

# **Case Study: Database Design and Development for E-commerce Platform (SneakerHead)**

**Name: Dhavalkumar Parmar**

**Course: Data Concepts**

**Enrolment No.: 000965569**



## **Table of Contents**

- 1. Introduction**
- 2. Mission**
- 3. Objectives**
- 4. Database Structure & Tables**
  - a. Tables and Fields**
  - b. Relationship**
- 5. Entity-Relationship Diagram (ERD)**
- 6. Conclusion**

## Introduction

Sneakerhead is an innovative e-commerce platform specializing in personalized, sustainable sneakers. The platform aims to enhance customer experiences through seamless online shopping while leveraging data-driven insights to optimize operations and product offerings. This report outlines the database design that supports Sneakerhead's business objectives.

## Mission

To use data insights to develop sustainable, high-quality footwear, streamline business operations, and enhance customer experiences.

## Objectives:

- Analyse customer data to develop products aligned with their preferences.
- Optimize inventory management and supply chain using data insights.
- Utilize customer feedback to enhance product satisfaction and engagement.

## Database Structure & Tables

The Sneakerhead database consists of multiple relational tables that efficiently store and manage data related to users, orders, products, suppliers, payments, and shipping.

### List of Tables:

#### 1. User & Order Management Tables:

- Users Table
- Orders Table
- Feedback Table

#### 2. Product & Supplier Management Tables:

- Products Table
- Product Categories Table
- Suppliers Table

#### 3. Transaction & Shipping Tables:

- Payments Method Table
- Shipping Information Table

**Users Table:**

- USER\_ID: Unique identifier for each user.
- FIRSTNAME & NAME: User's name details.
- EMAIL & PHONE: Contact details.
- ADDRESS: Shipping and residential address.

USERS	
USER_ID	INTEGER(20)
FIRSTNAME	VARCHAR(30)
Name	VARCHAR(100)
Email	VARCHAR(100)
Phone	VARCHAR(15)
Address	VARCHAR(250)

**Orders Table:**

- ORDER\_ID: Unique order identifier.
- USER\_ID: Links orders to users.
- ORDER DATE: Date of order placement.
- TOTAL AMOUNT: Total order cost.
- PAYMENT STATUS: Status of the payment.

ORDERS	
Order_ID	INTEGER(20)
User_ID	INT(20)
OrderDate	DATE
TotalAmount	DECIMAL
PaymentStatus	VARCHAR(50)

**Products Table:**

- PRODUCT\_ID: Unique identifier.
- NAME: Product name.
- CATEGORY\_ID: Links products to categories.
- ORDER\_ID & SUPPLIER\_ID: Links products to orders and suppliers.
- PRICE, MATERIAL, SIZE, COLOR, STOCK QUANTITY, LAUNCH DATE: Product details.

PRODUCTS	
Product_ID	INTEGER(20)
Name	VARCHAR(100)
Category_ID	INT(20)
Order_ID	INT(20)
Supplier_ID	INTEGER(20)
Price	DECIMAL
Material	VARCHAR(100)
Size	VARCHAR(10)
Color	VARCHAR(50)
StockQuantity	INT(20)
LaunchDate	DATE

**Product Categories Table:**

- CATEGORY\_ID: Unique category identifier.
- NAME & DESCRIPTION: Category details.

ProductCategories	
Category_ID	INTEGER(20)
Name	VARCHAR(100)
Description	VARCHAR(250)

### Suppliers Table:

- SUPPLIER\_ID: Unique supplier identifier.
- NAME, CONTACT PERSON, PHONE, EMAIL, ADDRESS: Supplier details.

Suppliers	
Supplier_ID	INTEGER(20)
Name	VARCHAR(100)
ContactPerson	VARCHAR(100)
Phone	VARCHAR(15)
Email	VARCHAR(100)
Address	VARCHAR(250)

### Payments Table:

- PAYMENT\_ID: Unique identifier.
- ORDER\_ID: Links payments to orders.
- PAYMENT DATE, METHOD, STATUS: Payment details.

Paymentmethod	
Payment_ID	integer(20)
OrderID	INT
PaymentDate	DATE
PaymentMethod	VARCHAR(50)
PaymentStatus	VARCHAR(50)

### Shipping Information Table:

- SHIPPING\_ID: Unique identifier.
- ORDER\_ID: Links shipping info to orders.
- SHIPPING ADDRESS, METHOD, DATES: Shipping details.

ShippingInformation	
Shipping_ID	INTEGER(20)
Order_ID	INT(20)
ShippingAddress	VARCHAR(250)
ShippingMethod	VARCHAR(100)
ShippingDate	DATE
DeliveryDate	DATE

**Feedback Table:**

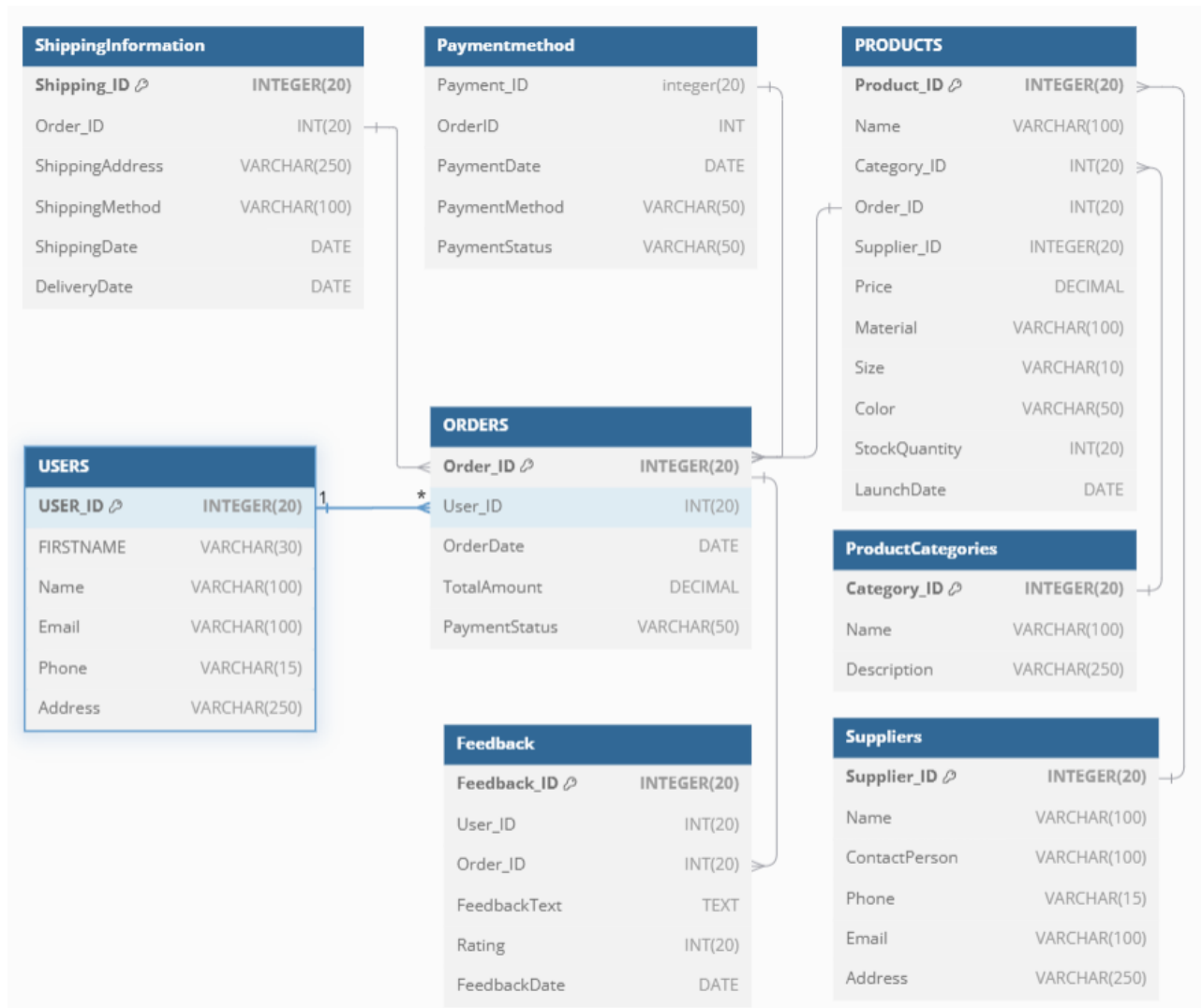
- FEEDBACK\_ID: Unique identifier.
- USER\_ID & ORDER\_ID: Links feedback to users and orders.
- FEEDBACK TEXT, RATING, FEEDBACK DATE: Customer reviews.

Feedback	
Feedback_ID 🔗	INTEGER(20)
User_ID	INT(20)
Order_ID	INT(20) ➤
FeedbackText	TEXT
Rating	INT(20)
FeedbackDate	DATE



## Entity Relationship Diagram

Entity-Relationship (ER) Diagram the ER diagram illustrates relationships between different database tables, ensuring a structured and efficient database schema for Sneakerhead's operations.



E-R Diagram

## Join Types in Database

### 1. INNER JOIN:

This join returns only the records that have matching values in both tables. For example, if we join the Users and Orders tables using an inner join, we will get a list of users who have placed at least one order. Users who have never placed an order will not be included in the result.

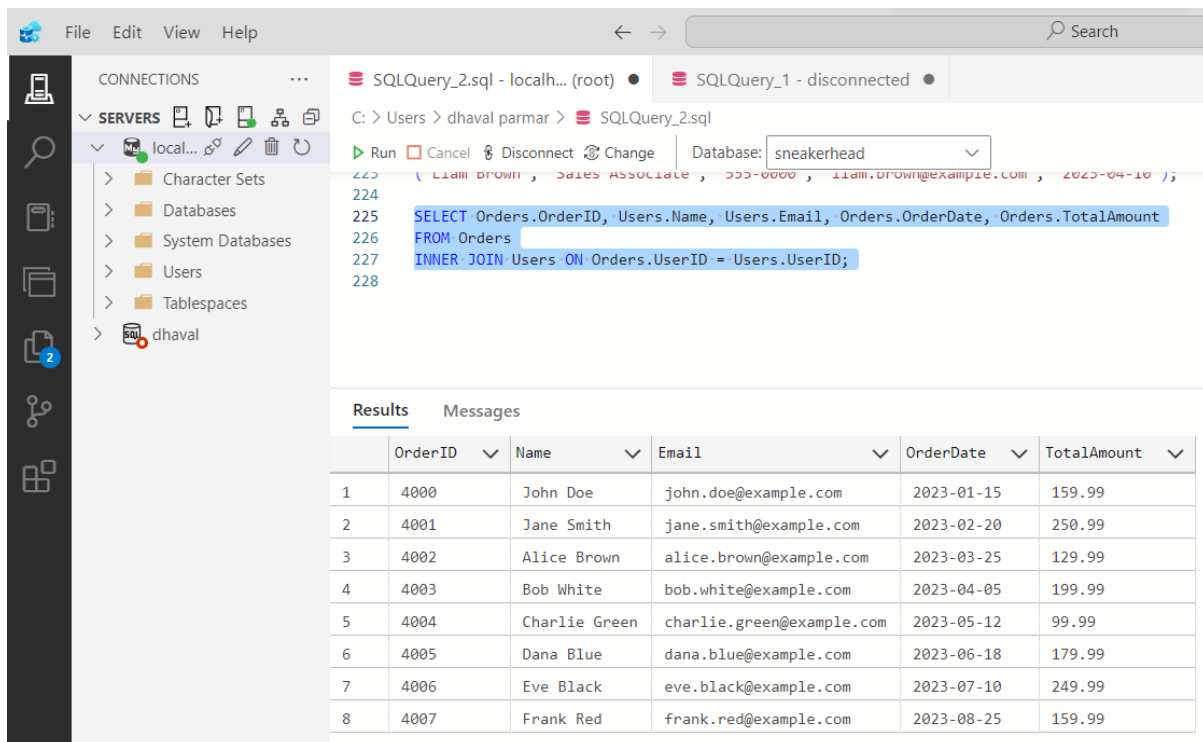
#### Query:

```
SELECT Orders.OrderID, Users.Name, Users.Email, Orders.OrderDate, Orders.TotalAmount
```

```
FROM Orders
```

```
INNER JOIN Users ON Orders.UserID = Users.UserID;
```

#### Output:



The screenshot shows the SQL Server Enterprise Manager interface. The left pane displays the 'SERVERS' tree with 'local...' expanded, showing 'Character Sets', 'Databases', 'System Databases', 'Users', 'Tablespaces', and 'dhaval'. The right pane shows the 'SQLQuery\_2.sql' file with the following query:

```

223  ( Liam Brown , Sales Associate , 555-0000 , liam.brown@example.com , 2023-04-10 );
224
225  SELECT Orders.OrderID, Users.Name, Users.Email, Orders.OrderDate, Orders.TotalAmount
226  FROM Orders
227  INNER JOIN Users ON Orders.UserID = Users.UserID;
228

```

The 'Results' tab is active, displaying the following data:

	OrderID	Name	Email	OrderDate	TotalAmount
1	4000	John Doe	john.doe@example.com	2023-01-15	159.99
2	4001	Jane Smith	jane.smith@example.com	2023-02-20	250.99
3	4002	Alice Brown	alice.brown@example.com	2023-03-25	129.99
4	4003	Bob White	bob.white@example.com	2023-04-05	199.99
5	4004	Charlie Green	charlie.green@example.com	2023-05-12	99.99
6	4005	Dana Blue	dana.blue@example.com	2023-06-18	179.99
7	4006	Eve Black	eve.black@example.com	2023-07-10	249.99
8	4007	Frank Red	frank.red@example.com	2023-08-25	159.99

#### Purpose:

Retrieves only matching records from both tables. If no match is found, the row is excluded.

#### Use Cases:

- Getting customers who have placed orders.
- Finding employees who belong to a department.
- Fetching products that have categories assigned.

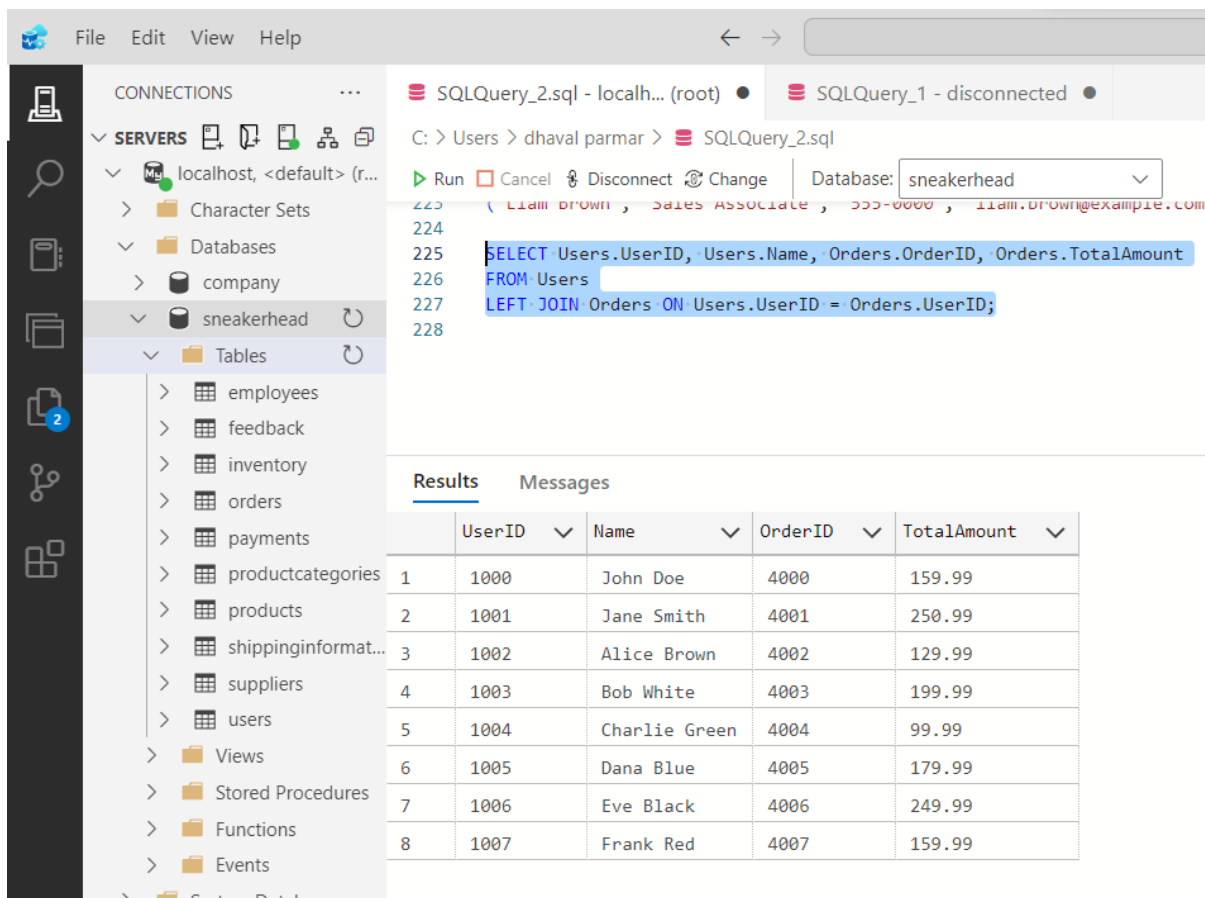
## 2. LEFT JOIN (or LEFT OUTER JOIN)

This join returns all records from the left table and only the matching records from the right table. If there are no matches, NULL values are displayed for columns from the right table. For example, if we join Users with Orders using a left outer join, we will get all users, including those who haven't placed an order. Users without orders will have NULL values in the order-related columns.

### Query:

```
SELECT Users.UserID, Users.Name, Orders.OrderID, Orders.TotalAmount
FROM Users
LEFT JOIN Orders ON Users.UserID = Orders.UserID;
```

### Output:



The screenshot shows the SQL Server Enterprise Manager interface. On the left, the 'Servers' tree is expanded to 'localhost, <default> (r...)' > 'Databases' > 'sneakerhead' > 'Tables'. The 'Tables' folder is expanded, showing a list of tables including employees, feedback, inventory, orders, payments, productcategories, products, shippinginformat..., suppliers, users, Views, Stored Procedures, Functions, and Events. The 'Users' table is selected. In the center, the 'SQLQuery\_2.sql' file is open, showing the following query:

```
SELECT Users.UserID, Users.Name, Orders.OrderID, Orders.TotalAmount
FROM Users
LEFT JOIN Orders ON Users.UserID = Orders.UserID;
```

The 'Database' dropdown is set to 'sneakerhead'. Below the query editor, the 'Results' tab is active, displaying the following data:

	UserID	Name	OrderID	TotalAmount
1	1000	John Doe	4000	159.99
2	1001	Jane Smith	4001	250.99
3	1002	Alice Brown	4002	129.99
4	1003	Bob White	4003	199.99
5	1004	Charlie Green	4004	99.99
6	1005	Dana Blue	4005	179.99
7	1006	Eve Black	4006	249.99
8	1007	Frank Red	4007	159.99

### Purpose:

Returns all records from the left table and matching records from the right table. If no match is found, NULL is returned.

### Use Cases:

- Finding users who haven't placed orders.
- Listing all products, even if they have no supplier.
- Getting employees, even if they are not assigned to a department.

### 3. RIGHT JOIN (or RIGHT OUTER JOIN)

This join returns all records from the right table and only the matching records from the left table. If there are no matches, NULL values are displayed for columns from the left table. For example, if we join Orders with Users using a right outer join, we will get all orders, including those that do not have associated users (e.g., orders placed by deleted accounts). Orders without users will have NULL values in the user-related columns.

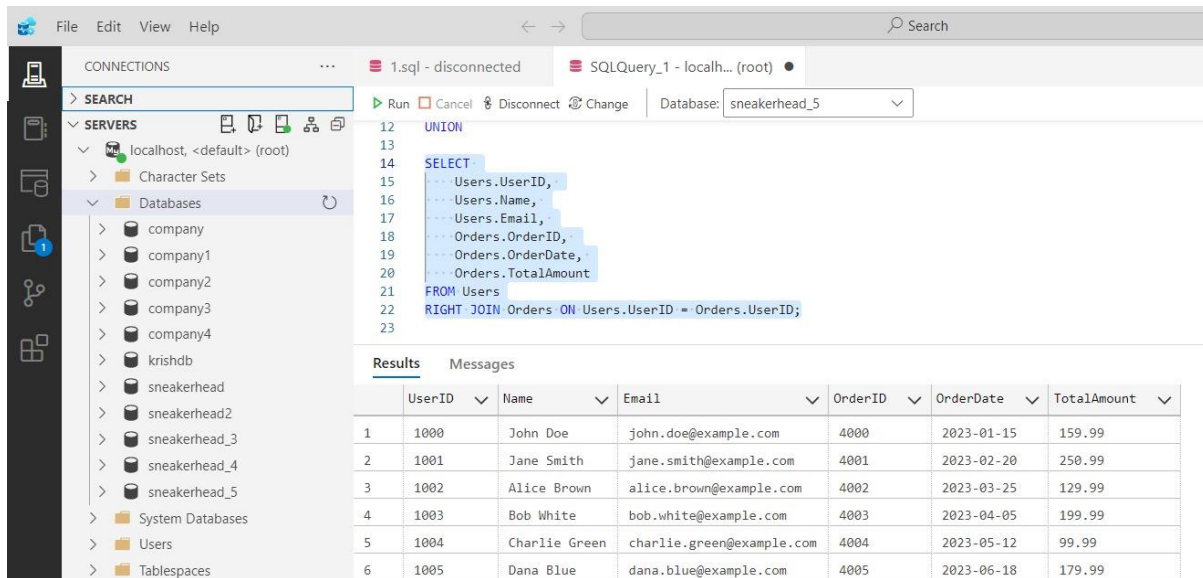
#### Query:

```
SELECT Suppliers.SupplierID, Suppliers.Name, Products.Name AS ProductName
```

```
FROM Suppliers
```

```
RIGHT JOIN Products ON Suppliers.SupplierID = Products.SupplierID;
```

#### Output:



The screenshot shows the SQL Server Enterprise Manager interface. On the left, the 'SERVERS' tree is expanded to 'Databases', showing a list of databases including 'sneakerhead\_5'. The central pane displays a SQL query in a window titled 'SQLQuery\_1 - localh... (root)'. The query is as follows:

```

12 UNION
13
14 SELECT
15     Users.UserID,
16     Users.Name,
17     Users.Email,
18     Orders.OrderID,
19     Orders.OrderDate,
20     Orders.TotalAmount
21 FROM Users
22 RIGHT JOIN Orders ON Users.UserID = Orders.UserID;
23

```

Below the query editor, the 'Results' tab is active, displaying a table with 6 rows and 7 columns. The columns are: UserID, Name, Email, OrderID, OrderDate, and TotalAmount. The data is as follows:

	UserID	Name	Email	OrderID	OrderDate	TotalAmount
1	1000	John Doe	john.doe@example.com	4000	2023-01-15	159.99
2	1001	Jane Smith	jane.smith@example.com	4001	2023-02-20	250.99
3	1002	Alice Brown	alice.brown@example.com	4002	2023-03-25	129.99
4	1003	Bob White	bob.white@example.com	4003	2023-04-05	199.99
5	1004	Charlie Green	charlie.green@example.com	4004	2023-05-12	99.99
6	1005	Dana Blue	dana.blue@example.com	4005	2023-06-18	179.99

#### Purpose:

Returns all records from the right table and matching records from the left table. If no match is found, NULL is returned.

#### Use Cases:

- Listing all suppliers, even those who haven't supplied any products.
- Finding departments that don't have any employees assigned.
- Getting all orders, even if they don't have a registered user.

## 4. CROSS JOIN

This join returns the Cartesian product of both tables, meaning every row from the first table is paired with every row from the second table. For example, if we perform a cross join between Users and Products, the result will show every user associated with every product, even if they have never purchased it. This can be useful for generating all possible combinations for recommendations or promotions.

### Query:

```
SELECT Products.Name AS ProductName, Suppliers.Name AS SupplierName
```

```
FROM Products
```

```
CROSS JOIN Suppliers;
```

### Output:

The screenshot shows the SQL Server Enterprise Manager interface. The left pane displays the 'Servers' tree with 'localhost' expanded, showing 'Databases' and 'Tables'. The 'products' table is selected. The right pane shows the SQL query editor with the following code:

```
224
225 SELECT Products.Name AS ProductName, Suppliers.Name AS SupplierName
226 FROM Products
227 CROSS JOIN Suppliers;
228
229
230 INSERT INTO Suppliers (SupplierID, Name)
231 VALUES
232 (1, 'ABC Supplies'),
233 (2, 'Global Traders');
```

Below the query editor, the 'Results' tab is active, displaying a table with 12 rows and 3 columns: 'ProductID', 'ProductName', and 'SupplierName'.

ProductID	ProductName	SupplierName
1	Heavy Duty Work Boot	ABC Supplies
2	Speed Runner	ABC Supplies
3	Casual Loafer	ABC Supplies
4	Elegant Stiletto	ABC Supplies
5	Comfort Step	ABC Supplies
6	Desert Walker	ABC Supplies
7	Mountain Explorer	ABC Supplies
8	Air Runner	ABC Supplies
9	Heavy Duty Work Boot	Global Traders
10	Speed Runner	Global Traders
11	Casual Loafer	Global Traders
12	Elegant Stiletto	Global Traders

### Purpose:

Creates a Cartesian product (all possible combinations of rows).

### Use Case:

- Listing all possible combinations of products and suppliers.
- Generating all possible matchups between players in a tournament.
- Creating pricing models for all combinations of services and discounts.

## Conclusion

The database design for SneakerHead has been developed with the goal of providing an efficient and structured approach to manage all aspects of the e-commerce platform. By implementing relational tables for users, orders, products, suppliers, payments, shipping, and feedback, the design ensures seamless data flow and accurate record-keeping, allowing the platform to meet its mission of delivering personalized, sustainable sneakers. The use of various SQL join types enhances data retrieval flexibility, enabling the platform to optimize operations, customer experiences, and decision-making. This database design serves as the backbone for SneakerHead's data-driven strategies, providing insights that support inventory management, customer satisfaction, and overall business growth.