refrigerator will be updated to 800 in the products table.
- Running SELECT * FROM products WHERE name = 'Refrigerator'; will confirm the update.



	product_id	name	price	description	stockQuantity
▶	7	Refrigerator	800.00	Energy-efficient	10

---

2. Remove All Cart Items for a Specific Customer

I am removing all items from the cart for a specific customer based on their customer_id.

Query:

DELETE FROM cart

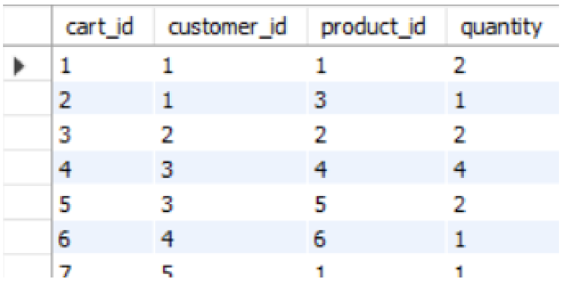
WHERE customer_id = 3;

Explanation:

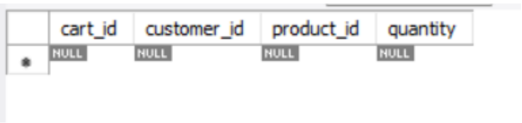
- DELETE FROM cart specifies that rows will be deleted from the cart table.
- WHERE customer_id = 3 ensures that only the cart items belonging to the customer with customer_id = 3 are removed.

Output:

- Running SELECT * FROM cart WHERE customer_id = 3; will return an empty result, confirming the deletion.



	cart_id	customer_id	product_id	quantity
▶	1	1	1	2
	2	1	3	1
	3	2	2	2
	4	3	4	4
	5	3	5	2
	6	4	6	1
	7	5	1	1



	cart_id	customer_id	product_id	quantity
*	NULL	NULL	NULL	NULL

Before . deletion After Deletion

3. Retrieve Products Priced Below $100

I am selecting all products from the products table where the price is less than 100.

Query:

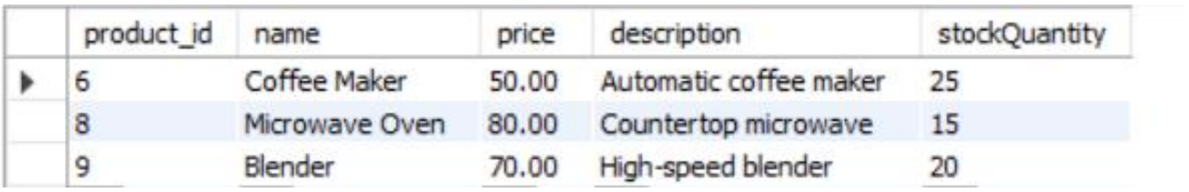
```
SELECT * FROM products
WHERE price < 100;
```

Explanation:

- SELECT * FROM products retrieves all columns from the products table.
- WHERE price < 100 filters the results to only include products with a price lower than 100.

Output:

- This will return all products priced below $100, such as:



product_id	name	price	description	stockQuantity
6	Coffee Maker	50.00	Automatic coffee maker	25
8	Microwave Oven	80.00	Countertop microwave	15
9	Blender	70.00	High-speed blender	20

4. Find Products with Stock Quantity Greater Than 5

I am retrieving all products from the products table where the stock quantity is greater than 5.

Query:

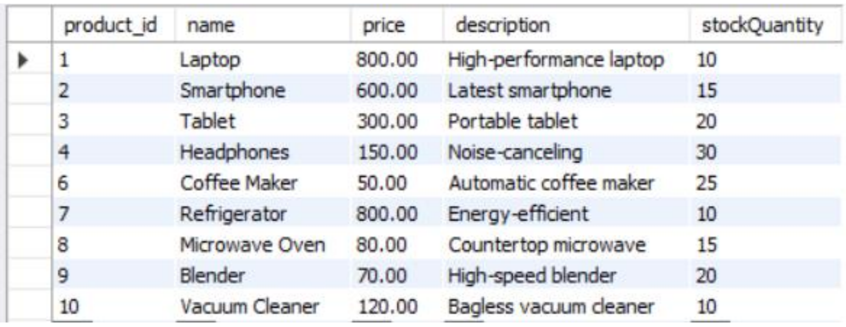
```
SELECT * FROM products
WHERE stockQuantity > 5;
```

Explanation:

- SELECT * FROM products selects all columns from the products table.
- WHERE stockQuantity > 5 filters the results to include only products that have a stock quantity greater than 5.

Output:

- This will return all products with stock quantities above 5, such as:



product_id	name	price	description	stockQuantity
1	Laptop	800.00	High-performance laptop	10
2	Smartphone	600.00	Latest smartphone	15
3	Tablet	300.00	Portable tablet	20
4	Headphones	150.00	Noise-canceling	30
6	Coffee Maker	50.00	Automatic coffee maker	25
7	Refrigerator	800.00	Energy-efficient	10
8	Microwave Oven	80.00	Countertop microwave	15
9	Blender	70.00	High-speed blender	20
10	Vacuum Cleaner	120.00	Bagless vacuum cleaner	10

5. Retrieve Orders with Total Amount Between $500 and $1000

I am retrieving all orders from the orders table where the total amount is between $500 and $1000.

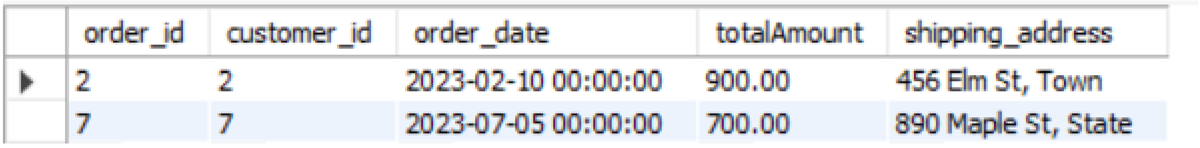
Query:

```
SELECT * FROM orders
WHERE totalAmount BETWEEN 500 AND 1000;
```

Explanation:

- SELECT * FROM orders selects all columns from the orders table.
- WHERE totalAmount BETWEEN 500 AND 1000 filters the results to include only orders with a total amount in the specified range.

Output:
This will return all orders where the total amount is between $500 and $1000, such as:



	order_id	customer_id	order_date	totalAmount	shipping_address
▶	2	2	2023-02-10 00:00:00	900.00	456 Elm St, Town
	7	7	2023-07-05 00:00:00	700.00	890 Maple St, State

6. Find Products Which Name End with Letter ‘r’

I am retrieving all products from the products table where the name ends with the letter 'r'.

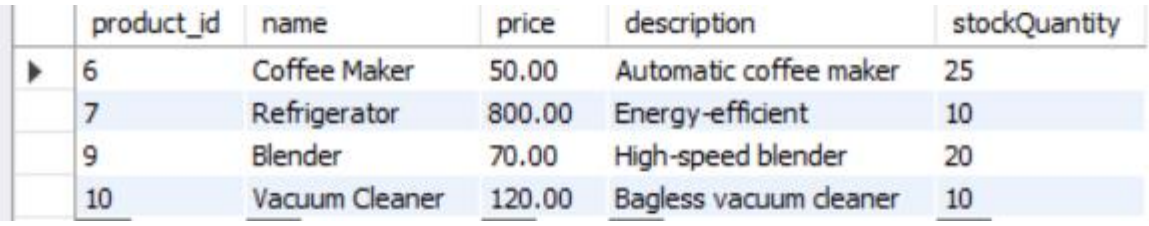
Query:

```
SELECT * FROM products
WHERE name LIKE '%r';
```

Explanation:

- SELECT * FROM products selects all columns from the products table.
- WHERE name LIKE '%r' filters products whose names end with the letter 'r'.
    - % is a wildcard that matches any number of characters before 'r'.

Output:
This will return products whose names end with 'r', such as:



	product_id	name	price	description	stockQuantity
▶	6	Coffee Maker	50.00	Automatic coffee maker	25
	7	Refrigerator	800.00	Energy-efficient	10
	9	Blender	70.00	High-speed blender	20
	10	Vacuum Cleaner	120.00	Bagless vacuum cleaner	10

7. Retrieve Cart Items for Customer 5

I am retrieving all cart items associated with customer ID 5 from the cart table.

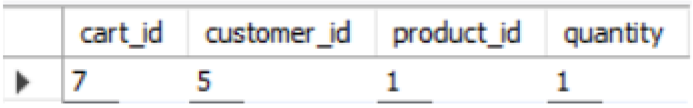
Query:

```
SELECT * FROM cart
WHERE customer_id = 5;
```

Explanation:

- SELECT * FROM cart selects all columns from the cart table.
- WHERE customer_id = 5 filters records to only include cart items belonging to customer ID 5.

Output:
This will return all products that customer 5 has added to their cart, such as:



cart_id	customer_id	product_id	quantity
7	5	1	1

8. Find Customers Who Placed Orders in 2023

I am retrieving the customers who have placed at least one order in the year 2023 from the orders table.

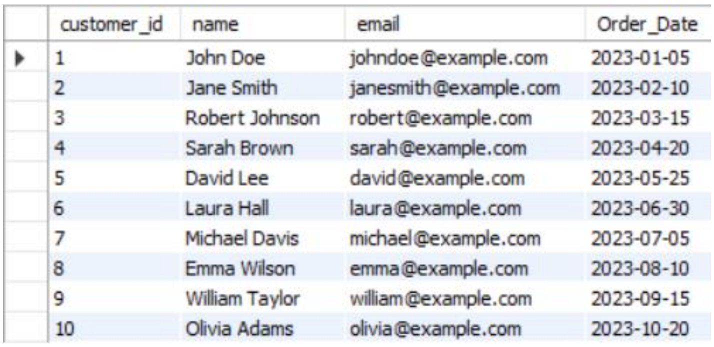
Query:

```
SELECT DISTINCT customers.customer_id, customers.name, customers.email , DATE(orders.order_date) AS Order_Date
FROM customers
JOIN orders ON customers.customer_id = orders.customer_id
WHERE YEAR(orders.order_date) = 2023;
```

Explanation:

- FROM customers JOIN orders ON customers.customer_id = orders.customer_id joins the customers and orders tables to link orders with their respective customers.
- WHERE YEAR(orders.order_date) = 2023 filters orders that were placed in the year 2023.

Output:
This will return a list of customers who placed orders in 2023, such as:



customer_id	name	email	Order_Date
1	John Doe	johndoe@example.com	2023-01-05
2	Jane Smith	janesmith@example.com	2023-02-10
3	Robert Johnson	robert@example.com	2023-03-15
4	Sarah Brown	sarah@example.com	2023-04-20
5	David Lee	david@example.com	2023-05-25
6	Laura Hall	laura@example.com	2023-06-30
7	Michael Davis	michael@example.com	2023-07-05
8	Emma Wilson	emma@example.com	2023-08-10
9	William Taylor	william@example.com	2023-09-15
10	Olivia Adams	olivia@example.com	2023-10-20

9. Determine the Minimum Stock Quantity for Each Product Category

I am retrieving the minimum stock quantity available for each product category from the products table.

For this , I need to modify the products table to include a category column and assign appropriate categories to each product.

```
ALTER TABLE products

ADD COLUMN category VARCHAR(50);
```

Then , need to update each product with a category

```
UPDATE products

SET category = 'Electronics' WHERE name IN ('Laptop', 'Smartphone', 'Tablet', 'Headphones', 'TV');

UPDATE products

SET category = 'Kitchen Appliance' WHERE name IN ('Coffee Maker', 'Microwave Oven', 'Blender');

UPDATE products

SET category = 'Home Appliance' WHERE name IN ('Refrigerator', 'Vacuum Cleaner');
```

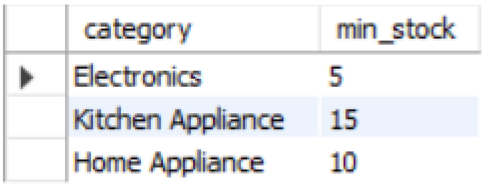
Query:

```
SELECT category, MIN(stockQuantity) AS min_stock

FROM products

GROUP BY category;
```

Explanation:

- SELECT category, MIN(stockQuantity) AS min_stock selects the product category and calculates the minimum stock quantity.
- FROM products specifies the table to fetch data from.
- GROUP BY category groups the results by product category to get the minimum stock for each category.

Output:
This will return the minimum stock quantity for each product category, such as:



category	min_stock
Electronics	5
Kitchen Appliance	15
Home Appliance	10

10. Calculate the Total Amount Spent by Each Customer

I am calculating the total amount spent by each customer based on their orders in the orders table.

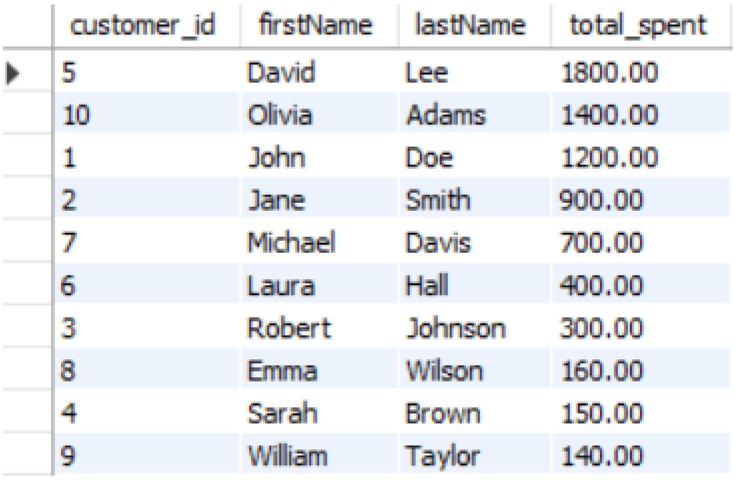
Query:

```
SELECT c.customer_id, c.firstName, c.lastName, SUM(o.totalAmount) AS total_spent

FROM customers c

JOIN orders o ON c.customer_id = o.customer_id

GROUP BY c.customer_id, c.firstName, c.lastName

ORDER BY total_spent DESC;
```

Explanation:

- SELECT c.customer_id, c.first_name, c.last_name, SUM(o.totalAmount) AS total_spent retrieves the customer details and the total amount they have spent.
- FROM customers specifies the table containing customer information.
- JOIN orders ON c.customer_id = o.customer_id links the orders table with the customers table using customer_id.
- GROUP BY c.customer_id, c.first_name, c.last_name groups the results by customer.
- ORDER BY total_spent DESC sorts the results in descending order, showing the highest spenders first.

Output:
This will return a list of customers along with the total amount they have spent, such as:



customer_id	firstName	lastName	total_spent
5	David	Lee	1800.00
10	Olivia	Adams	1400.00
1	John	Doe	1200.00
2	Jane	Smith	900.00
7	Michael	Davis	700.00
6	Laura	Hall	400.00
3	Robert	Johnson	300.00
8	Emma	Wilson	160.00
4	Sarah	Brown	150.00
9	William	Taylor	140.00

---

11. Find the Average Order Amount for Each Customer

I am calculating the average order amount for each customer based on their orders in the orders table.

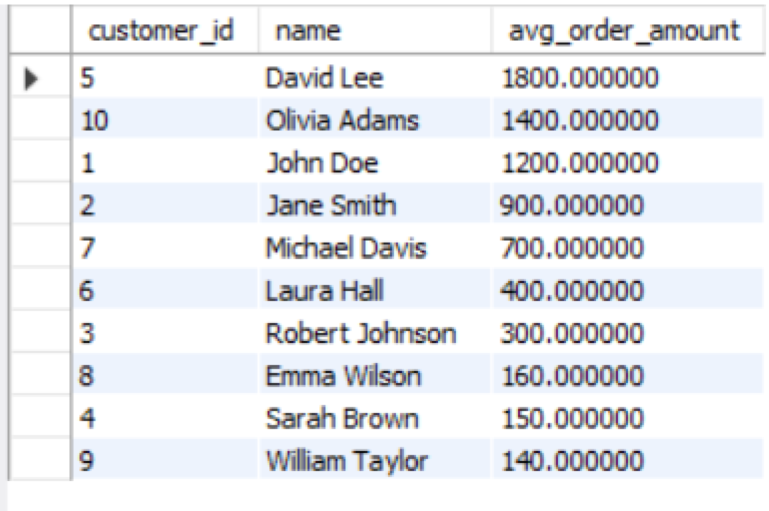
Query:

```
SELECT c.customer_id, c.name AVG(o.totalAmount) AS avg_order_amount
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.name
ORDER BY avg_order_amount DESC;
```

Explanation:

- SELECT c.customer_id, c.name, AVG(o.totalAmount) AS avg_order_amount retrieves the customer details along with their average order amount.
- FROM customers c assigns an alias c to the customers table for simplicity.
- JOIN orders o ON c.customer_id = o.customer_id assigns an alias o to the orders table and links it with customers using customer_id.
- GROUP BY c.customer_id, c.name groups the results by customer.
- ORDER BY avg_order_amount DESC sorts the results in descending order, showing customers with the highest average order amounts first.

Output:
This will return a list of customers along with their average order amount, such as:



customer_id	name	avg_order_amount
5	David Lee	1800.000000
10	Olivia Adams	1400.000000
1	John Doe	1200.000000
2	Jane Smith	900.000000
7	Michael Davis	700.000000
6	Laura Hall	400.000000
3	Robert Johnson	300.000000
8	Emma Wilson	160.000000
4	Sarah Brown	150.000000
9	William Taylor	140.000000

---

12. Count the Number of Orders Placed by Each Customer

I am counting the number of orders placed by each customer using the orders table.

Query:

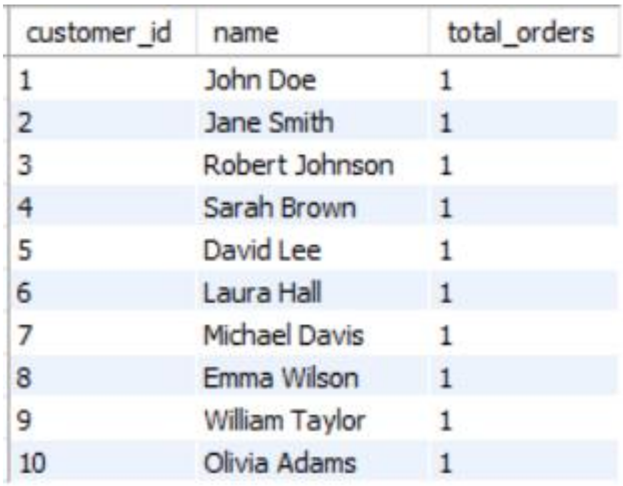
```
SELECT c.customer_id, c.name, COUNT(o.order_id) AS total_orders
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
```

```
GROUP BY c.customer_id, c.name;
```

Explanation:

- SELECT c.customer_id, c.name, COUNT(o.order_id) AS total_orders: Retrieves the customer ID, name, and the total number of orders they placed.
- FROM customers c LEFT JOIN orders o ON c.customer_id = o.customer_id: Joins the customers table with the orders table based on customer_id, ensuring all customers are included even if they haven’t placed an order.
- GROUP BY c.customer_id, c.name: Groups results by customer to count their respective orders.

Output:



customer_id	name	total_orders
1	John Doe	1
2	Jane Smith	1
3	Robert Johnson	1
4	Sarah Brown	1
5	David Lee	1
6	Laura Hall	1
7	Michael Davis	1
8	Emma Wilson	1
9	William Taylor	1
10	Olivia Adams	1

13. Find the Maximum Order Amount for Each Customer

I am retrieving the highest order amount placed by each customer from the orders table.

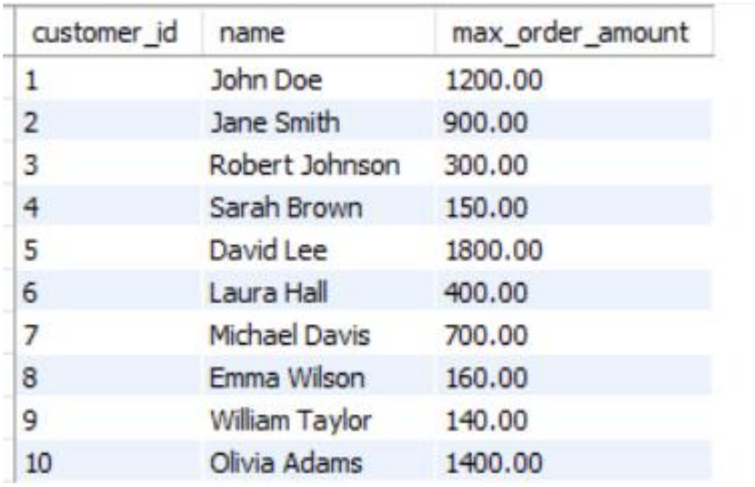
Query:

```
SELECT c.customer_id, c.name, MAX(o.totalAmount) AS max_order_amount

FROM customers c

LEFT JOIN orders o ON c.customer_id = o.customer_id

GROUP BY c.customer_id, c.name;
```

Explanation:

- SELECT c.customer_id, c.name, MAX(o.totalAmount) AS max_order_amount: Fetches the customer ID, name, and the maximum order amount they have placed.
- FROM customers c LEFT JOIN orders o ON c.customer_id = o.customer_id: Joins customers with orders based on customer_id, ensuring all customers are included even if they have no orders.
- GROUP BY c.customer_id, c.name: Groups the results by customer to find their maximum order amount.

Output:



customer_id	name	max_order_amount
1	John Doe	1200.00
2	Jane Smith	900.00
3	Robert Johnson	300.00
4	Sarah Brown	150.00
5	David Lee	1800.00
6	Laura Hall	400.00
7	Michael Davis	700.00
8	Emma Wilson	160.00
9	William Taylor	140.00
10	Olivia Adams	1400.00

---

14. Get Customers Who Placed Orders Totaling Over $1000

I am retrieving customers whose total order amount exceeds $1000.

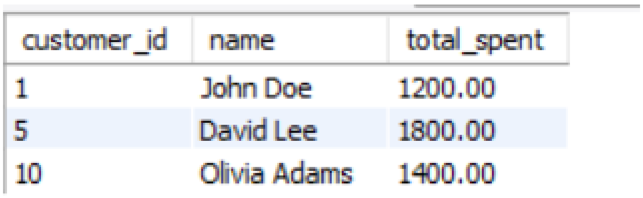
Query:

```
SELECT c.customer_id, c.name, SUM(o.totalAmount) AS total_spent

FROM customers c

JOIN orders o ON c.customer_id = o.customer_id

GROUP BY c.customer_id, c.name

HAVING SUM(o.totalAmount) > 1000;
```

Explanation:

- SELECT c.customer_id, c.name, SUM(o.totalAmount) AS total_spent: Retrieves the customer ID, name, and total amount spent.
- FROM customers c JOIN orders o ON c.customer_id = o.customer_id: Joins customers with orders based on customer_id to get order details.
- GROUP BY c.customer_id, c.name: Groups by customer to calculate total spending.
- HAVING SUM(o.totalAmount) > 1000: Filters out customers whose total spending is greater than $1000.

Output:



customer_id	name	total_spent
1	John Doe	1200.00
5	David Lee	1800.00
10	Olivia Adams	1400.00

---

15. Subquery to Find Products Not in the Cart

I am retrieving products that are not present in any cart.

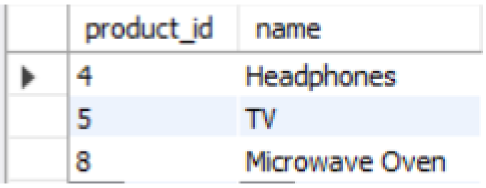
Query:

```
SELECT p.product_id, p.name
FROM products p
WHERE p.product_id NOT IN (SELECT DISTINCT c.product_id FROM cart c);
```

Explanation:

- SELECT p.product_id, p.name: Retrieves product details.
- FROM products p: Queries the products table.
- WHERE p.product_id NOT IN (...): Filters products that are **not** in the cart.
- The subquery SELECT DISTINCT c.product_id FROM cart c retrieves all unique product IDs present in the cart.
- The NOT IN condition ensures only products **missing** from the cart are selected.

Output:



product_id	name
4	Headphones
5	TV
8	Microwave Oven

---

16. Subquery to Find Customers Who Haven't Placed Orders

I am retrieving customers who have not placed any orders.

To test this query, I will first insert a new customer who has **not placed any orders**.

```
INSERT INTO customers (name, email, password, firstName, lastName, address)
VALUES ('Sam Wilson', 'sam@example.com', 'password123', 'Sam', 'Wilson', '456 Green St, City');
```

Now, we can execute the **subquery to find customers who haven't placed orders**:

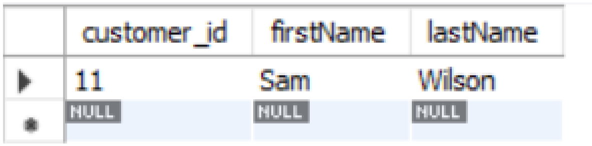
Query:

```
SELECT c.customer_id, c.first_name, c.last_name
FROM customers c
WHERE c.customer_id NOT IN (SELECT DISTINCT o.customer_id FROM orders o);
```

Explanation:

- SELECT c.customer_id, c.first_name, c.last_name: Retrieves customer details.
- FROM customers c: Queries the customers table.
- WHERE c.customer_id NOT IN (...): Filters customers who have **not** placed an order.
- The subquery SELECT DISTINCT o.customer_id FROM orders o retrieves all unique customer IDs who have placed orders.
- The NOT IN condition ensures only customers **without** orders are selected.

Output:



customer_id	firstName	lastName
11	Sam	Wilson
NULL	NULL	NULL

17. Subquery to Calculate the Percentage of Total Revenue for a Product

I am calculating the percentage contribution of each product's total sales (itemAmount) to the overall revenue from the order_items table.

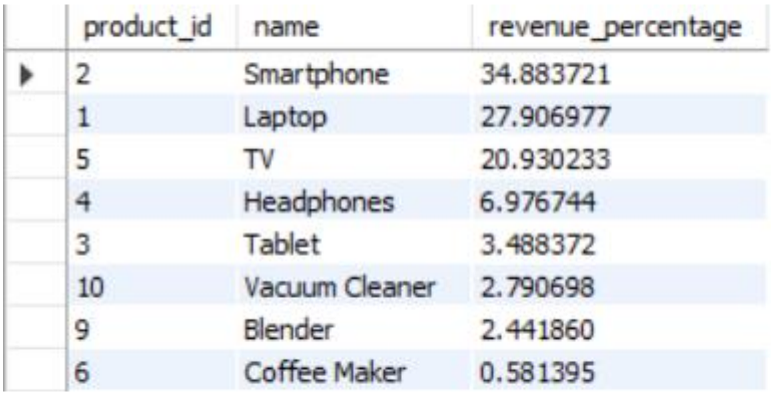
Query:

```
SELECT
   p.product_id,
   p.name,
   (SUM(oi.itemAmount) / (SELECT SUM(itemAmount) FROM order_items) * 100) AS revenue_percentage
FROM products p
JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY p.product_id, p.name
ORDER BY revenue_percentage DESC;
```

Explanation:

- SUM(oi.itemAmount) gets the total revenue from each product.
- (SELECT SUM(itemAmount) FROM order_items) calculates the total revenue from all products.
- The division (SUM(oi.itemAmount) / total_revenue) * 100 finds the percentage contribution.
- GROUP BY p.product_id, p.name ensures each product is listed separately.
- ORDER BY revenue_percentage DESC sorts products by highest revenue percentage.

Output :



product_id	name	revenue_percentage
2	Smartphone	34.883721
1	Laptop	27.906977
5	TV	20.930233
4	Headphones	6.976744
3	Tablet	3.488372
10	Vacuum Cleaner	2.790698
9	Blender	2.441860
6	Coffee Maker	0.581395

18. Subquery to Find Products with Low Stock

I am retrieving products whose stock quantity is lower than the average stock quantity of all products.

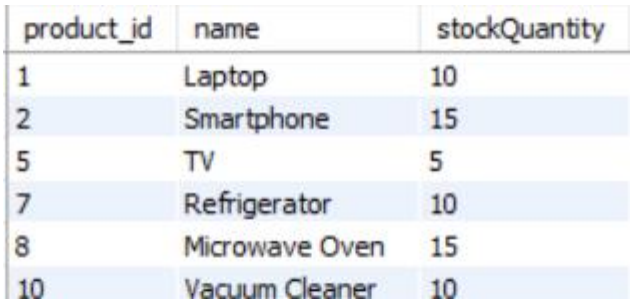
Query:

```
SELECT
    product_id,
    name,
    stockQuantity
FROM products
WHERE stockQuantity < (SELECT AVG(stockQuantity) FROM products);
```

Explanation:

- SELECT AVG(stockQuantity) FROM products calculates the average stock quantity of all products.
- WHERE stockQuantity < (subquery) filters out products with stock below this average.
- This helps identify low-stock products that may need restocking.

Output :



product_id	name	stockQuantity
1	Laptop	10
2	Smartphone	15
5	TV	5
7	Refrigerator	10
8	Microwave Oven	15
10	Vacuum Cleaner	10

19. Subquery to Find Customers Who Placed High-Value Orders

I am retrieving customers who have placed at least one order with a total amount greater than the average order amount.

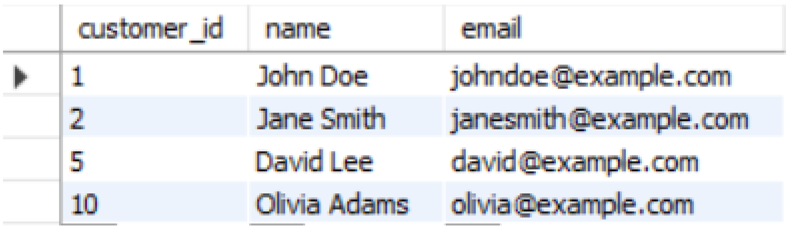
Query:

```
SELECT
   c.customer_id,   c.name,   c.email
FROM customers c
WHERE c.customer_id IN (
   SELECT o.customer_id
   FROM orders o
   WHERE o.totalAmount > (SELECT AVG(totalAmount) FROM orders)
);
```

Explanation:

- (SELECT AVG(totalAmount) FROM orders) calculates the average order total.
- The inner subquery retrieves customer_ids of those who placed orders above this average.
- The outer query fetches customer details for these high-value orders.

Output :



customer_id	name	email
1	John Doe	johndoe@example.com
2	Jane Smith	janesmith@example.com
5	David Lee	david@example.com
10	Olivia Adams	olivia@example.com

Conclusion

Through this series of SQL queries, I performed various operations on the database, including updating records, retrieving specific data, filtering information using conditions, and utilizing subqueries for complex data retrieval. These queries helped in:

- Modifying product details (price updates, stock adjustments).Managing customer orders and cart items efficiently.
- Extracting meaningful insights, such as customer spending patterns, high-value orders, and low-stock products.
- Using subqueries to analyze data relationships, such as finding customers who haven't placed orders or products not in the cart.

By working with these queries, I can efficiently manage e-commerce databases, optimize business logic, and enhance data-driven decision-making.