E-Commerce Platform Database

# Database Schema

# Create Users Table

CREATE TABLE Users (

user\_id INT AUTO\_INCREMENT PRIMARY KEY,

username VARCHAR(50) NOT NULL,

email VARCHAR(50) NOT NULL UNIQUE,

password VARCHAR(50) NOT NULL,

role VARCHAR(50) NOT NULL

);

# Create Products Table

CREATE TABLE Products (

product\_id INT AUTO\_INCREMENT PRIMARY KEY,

product\_name VARCHAR(50) NOT NULL,

category VARCHAR(50),

price DECIMAL(10, 2) NOT NULL,

stock\_quantity INT NOT NULL

);

# Create Orders Table

CREATE TABLE Orders (

order\_id INT AUTO\_INCREMENT PRIMARY KEY,

user\_id INT,

order\_date DATETIME DEFAULT CURRENT\_TIMESTAMP,

total\_amount DECIMAL(10, 2) NOT NULL,

order\_status VARCHAR(50) DEFAULT 'pending',

FOREIGN KEY (user\_id) REFERENCES Users(user\_id)

);

# Create OrderDetails Table

CREATE TABLE OrderDetails (

order\_detail\_id INT AUTO\_INCREMENT PRIMARY KEY,

order\_id INT,

product\_id INT,

quantity INT NOT NULL,

unit\_price DECIMAL(10, 2) NOT NULL,

FOREIGN KEY (order\_id) REFERENCES Orders(order\_id),

FOREIGN KEY (product\_id) REFERENCES Products(product\_id)

);

# Create Payments Table

CREATE TABLE Payments (

payment\_id INT AUTO\_INCREMENT PRIMARY KEY,

order\_id INT,

payment\_date DATETIME DEFAULT CURRENT\_TIMESTAMP,

payment\_method VARCHAR(50) NOT NULL,

amount DECIMAL(10, 2) NOT NULL,

FOREIGN KEY (order\_id) REFERENCES Orders(order\_id)

);

# Create Reviews Table

CREATE TABLE Reviews (

review\_id INT AUTO\_INCREMENT PRIMARY KEY,

product\_id INT,

user\_id INT,

review\_text TEXT,

rating INT CHECK(rating BETWEEN 1 AND 5),

review\_date DATETIME DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (product\_id) REFERENCES Products(product\_id),

FOREIGN KEY (user\_id) REFERENCES Users(user\_id)

);

For the database schema, I followed the projects names and columns adding only constraints to connect the tables, such as keys, and some specific constraints. In general, most values are not null except for the foreign keys. Email has to be unique as every one is different. dates are type datetime with the current date. Review is type text because the length of reviews can’t be controlled, and rating must be between 1 and 5 stars.

# Sample Data

# Insert sample users

INSERT INTO Users (username, email, password, role) VALUES

('john\_doe', 'john@example.com', 'password123', 'customer'),

('jane\_doe', 'jane@example.com', 'password456', 'customer'),

('admin\_user', 'admin@example.com', 'adminpassword', 'admin');

# Insert sample products

INSERT INTO Products (product\_name, category, price, stock\_quantity) VALUES

('Smartphone', 'Electronics', 599.99, 50),

('Laptop', 'Electronics', 899.99, 30),

('Headphones', 'Electronics', 199.99, 100),

('Coffee Maker', 'Home Appliances', 49.99, 200),

('Running Shoes', 'Sportswear', 79.99, 150);

# Insert sample orders

INSERT INTO Orders (user\_id, total\_amount, order\_status) VALUES

(1, 799.98, 'shipped'),

(2, 1499.98, 'pending'),

(1, 49.99, 'delivered');

# Insert order details

INSERT INTO OrderDetails (order\_id, product\_id, quantity, unit\_price) VALUES

(1, 1, 1, 599.99),

(1, 3, 1, 199.99),

(2, 2, 1, 899.99),

(3, 4, 1, 49.99);

# Insert sample payments

INSERT INTO Payments (order\_id, payment\_method, amount) VALUES

(1, 'credit\_card', 799.98),

(2, 'paypal', 1499.98),

(3, 'credit\_card', 49.99);

# Insert sample reviews

INSERT INTO Reviews (product\_id, user\_id, review\_text, rating) VALUES

(1, 1, 'Great smartphone with amazing features!', 5),

(2, 2, 'Laptop works perfectly for my needs.', 4),

(3, 1, 'The headphones have great sound quality.', 5),

(4, 2, 'The coffee maker is easy to use and brews quickly.', 4);

For the sample data, I used ChatGPT to generate it. I ensured that the data was diverse enough for different examples during the queries. Adding the columns helped to ensure which value was going to which field and not confuse myself.

# Explanations

# Retrieve the list of all products in a specific category

SELECT product\_name, price, stock\_quantity

FROM Products

WHERE category = 'Electronics';

The user doesn’t need to know the product id so I left it out. While leaving category out, since that is what we are searching, I included the rest of the columns and put the condition for the category with more fields.

# Retrieve the details of a specific user by providing their user\_id

SELECT \*

FROM Users

WHERE user\_id = 1;

Since we’re searching with the id, I included every column from users. Only the condition needs to change.

# Retrieve the order history for a particular user

SELECT \*

FROM Orders

WHERE user\_id = 2;

Same as above. Since we’re searching with the id, I included every column from orders. Only the condition needs to change.

# Retrieve the products in an order along with their quantities and prices

SELECT p.product\_name, od.quantity, od.unit\_price

FROM OrderDetails od

JOIN Products p ON od.product\_id = p.product\_id

WHERE od.order\_id = 1;

This query requires a join between OrderDetails and products in order to get the product name. From there, add the quantity and price of the order and a condition to search by id.

# Retrieve the average rating of a product

SELECT AVG(rating) AS average\_rating

FROM Reviews

WHERE product\_id = 2;

As this query requires the average, I added an aggregation of rating with a function of average. The alias is there to clarify what I’m aggregating. The condition can be easily changed to search by a different id.

# Retrieve the total revenue for a given month

SELECT SUM(amount) AS total\_revenue

FROM Payments

WHERE MONTH(payment\_date) = 5 AND YEAR(payment\_date) = 2025;

Like the previous query, this one has an aggregation of the sum of the amount with an alias. From there, I used the function month from the datetime datatype to specify the month. Afterwards, I added the year function to separate the sales of the month by year instead of having the sum of all the sales of an specific month cluster together.

# Add a new product to the inventory

INSERT INTO Products (product\_name, category, price, stock\_quantity) VALUES

('Smartwatch', 'Electronics', 199.99, 75);

A simple insert into products while ensuring all the fields are correct.

# Place a new order for a user

INSERT INTO Orders (user\_id, total\_amount, order\_status) VALUES

(1, 699.99, 'pending');

INSERT INTO OrderDetails (order\_id, product\_id, quantity, unit\_price) VALUES

(4, 1, 1, 599.99),

(4, 5, 1, 79.99);

Same as above, but this time there is also an input into orderDetails to indicate what was bought in the new order.

# Update the stock quantity of a product

UPDATE Products

SET stock\_quantity = stock\_quantity - 1

WHERE product\_id = 1;

A change to the amount of a product based from the previous query. I wasn’t sure if the – decrement or the -= contraction worked so I used the whole name for the update.

# Remove a user's review

DELETE FROM Reviews

WHERE review\_id = 2;

A simple delete of a single review. If there is WHERE clause there is a danger of deleting all the fields, so there is a need to be careful.

# Identify the top-selling products

SELECT p.product\_name, SUM(od.quantity) AS total\_sold

FROM OrderDetails od

JOIN Products p ON od.product\_id = p.product\_id

GROUP BY od.product\_id

ORDER BY total\_sold DESC

LIMIT 5;

A join between OrderDetails and Products to determine with an aggregation which product has been ordered the most. There is an alias in the sum in order to not repeat the whole sum part for the ORDER BY from greater to lower. There is a limit to indicate the top products.

# Find users who have placed orders exceeding a certain amount

SELECT u.username, SUM(o.total\_amount) AS total\_spent

FROM Orders o

JOIN Users u ON o.user\_id = u.user\_id

GROUP BY o.user\_id

HAVING total\_spent > 1000;

A join between Orders and Users to find the user who spend most in the orders. The sum has an alias to reduce what needs to be written in HAVING. Having, for its part, is used for aggregations instead of WHERE. The condition indicates the certain amount and can be changed as needed.

# Calculate the overall average rating for each product category

SELECT p.category, AVG(r.rating) AS average\_rating

FROM Reviews r

JOIN Products p ON r.product\_id = p.product\_id

GROUP BY p.category;

A join between the product categories and their reviews. The average can be determined through an aggrupation. In this case, I’m searching by category so the grouping uses it too.

# Automatically update the order status based on order processing

Couldn’t determine how to properly make this query. From investigating, I noticed that I needed a trigger and an enumeration to determine the possible status of the order.

# Generate a report on the most active users

SELECT u.username, COUNT(o.order\_id) AS number\_of\_orders

FROM Orders o

JOIN Users u ON o.user\_id = u.user\_id

GROUP BY o.user\_id

ORDER BY number\_of\_orders DESC

LIMIT 5;

A join between the users and a count of how many orders each of them made. Grouped by the username since it is the constant and order by the count of orders, from greater to less. The alias helps to reduce the size of the query and the limit to indicate that only the top is being searched.