­­­

|  |
| --- |
| ARDEN UNIVERSITY |
| BSC (HONS)COMPUTING |
| ADVANCED DATABASE |
| ADVANCED DATABASE |
|  |
| KRITHIGA DURAISAMY |
| 2700 |

Contents

[TASK - 1 3](#_Toc171985692)

[ERD DIAGRAM (entity relationship diagram) 3](#_Toc171985693)

[TASK 2 6](#_Toc171985694)

[CREATING VIEW 15](#_Toc171985695)

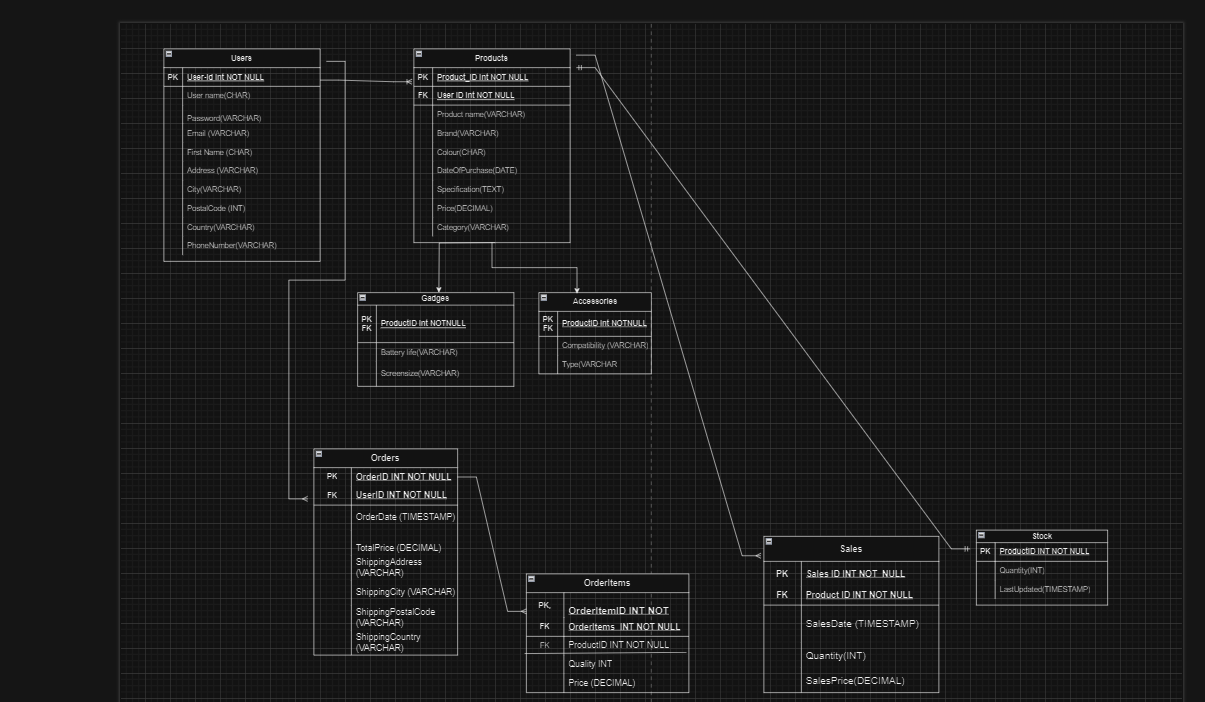
[STORED PROCEDURES 20](#_Toc171985696)

[TRIGGERS 24](#_Toc171985697)

[REFERENCES 26](#_Toc171985698)

# TASK - 1

# ERD DIAGRAM (entity relationship diagram)



Entities

Users (UserID, Username, Password, Email, FirstName, LastName, Address, City, PostalCode, Country, PhoneNumber) Products (ProductID, Name, Brand, Model, Colour, DateOfPurchase, Specification, Condition, Price, UserID, Category) Gadgets (ProductID, BatteryLife,

ScreenSize) Accessories (ProductID, Compatibility, Type) Orders (OrderID, UserID, OrderDate, TotalPrice, ShippingAddress, ShippingCity, ShippingPostalCode, ShippingCountry) OrderItems (OrderItemID, OrderID, ProductID, Quantity, Price) Sales (SaleID, ProductID, SaleDate, Quantity, SalePrice) Stock (ProductID, Quantity, LastUpdated)

Relationships

Users --< Products: One user can have many products.

Products --< OrderItems: One product can be in many order items.

Orders --< OrderItems: One order can have many order items.

Users --< Orders: One user can place many orders.

Products --< Sales: One product can have many sales records.

Products -- Stock: One product has one stock record.

Products (inheritance) -> Gadgets: A gadget is a type of product.

Products (inheritance) -> Accessories: An accessory is a type of product.

Super/Sub-typing (Inheritance)

Super/Sub-typing is done using table inheritance in SQL, and it is the primary method of effectively managing data between several product categories. This was particularly chosen to ensure that the database could support maintainability and extension. Central management of shared attributes helps reduce the level of redundancy and possible inconsistencies which can result from the creation of separate sub-types, the likes of Gadgets and Accessories, while a generalized Products database structures the super-type. This inheritance technique leverages the strong table inheritance that PostgreSQL offers to allow for effective querying and indexing of both the super-type and the sub-type tables. Table inheritance not only helps improve query efficiency when working with hierarchical data structures but it also helps make the database layout simpler. Additionally, table inheritance fits well with object-oriented principles archetype as it encourages discrete separations of specialized and generalized data which is beneficial for complex applications that require a wide array of interconnected datasets.

De-normalization

Denormalization involves the inclusion of shipping-specific information that includes ShippingAddress, ShippingCity, ShippingPostalCode, and ShippingCountry into address. Denormalization was included in this database mainly for performance optimization. In this situation, denormalization is often used in read-heavy scenarios where real-time performance is essential. For an e-commerce platform to run smoothly and efficiently, users must be able to access order and shipping information immediately. Ponniah Reveals that this strategic denormalization enables significant performance improvement, especially in high-transaction scenarios.

In addition, as the shipping information is typically not editable after an order request has been placed using this approach helps minimise issues like data anomalies and it alleviates common concerns about denormalised including inconsistent data or unusual updates.

Non-key Indexes

This lead to a non-key index added on the Products table to help improve search capabilities. You can significantly increase query performance by indexing this field, as product searches are likely to be one of the highest trafficked operations on your site.

Non-key indexing is a valid method for increasing the performance of search queries (mainly in read- intensive databases ) As Stonebraker (2018) notes, non-key indexes can be even better for large datasets because it allows higher data retrieval performance and fewer computational overhead. Indexing the Name attribute will allow this so that rows which will already have matches for our search criteria, can be found rapidly from database and user experience would improve as they get results in secs rather than having to wait minutes.

This is especially applicable in the world of online marketplaces, where users demand immediate and accurate search results. The trade-off is thus generally satisfied by the main use case of those indexes: to aid frequent and fast read operations - making a performance jump over very small amount of extra storage not worth it, yet without an effect on write time.

CONCLUTION

The denormalization, super/sub-typing and non-key indexing methods chosen were meant to meet the functional as well performance-driven requirements of the online platform. Inheritance: Enables inheritance between tables (ie., table is child of parent table) thus permitting a structured, organized schema while still enforcing the same performance characteristics for read-heavy operations that we get by selectively denormalizing Orders as an alternative to data joins. One look at the database design we can confirm that it is solid and also works for what was planned, by creating non-key indexes on keys which are frequently searched properties helps to optimize queries even more. This reliance on industry best practices and rules enforced by some of the biggest players in warehousing design ensures that this database system is ready to handle the challenges as well as complexities associated with used electronic devices and accessories.

# TASK 2

IMPLEMENTATION

CREATING TABLES

/\*Create Users Table\*/

CREATE TABLE Users (

UserID SERIAL PRIMARY KEY,

Username VARCHAR(50) UNIQUE NOT NULL,

Password VARCHAR(255) NOT NULL,

Email VARCHAR(100) UNIQUE NOT NULL,

FirstName VARCHAR(50),

LastName VARCHAR(50),

Address VARCHAR(255),

City VARCHAR(100),

PostalCode VARCHAR(20),

Country VARCHAR(50),

PhoneNumber VARCHAR(20)

);

/\*Create Products Table (Super-Type)\*/

CREATE TABLE Products (

ProductID SERIAL PRIMARY KEY,

Name VARCHAR(100) NOT NULL,

Brand VARCHAR(50),

Model VARCHAR(50),

Colour VARCHAR(20),

DateOfPurchase DATE,

Specification TEXT,

ProductCondition VARCHAR(20) CHECK (ProductCondition IN ('New', 'Like New', 'Used', 'Refurbished')),

Price DECIMAL(10, 2) NOT NULL,

UserID INT REFERENCES Users(UserID),

Category VARCHAR(50)

);

/\*3. Create Gadgets Table (Sub-Type)\*/

CREATE TABLE Gadgets (

ProductID SERIAL PRIMARY KEY,

BatteryLife VARCHAR(50),

ScreenSize VARCHAR(50),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID)

);

/\*4. Create Accessories Table (Sub-Type)\*/

CREATE TABLE Accessories (

ProductID SERIAL PRIMARY KEY,

Compatibility VARCHAR(50),

Type VARCHAR(50),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID)

);

/\*Create Orders Table\*/

CREATE TABLE Orders (

OrderID SERIAL PRIMARY KEY,

UserID INT REFERENCES Users(UserID),

OrderDate TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

TotalPrice DECIMAL(10, 2) NOT NULL,

ShippingAddress VARCHAR(255),

ShippingCity VARCHAR(100),

ShippingPostalCode VARCHAR(20),

ShippingCountry VARCHAR(50)

);

/\*6. Create OrderItems Table\*/

CREATE TABLE OrderItems (

OrderItemID SERIAL PRIMARY KEY,

OrderID INT REFERENCES Orders(OrderID),

ProductID INT REFERENCES Products(ProductID),

Quantity INT NOT NULL,

Price DECIMAL(10, 2) NOT NULL

);

/\*7. Create Sales Table\*/

CREATE TABLE Sales (

SaleID SERIAL PRIMARY KEY,

ProductID INT REFERENCES Products(ProductID),

SaleDate TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

Quantity INT NOT NULL,

SalePrice DECIMAL(10, 2) NOT NULL

);

/\*8. Create Stock Table\*/

CREATE TABLE Stock (

ProductID INT PRIMARY KEY REFERENCES Products(ProductID),

Quantity INT NOT NULL CHECK (Quantity >= 0),

LastUpdated TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

/\*9. Create Indexes\*/

CREATE INDEX idx\_product\_name ON Products(Name);

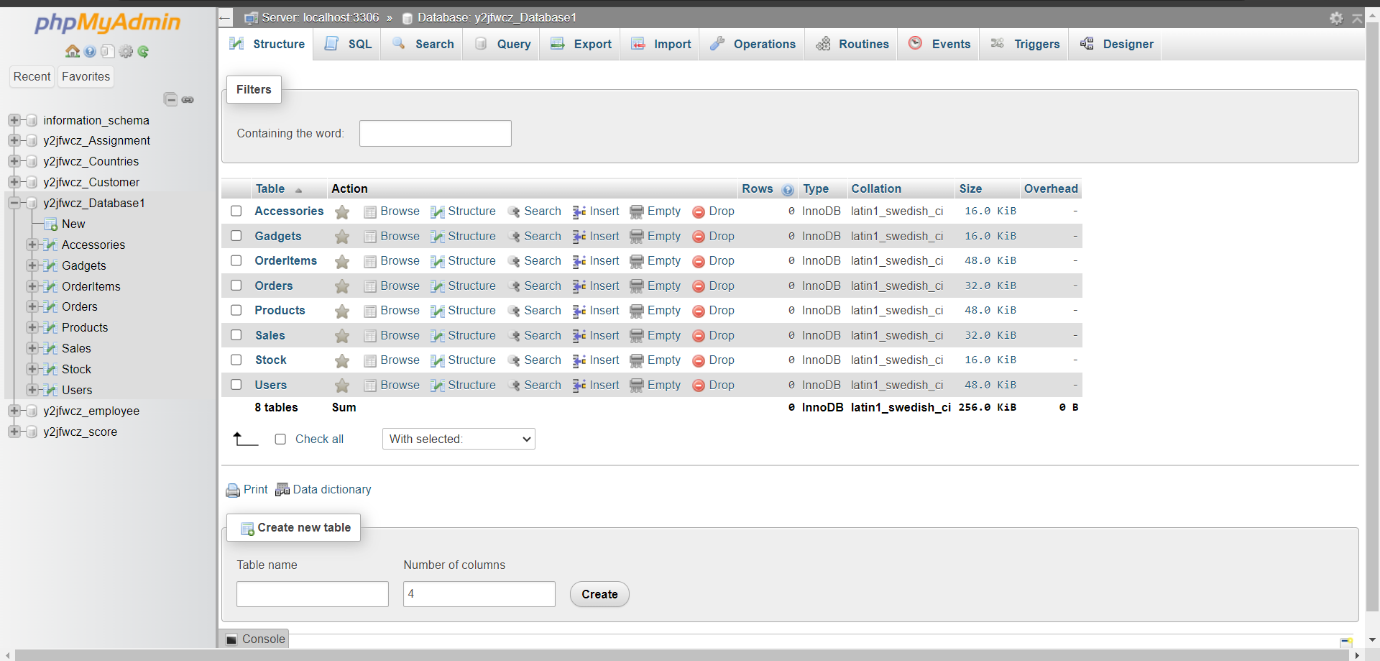


Fig 1.1

This is the tables created in Database\_1

INSERTING DATA

1. INSERTING DATA INTO USERS TABLE

INSERT INTO Users (Username, Password, Email, FirstName, LastName, Address, City, PostalCode, Country, PhoneNumber)

VALUES

('john\_doe', 'hashed\_password1', 'john@example.com', 'John', 'Doe', '123 Elm Street', 'London', 'E1 6AN', 'UK', '07123456789'),

('jane\_smith', 'hashed\_password2', 'jane@example.com', 'Jane', 'Smith', '456 Oak Street', 'Manchester', 'M1 3AU', 'UK', '07234567890'),

('alice\_jones', 'hashed\_password3', 'alice@example.com', 'Alice', 'Jones', '789 Pine Street', 'Birmingham', 'B2 5AL', 'UK', '07345678901');

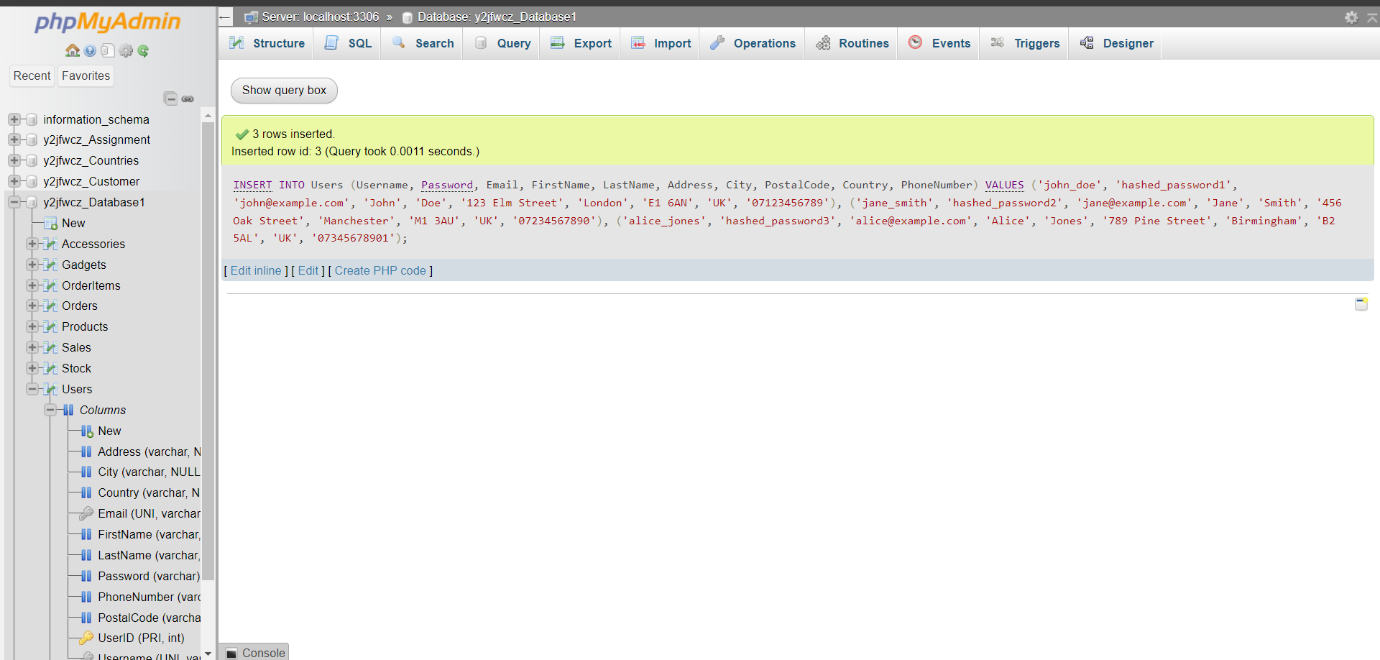


FIG 2.1

1. INSERTING DATA INTO PRODUCTS TABLE

INSERT INTO Products (Name, Brand, Model, Colour, DateOfPurchase, Specification, ProductCondition, Price, UserID, Category)

VALUES

('iPhone 11', 'Apple', 'A2221', 'Black', '2019-09-20', '64GB, Dual SIM', 'Used', 450.00, 1, 'Gadget'),

('Galaxy S10', 'Samsung', 'SM-G973F', 'White', '2019-03-08', '128GB, Dual SIM', 'Like New', 500.00, 2, 'Gadget'),

('AirPods Pro', 'Apple', 'A2084', 'White', '2020-01-15', 'Active Noise Cancellation', 'New', 200.00, 3, 'Accessory');

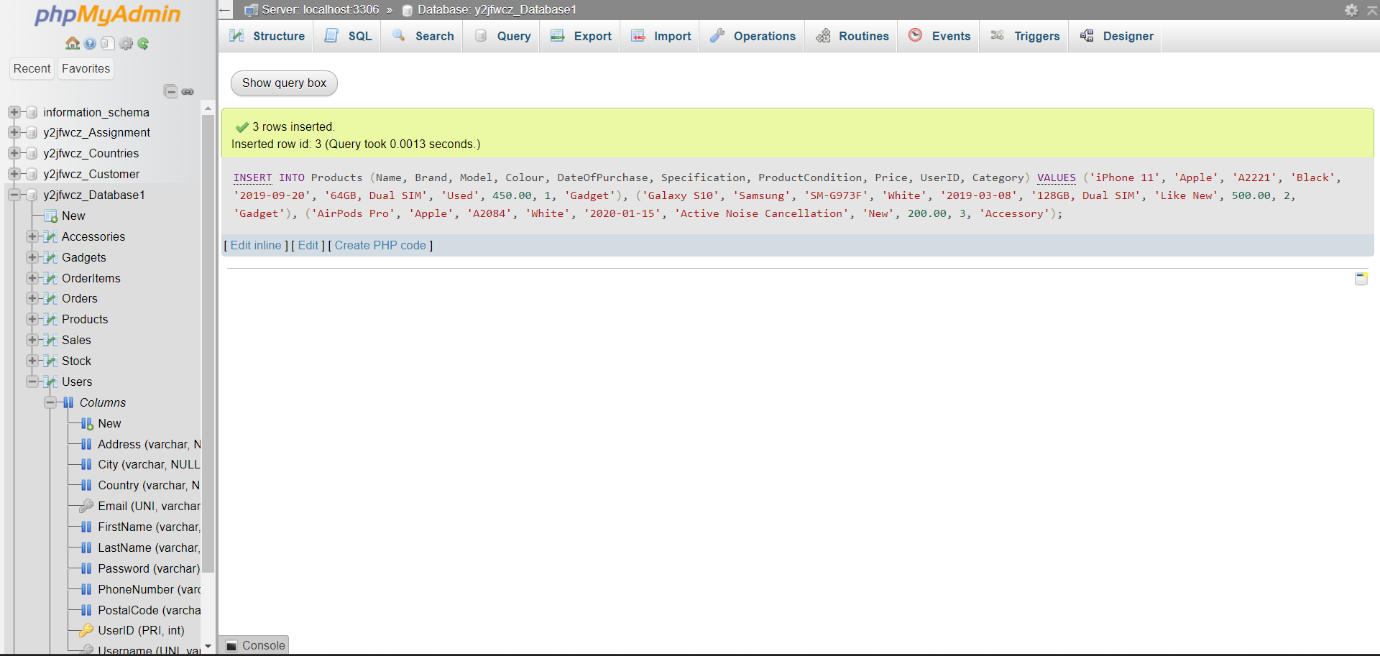


FIG 2.2

2.1 INSERTING DATA INTO GADGETS TABLE

INSERT INTO Gadgets (ProductID, BatteryLife, ScreenSize)

VALUES

(1, '10 hours', '6.1 inches'),

(2, '12 hours', '6.4 inches');

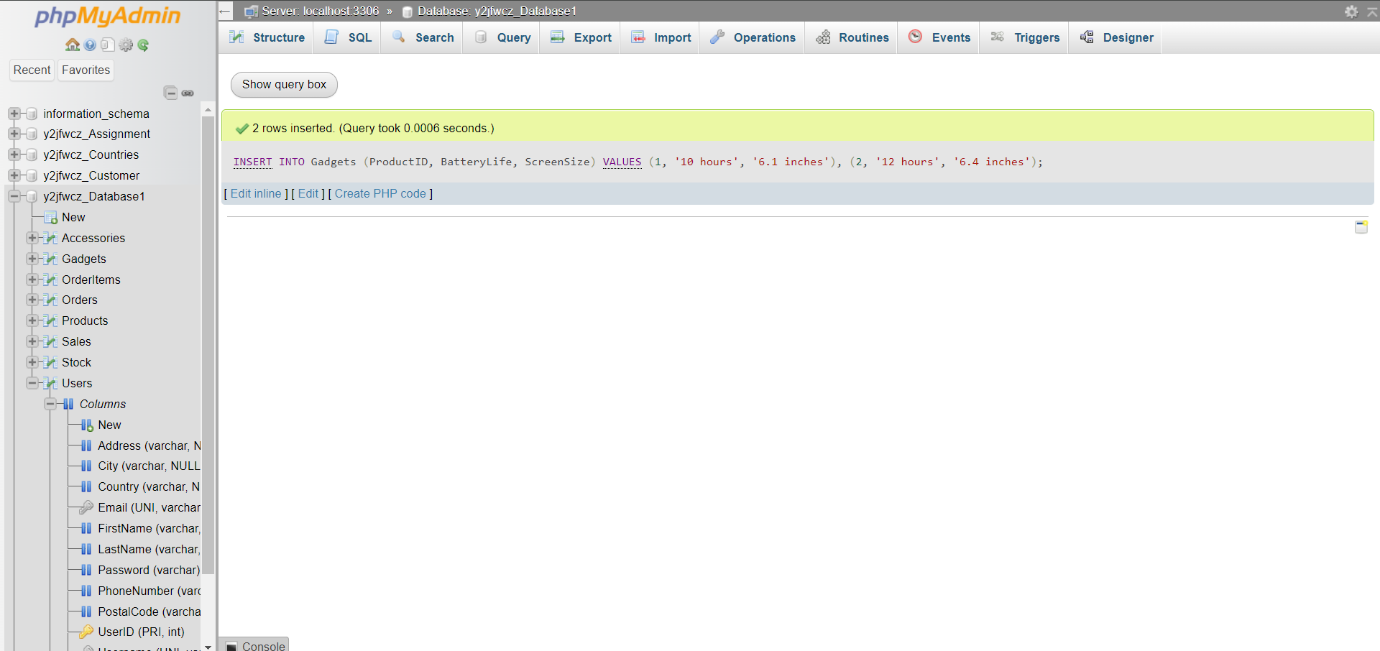


FIG 2.3

* 1. INSERTING DATA INTO ACCESSORIES

INSERT INTO Accessories (ProductID, Compatibility, Type)

VALUES

(3, 'iOS devices', 'Wireless Earbuds');

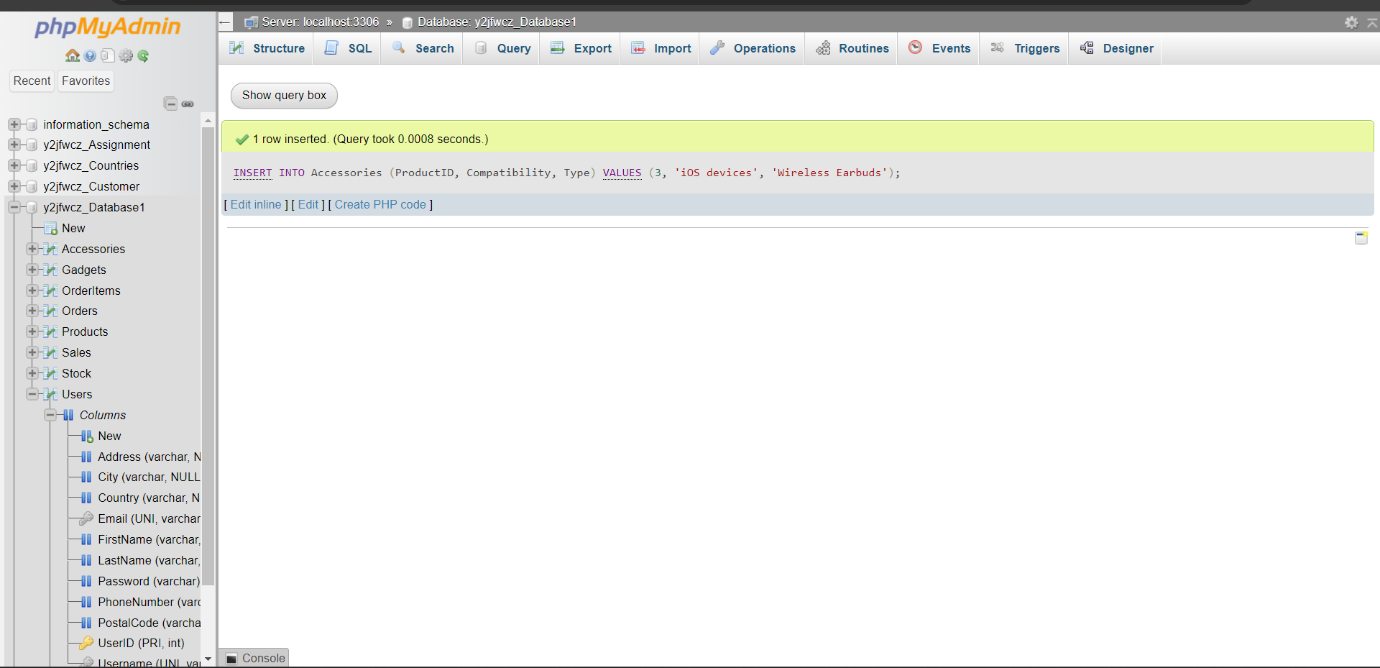


FIG 2.4

1. INSERTING DATA INTO ORDERS TABLE

INSERT INTO Orders (UserID, OrderDate, TotalPrice, ShippingAddress, ShippingCity, ShippingPostalCode, ShippingCountry)

VALUES

(1, '2024-06-01 10:00:00', 450.00, '123 Elm Street', 'London', 'E1 6AN', 'UK'),

(2, '2024-06-02 11:30:00', 500.00, '456 Oak Street', 'Manchester', 'M1 3AU', 'UK');

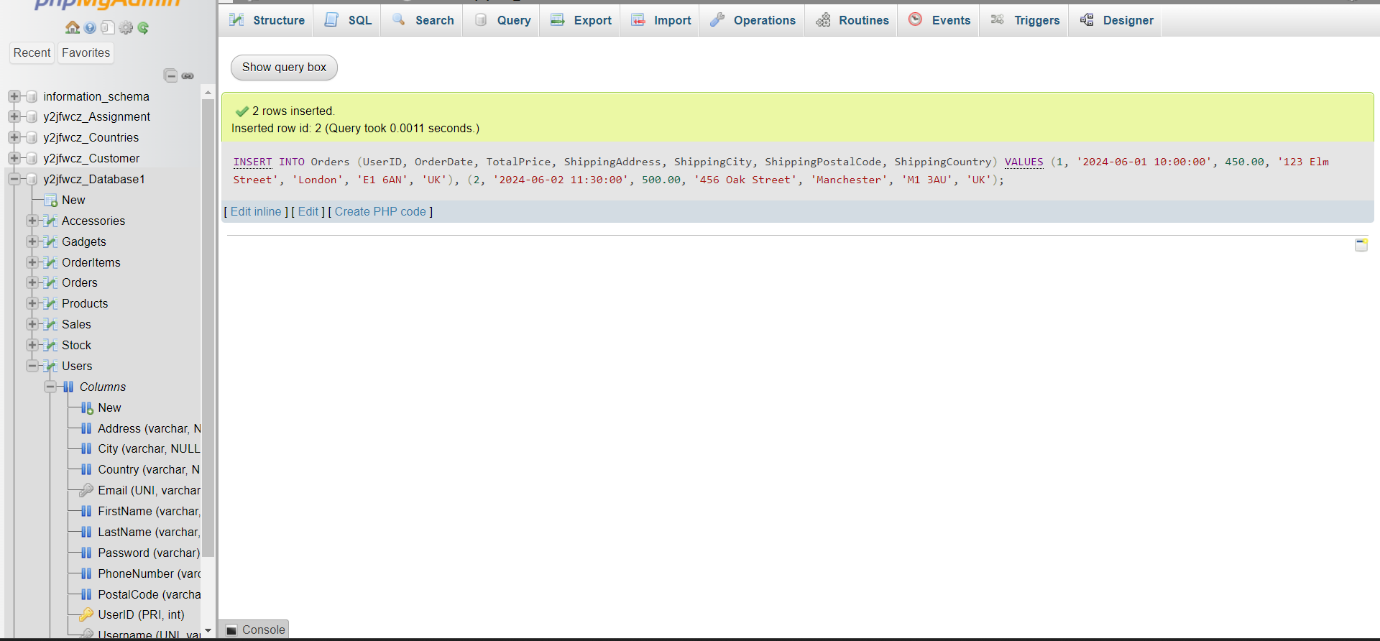


fig 2.5

1. INSERTING DATA INTO OrderItem TABLE

INSERT INTO OrderItems (OrderID, ProductID, Quantity, Price)

VALUES

(1, 1, 1, 450.00),

(2, 2, 1, 500.00);

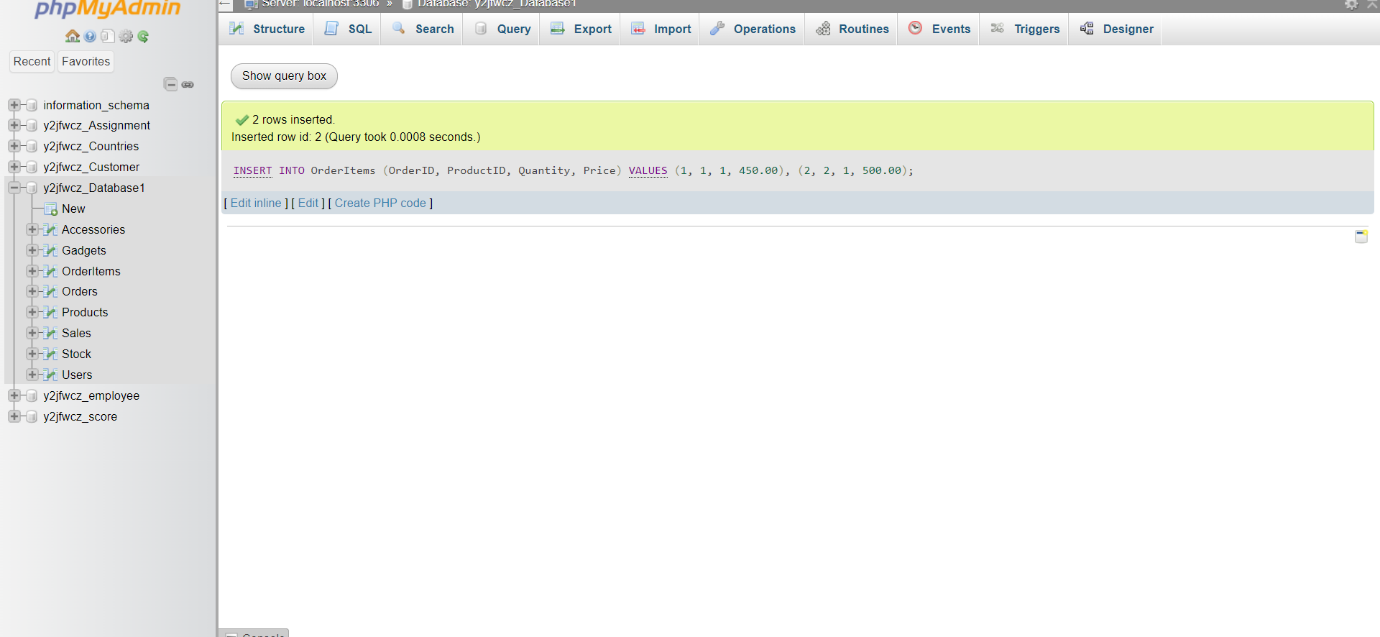


Fig 2.6

1. INSERT DATA INTO SALES TABLE

INSERT INTO Sales (ProductID, SaleDate, Quantity, SalePrice)

VALUES

(1, '2024-06-01 10:00:00', 1, 450.00),

(2, '2024-06-02 11:30:00', 1, 500.00);

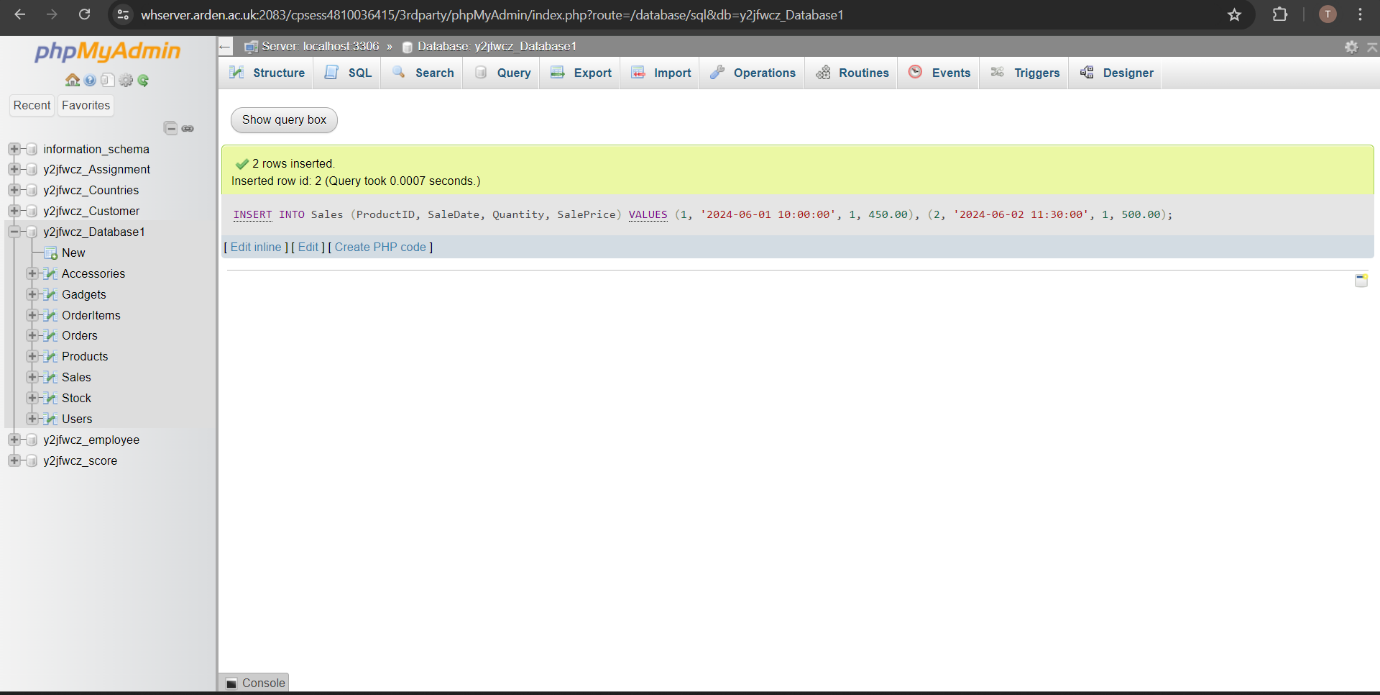


Fig 2.7

1. INSERT DATA INTO STOCK TABLE

INSERT INTO Stock (ProductID, Quantity, LastUpdated)

VALUES

(1, 5, '2024-06-01 10:00:00'),

(2, 3, '2024-06-02 11:30:00'),

(3, 10, '2024-06-03 14:45:00');

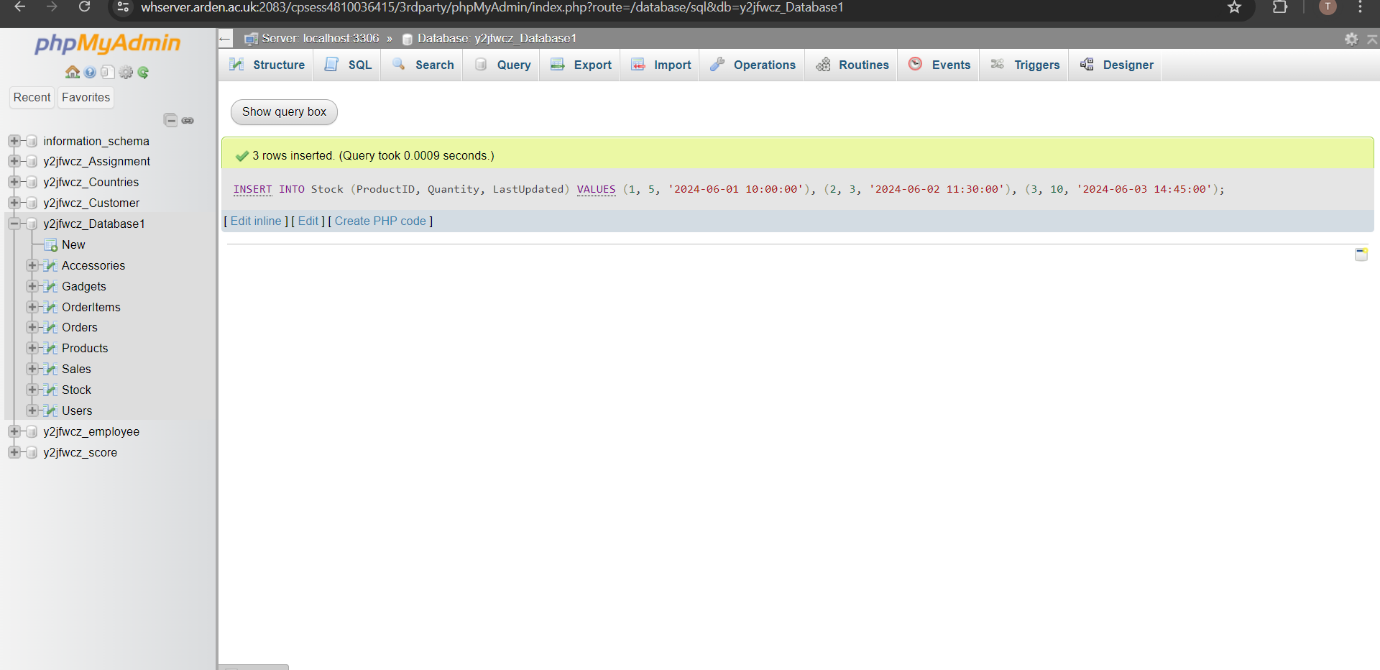


Fig 2.8

# CREATING VIEW

In SQL, views are a type of virtual table. Similar to tables, views also have rows and columns, however unlike tables, views do not save data on the disk. A view is a specially designed query that shows data as though it is from a single source while retrieving information from one or more tables.

VIEW-1

‘OrderSummery’

A summary of user orders is shown in this view, together with information on the users, the orders, and the total amount spent by each user. The platform administrators can monitor user behavior and identify important consumers with the aid of this view.

/\*CREATE VIEW OrderSummary AS

SELECT

Users.UserID,

Users.Username,

Users.Email,

COUNT(Orders.OrderID) AS TotalOrders,

SUM(Orders.TotalPrice) AS TotalAmount,

MAX(Orders.OrderDate) AS LastOrderDate

FROM Users

JOIN Orders

ON Users.UserID = Orders.UserID

GROUP BY

Users.UserID, Users.Username, Users.Email;\*/

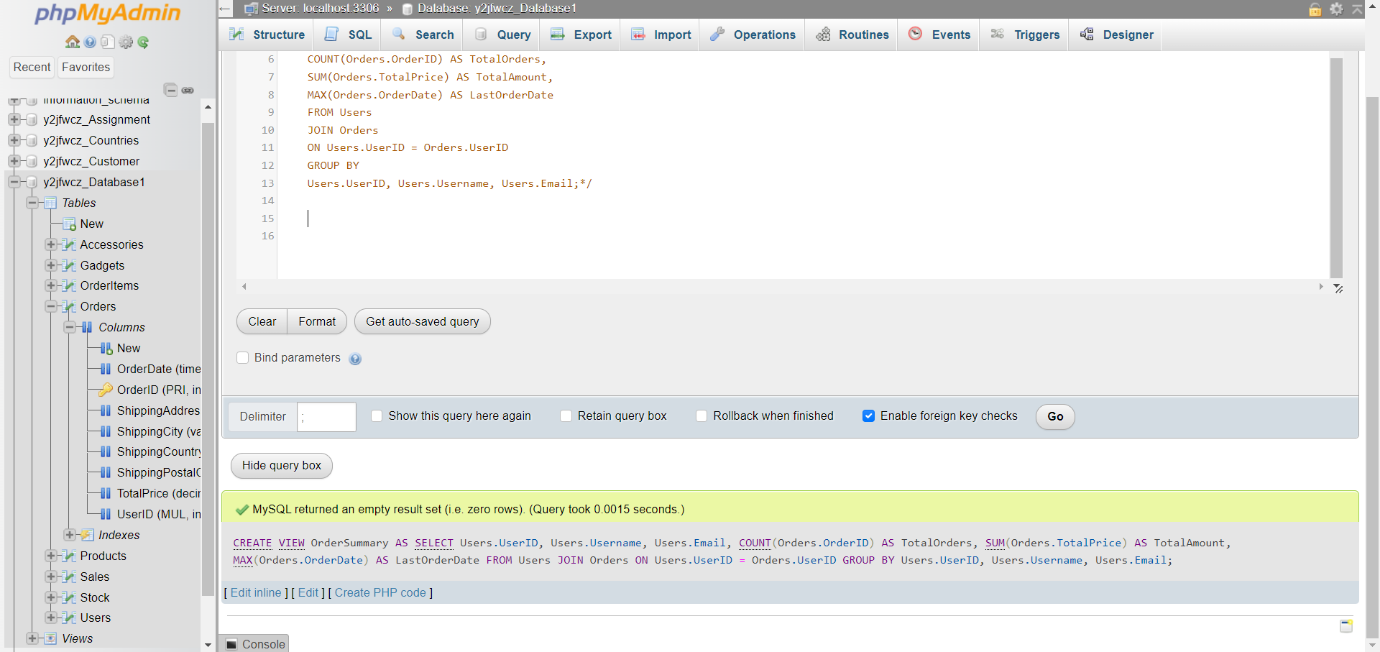


Fig 3.1

IMPLEMENTATION

We use, SELECT \* FROM OrderSummary;

Testing yields the results shown in the accompanying figure.

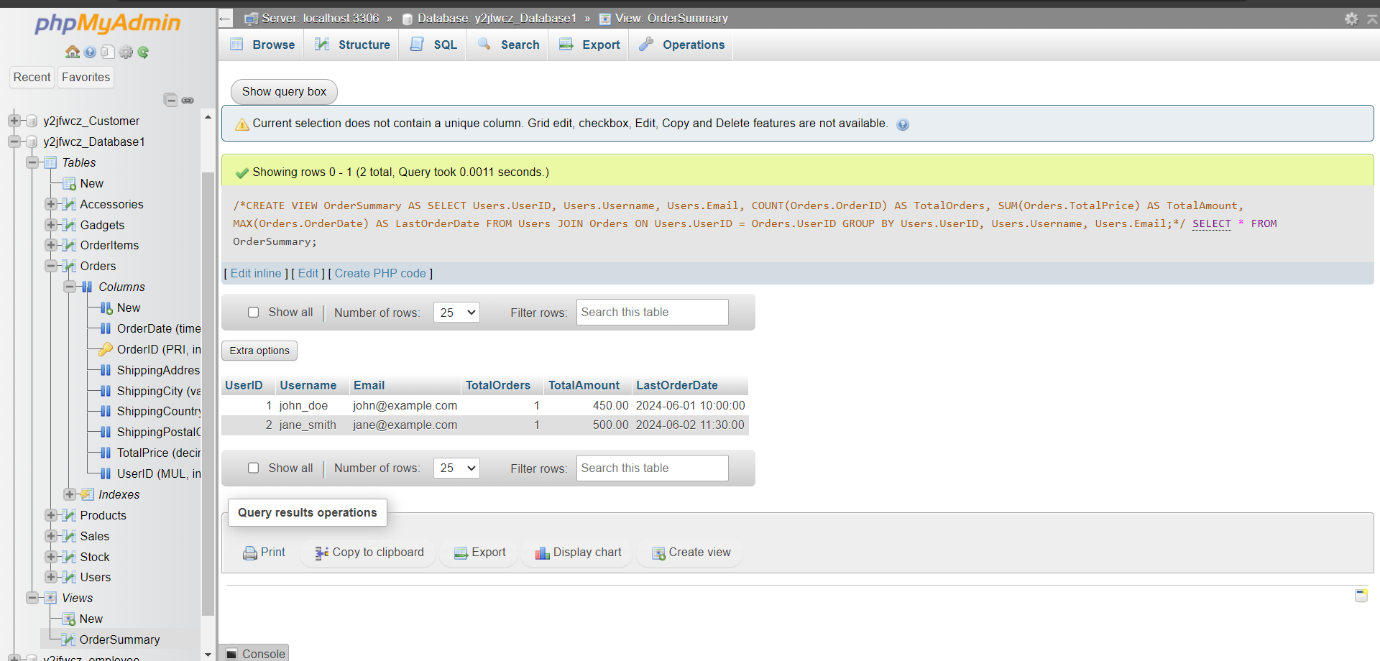


Fig 3.2

EXPLANATION

1. Joins: Information from the Users and Orders tables is combined using the JOIN clause. In particular, it connects rows from the Users table to rows from the Orders table when the UserIDs in the two tables match.
2. Grouping: The results are grouped by UserID, Username, and Email using the GROUP BY clause. This indicates that rows that share the same UserID, Username, and Email address are grouped together as a single unit. We must do this in order to compute aggregate values for every user user.Email, user.Name, and user.UserID: The groups are defined by these columns. Every group represents a distinct user.
3. Aggregations: To provide a summary of the data for every user, the view has multiple aggregate functions, including:

The total number of orders placed by each user is counted using the formula COUNT(order.OrderID) AS TotalOrders. Each user's total number of OrderID entries is counted using the COUNT function.

SUM(order.TotalPrice) AS TotalSpent: Adds up the total cost of each user's orders.

To determine the total amount spent by each user, use the SUM function.The most recent order date for each user is found using MAX(o.OrderDate) AS LastOrderDate. To obtain each user's most recent order date, the MAX function is utilized.

VIEW -2

‘ProductSalesPerformance’

For every product, this view offers comprehensive performance metrics, such as the quantity sold, total income, and average sale price. For managing inventories and sales, this is helpful.

CREATE VIEW ProductSalesPerformance AS

SELECT

Products.ProductID,

Products.Name,

Products.Brand,

Products.Category,

COUNT(Sales.SaleID) AS TotalSales,

SUM(Sales.SalePrice) AS TotalRevenue,

AVG(Sales.SalePrice) AS AverageSalePrice

FROM

Products

JOIN

Sales

ON Products.ProductID = Sales.ProductID

GROUP BY

Products.ProductID, Products.Name, Products.Brand, Products.Category;

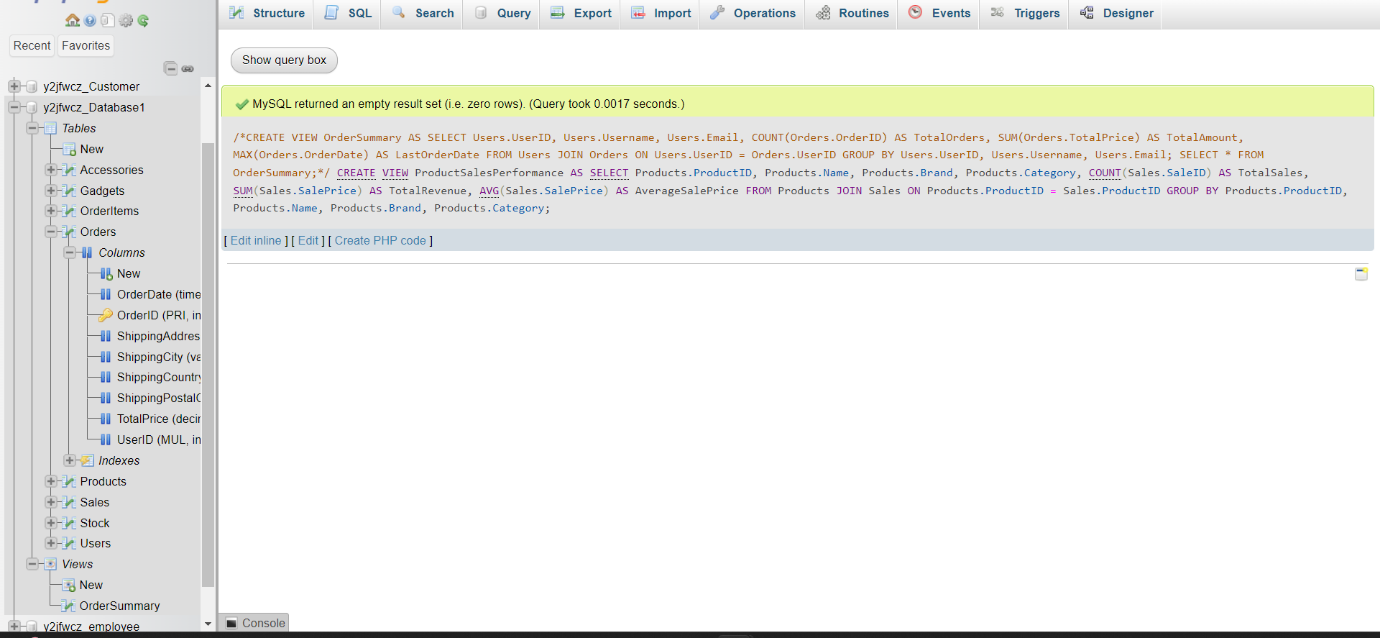


Fig 3.3

IMPLIMENTATION

We use, SELECT \* FROM ProductSalesPerformance;

Testing yields the results shown in the accompanying figure

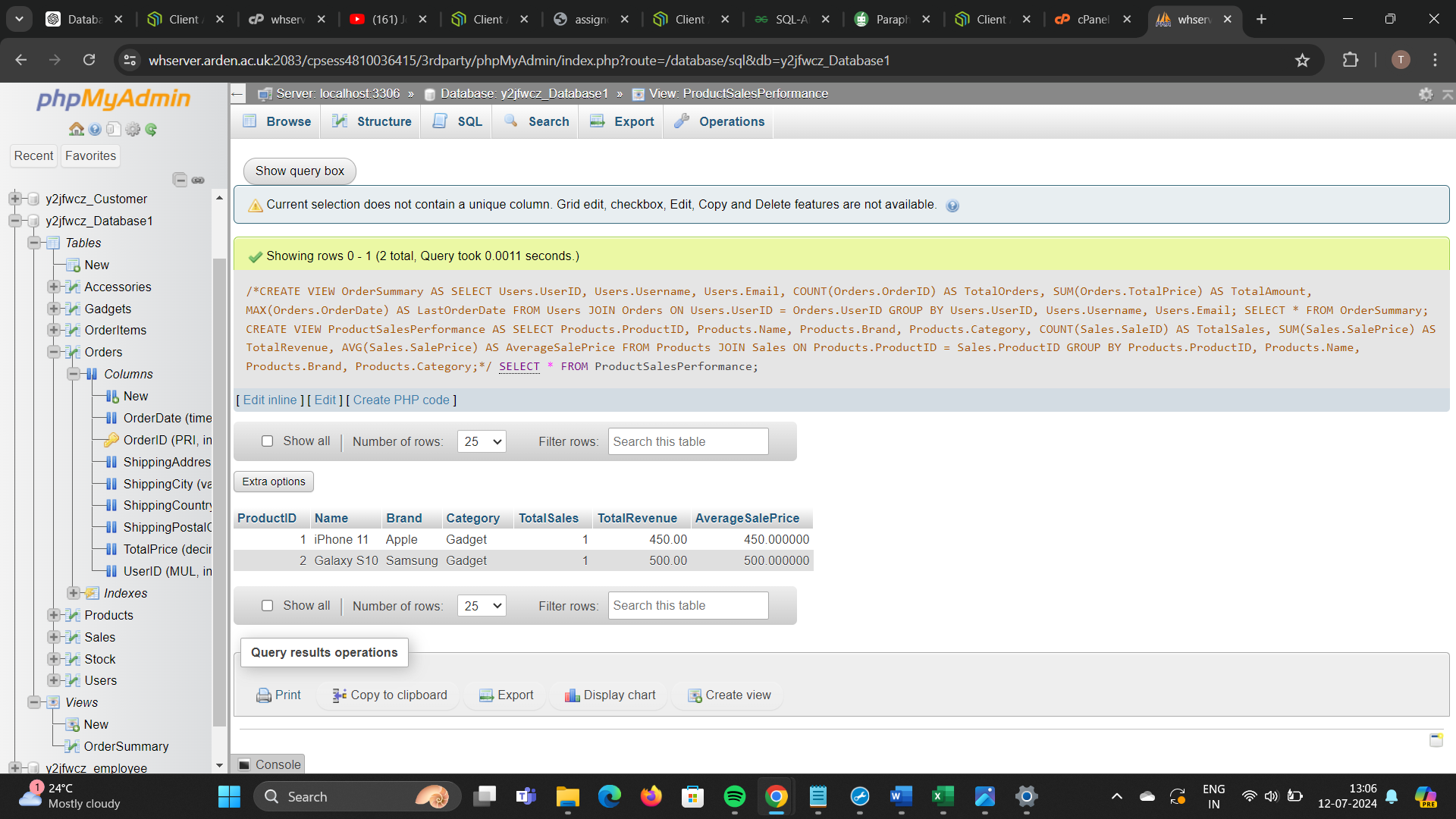


Fig 3.4

EXPLANATION

1. Joins: Information from the Sales and Products tables is combined using the JOIN clause. It specifically associates rows in the Sales table where the ProductID matches the ProductID in the Products table with rows in the Products table.
2. Grouping: The results are grouped by ProductID, Name, Brand, and Category using the GROUP BY clause. This indicates that a single group is formed from all rows that share the same ProductID, Name, Brand, and Category. We need to compute aggregate values for every product, therefore this is essential.

The columns products.ProductID, products.Name, products.Brand, and products.Category are utilized to define the groupings. Every group represents a distinct product.

1. Aggregations: To provide a summary of the data for every product, the view has multiple aggregate functions, including:

* COUNT(s.SaleID) AS TotalSales: This function counts the quantity of each product sold in total. To find the total number of SaleID entries for every product, utilize the COUNT function.
* SUM(s.SalePrice) AS TotalRevenue: Adds up all of the money made from every sale of every product. The entire sales revenue is computed using the SUM function.
* The average sale price for every product is determined using the formula AVG(s.SalePrice) AS AverageSalePrice. To find the average price at which each product was sold, using the AVG function.

# STORED PROCEDURES

Prepared SQL code that you may save and reuse repeatedly is called a stored procedure. To avoid writing the same SQL query twice, consider saving it as a stored procedure that can be called whenever needed.   
A stored procedure can also receive parameters, which allow it to take action based on the parameter value or values that are supplied.

1. ‘GetUserOrderSummery’

A summary of a user's orders, including the total number of orders, the total amount spent, and the date of the most recent order, is retrieved by this stored method.

/\*DELIMITER //

CREATE PROCEDURE GetUserOrderSummary(

IN pUserID INT

)

BEGIN

SELECT

Users.UserID,

Users.Username,

Users.Email,

COUNT(Orders.OrderID) AS TotalOrders,

SUM(Orders.TotalPrice) AS TotalSpent,

MAX(Orders.OrderDate) AS LastOrderDate

FROM

Users

JOIN

Orders

ON Users.UserID = Orders.UserID

WHERE

Users.UserID = pUserID

GROUP BY

Users.UserID, Users.Username, Users.Email;

END //

DELIMITER ;\*/

IMPLEMENTING

With the help of UserID 1, this command will obtain the order summary for the user, showing the total number of orders, the total amount spent, and the date of the most recent transaction.

CALL GetUserOrderSummary(1);

/\*DELIMITER //

CREATE PROCEDURE GetUserOrderSummary(

IN pUserID INT

)

BEGIN

SELECT

Users.UserID,

Users.Username,

Users.Email,

COUNT(Orders.OrderID) AS TotalOrders,

SUM(Orders.TotalPrice) AS TotalSpent,

MAX(Orders.OrderDate) AS LastOrderDate

FROM

Users

JOIN

Orders

ON Users.UserID = Orders.UserID

WHERE

Users.UserID = pUserID

GROUP BY

Users.UserID, Users.Username, Users.Email;

END //

IMPLEMENTATION

CALL GetUserOrderSummary(1);

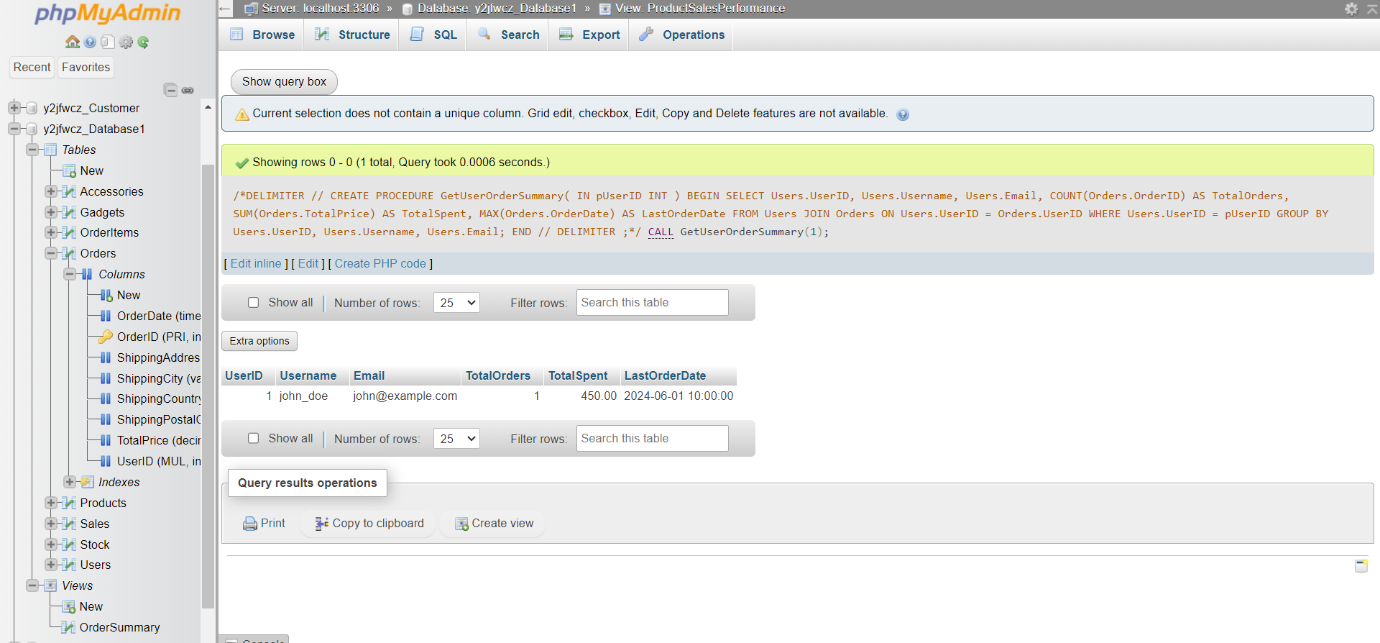


Fig 3.5

EXPLANATION

Purpose: The GetUserOrderSummary stored procedure's goal is to obtain an overview of a particular user's order history. It contains the total amount spent, the number of orders placed, and the date of the most recent order.

Input parameter: The UserID of the user for whom the order summary is to be retrieved is represented by the parameter pUserID INT.

Join and Aggregation: To get data from both tables, the process links the Users and Orders tables.

It determines the total number of orders, the total amount spent, and the date of the last order using aggregate methods like COUNT, SUM, and MAX.

Store Procedure: 2

GetProductUserDetail

/\*DELIMITER //

CREATE PROCEDURE GetProductUserDetails(

IN ProductID INT

)

BEGIN

SELECT

Products.ProductID,

Products.Name AS ProductName,

Products.Brand,

Products.Price,

Users.UserID,

Users.Username,

Users.Email

FROM

Products

JOIN

Users

ON Products.UserID = Users.UserID

WHERE

Products.ProductID = ProductID;

END //

DELIMITER ;\*/

IMPLIMENTATION

CALL GetProductUserDetails(1);

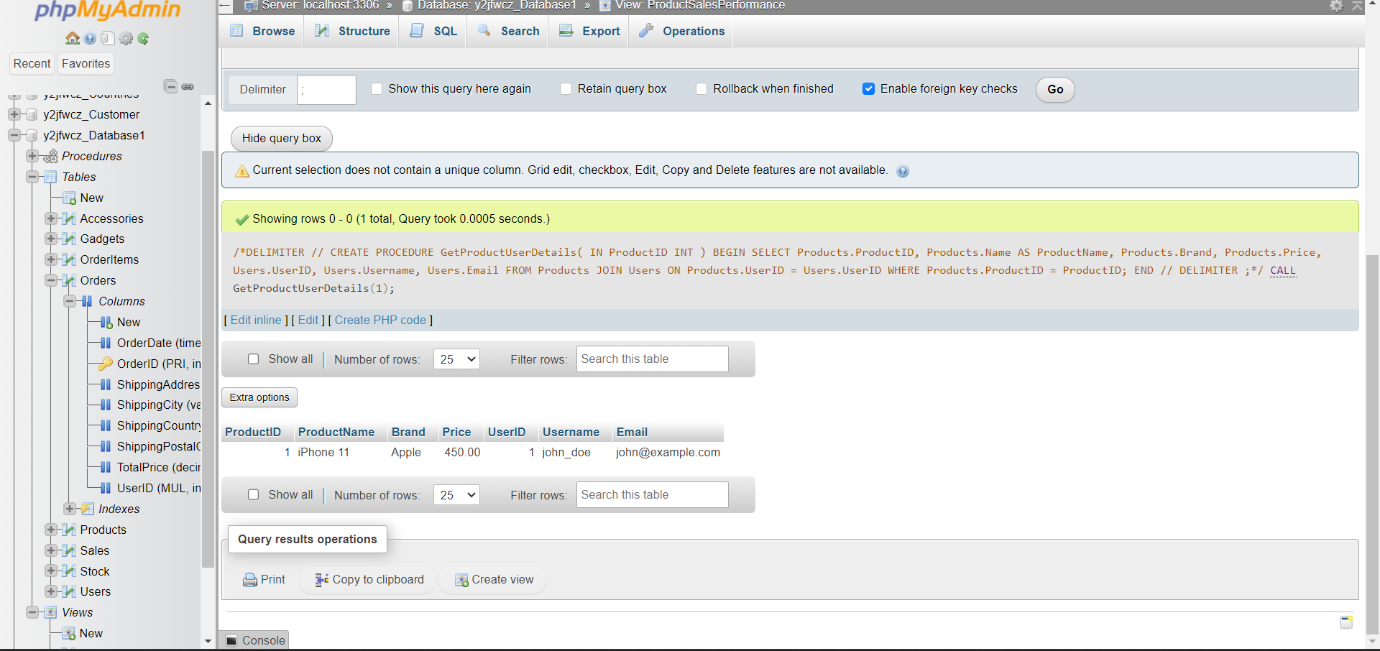


Fig 3.6

EXPLANATION

The GetProductUserDetails stored procedure's main objective is to obtain detailed product information and the identity of the person who posted the goods for sale.

JOIN : JOIN Users u ON products.UserID = user.UserID: This line joins the Users table to the Products table based on the UserID:

ON products.UserID = user.UserID: This condition specifies that the UserID in the Products table must match the UserID in the Users table to include the records.

Purpose of the Join

Purpose of JOIN: Get not just the product information but also the user information (email address and username) related to that product. This thorough information gives the information being retrieved context and aids in determining who listed the commodity.

# TRIGGERS

A trigger is a database stored procedure that automatically fires whenever an event (insert, update,... For example, you can have a trigger to fire when a row is added for certain table or if specific columns in the tables are changed. In simple words a trigger is essentially, set of named SQL statements stored in memory. It belongs to a distinct type of stored procedures that are automatically called by database server in

response due to some event. Every trigger is bound to a table

STEP 1: ALTER TABLE ‘Products’

ALTER TABLE Products

ADD StockQuantity INT DEFAULT 0,

ADD Status VARCHAR(20) DEFAULT 'In Stock';

STEP 2: CREATE TRIGGER

DELIMITER //

CREATE TRIGGER UpdateProductStatus

BEFORE UPDATE ON Products

FOR EACH ROW

BEGIN

IF NEW.StockQuantity < 5 THEN

SET NEW.Status = 'Out of Stock';

ELSE

SET NEW.Status = 'In Stock';

END IF;

END //

DELIMITER ;

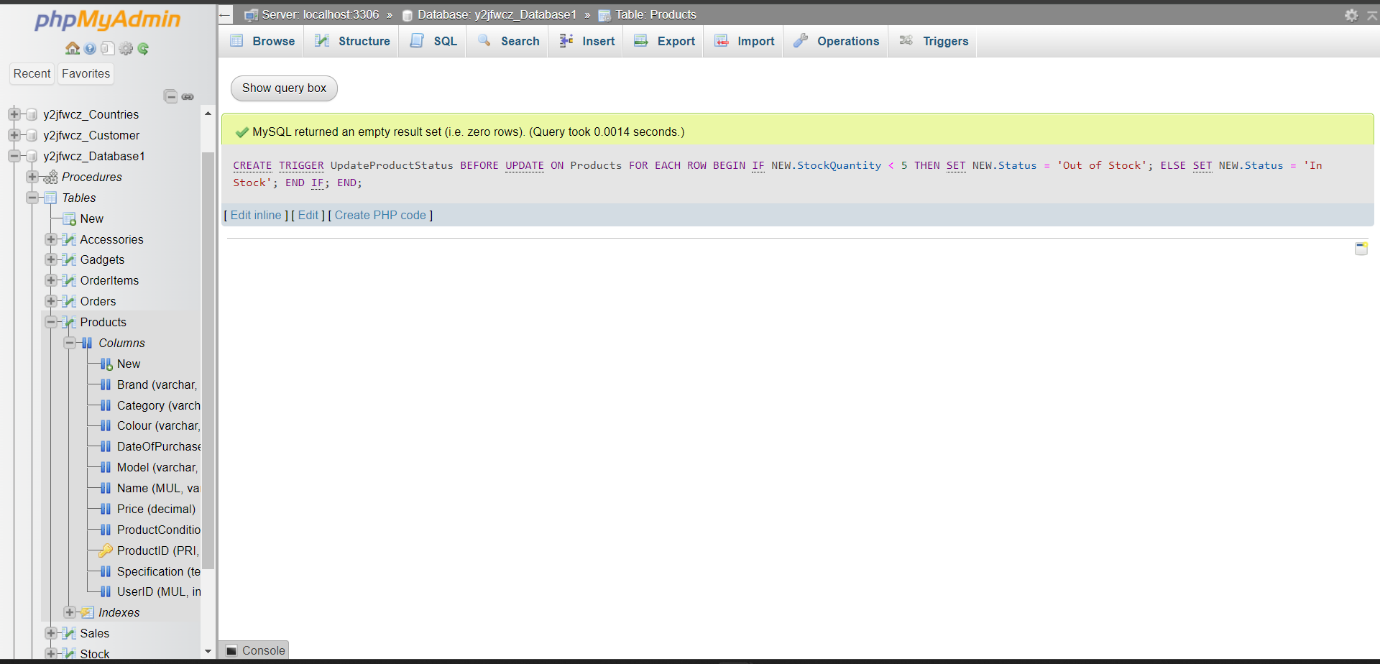


FIG 3.7

STEP 3: UPDATE Products SET StockQuantity = 3 WHERE ProductID = 1;

STEP4: OUTPUT

SELECT \* FROM Products WHERE ProductID = 1;

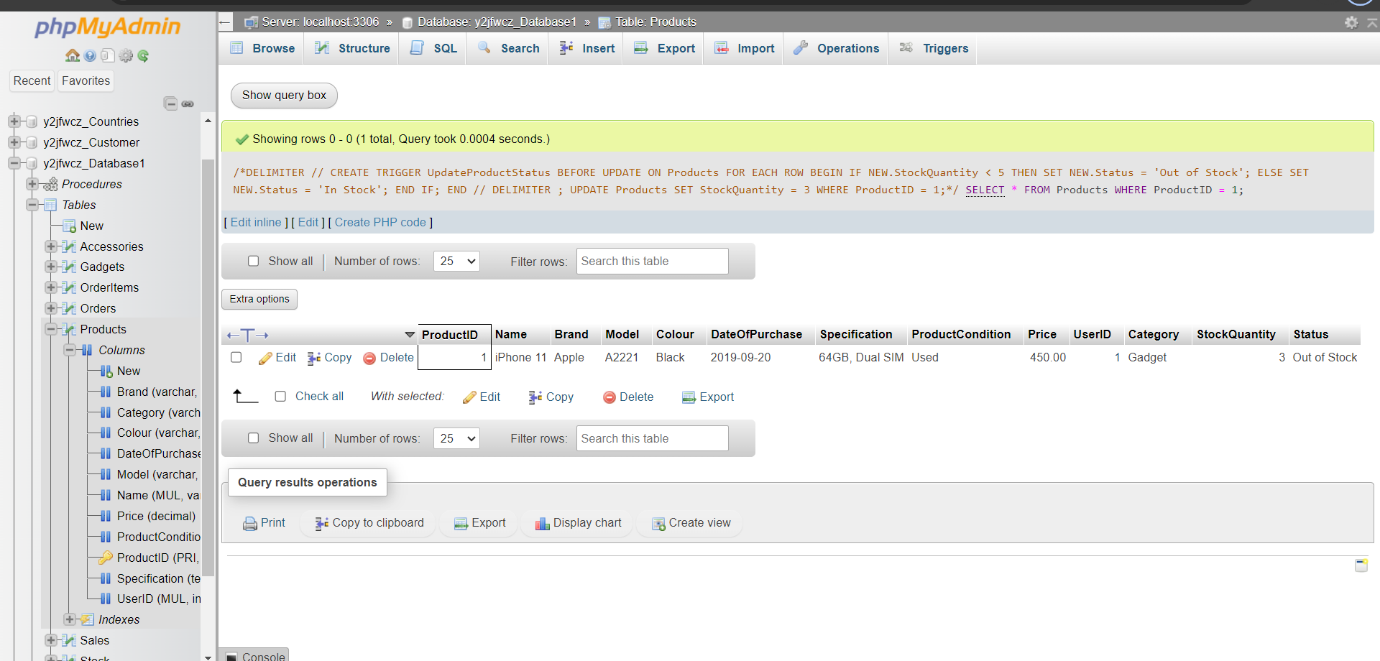


FIG 3.8

EXPLANATION

CREATE TRIGGER

SYNTAX: ‘CREATE TRIGGER UpdateProductStatus’

PURPOSE: A new trigger called UpdateProductStatus is defined on this line. Naming standards usually contain the item being modified (e.g., ProductStatus) and the action being done (e.g., Update).

IF statement

PURPOSE: The NEW.StockQuantity, or the stock quantity that is about to be updated, is evaluated by this conditional logic.

It sets NEW if the new stock amount is less than five.Marked as "Out of Stock."

It sets NEW otherwise."In Stock" is the status.

# REFERENCES

Mullins, C. (2012). PostgreSQL 9.0 High Performance. Packt Publishing Ltd.

Ponniah, P. (2010). Database Design and Development: An Essential Guide for IT Professionals. John Wiley & Sons.

Stonebraker, M. (2018). Readings