

Case Study

Database Design & Development for E-commerce Platform (Sneakerhead)

Name: Krish Patel

Course: Data concepts

Enrollment No: 000965430



Table of Contents:-

1. Introduction
2. Mission
3. Objectives
4. Database Design
 - a. Tables and Fields
 - b. Relationship
5. Entity-Relationship Diagram (ERD)
6. Conclusion
7. Appendix
 - a. SQL Test Database

Introduction:-

Sneaker Head is an innovative e-commerce platform specializing in personalized, sustainable sneakers. We offer a curated collection of high quality footwear, enhancing customer experiences and seamless online shopping.

Mission:

To use data-driven insights to create sustainable, high-quality footwear, streamline operations, and deliver outstanding customer experiences.

Objective:

- Analyze customer data to develop products that align with their needs.
- Optimize inventory management and supply chain using data insights.
- Gather and utilize customer feedback to enhance satisfaction.

4. Database Design

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 Table
- Shipping Information Table

A. Tables and Fields


a. User Table:

- USER_ID: Unique identifier for each user.
- FIRSTNAME AND 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)

b. Shipping Information Table:

- Shipping_ID Unique identifier for each shipping transaction.
- Order_ID Foreign key referencing Orders.
- Shipping Address: Address where the order is being shipped.
- Shipping Method: Shipping method (e.g., Standard, Express).
- Shipping Date: Date when the order was shipped.
- Delivery Date: Estimated or actual delivery date.

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

c. Payment Method Table:

- Payment_ID : Unique identifier for each payment.
- OrderID: Foreign key referencing Orders, linking payments to orders.
- PaymentDate: Date when the payment was made.
- PaymentMethod: Method of payment.
- PaymentStatus: Status of the payment.

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

d. Order Table:

- Order_ID (INTEGER): Unique identifier for each order.
- USER_ID (INTEGER): Foreign key referencing USERS, linking orders to users.
- Order Date (DATE): Date when the order was placed.
- Total Amount (DECIMAL): Total amount of the order.
- Payment Status (VARCHAR(50)): Status of the payment

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

e. Feedback Table:

- Feedback_ID: Unique identifier for each feedback entry.
- User_ID: Foreign key referencing USERS.
- Order_ID: Foreign key referencing ORDERS.
- Feedback Text: Customer feedback or review.
- Rating: Rating given by the customer (e.g., 1-5 stars).
- Feedback Date: Date when the feedback was submitted.

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

f. Product Table:

- **PRODUCT_ID**: Unique identifier for each product.
- **Name**: Name of the product.
- **CATEGORY_ID**: Foreign key referencing Product Categories.
- **Order_ID**: Foreign key referencing Orders (optional, linking products to specific orders).
- **SUPPLIER_ID**: Foreign key referencing Suppliers.
- **Price**: Price of the product.
- **Material**: Material used in the product.
- **Size**: Size of the product (e.g., shoe size).
- **Color**: Color of the product.
- **Stock Quantity**: Available stock of the product.
- **Launch Date**: Date when the product was launched.

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

g. Product Categories Table:

- Category_ID: Unique identifier for each product category.
- Name: Name of the category (e.g., Sneakers, Running Shoes).
- Description: Description of the category

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

h. Suppliers Table:

- Supplier_ID: Unique identifier for each supplier.
- Name: Name of the supplier.
- Contact Person: Contact person for the supplier.
- Phone: Phone number of the supplier.
- Email: Email address of the supplier.
- Address: Address of the supplier.

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

B. Relationships:

The relationships between the tables are vital to understanding the flow of information across different modules in our system. The primary relationships are as follows:

USERS & ORDERS: One-to-Many relationship: A User (USER_ID) can place multiple Orders (ORDER_ID), but each order is placed by only one user.

ORDERS & PAYMENTS: One-to-One relationship: Each Order (Order_ID) has a corresponding Payment (PAYMENT_ID) with details like payment method and status.

ORDERS & SHIPPING INFORMATION: One-to-One Relationship: Each Order (Order_ID) has a single Shipping Information record (SHIPPING_ID), which tracks details like shipping method and delivery date.

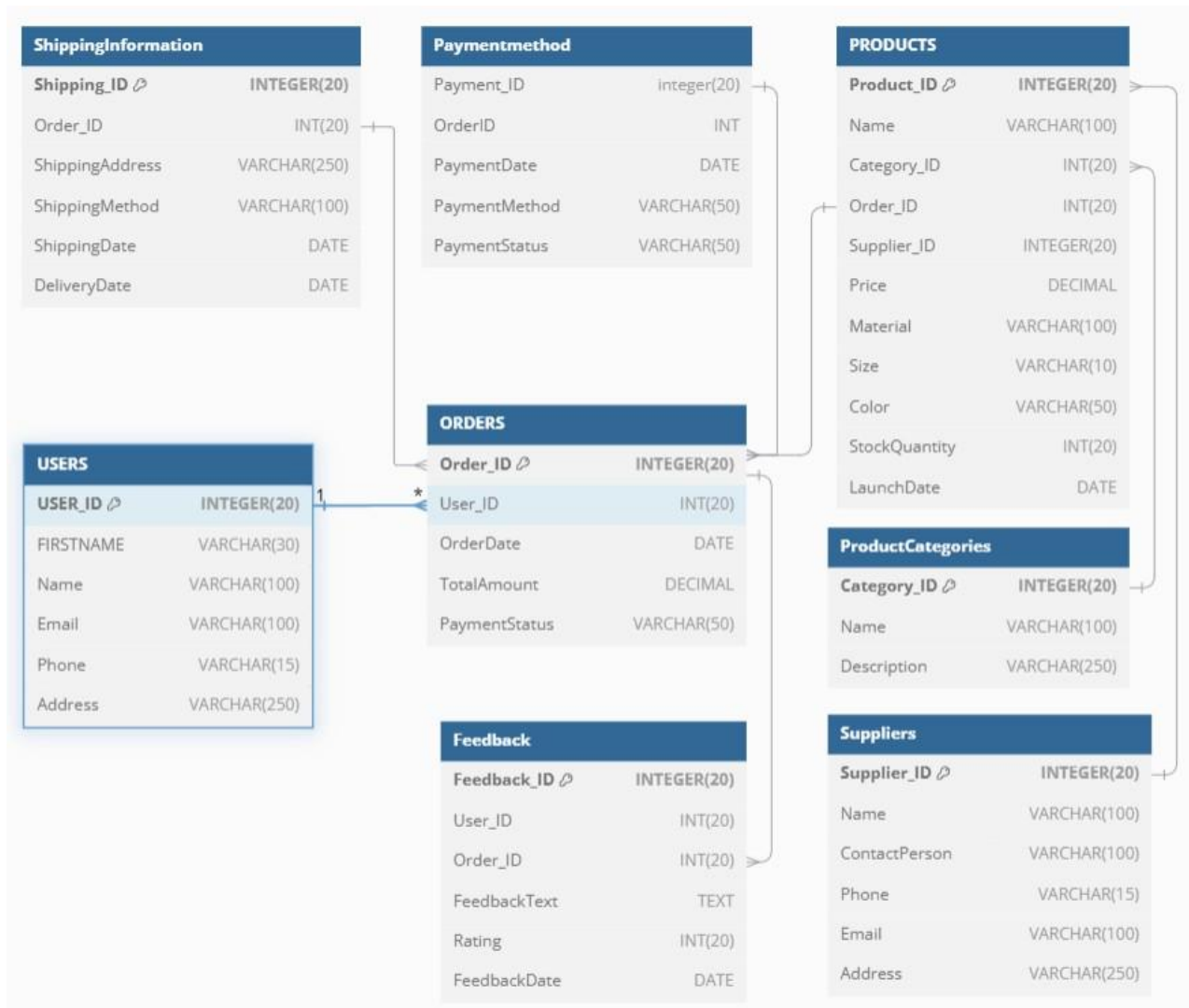
ORDERS & FEEDBACK: One-to-Many Relationship: A single Order (Order_ID) can have multiple Feedback entries (FEEDBACK_ID), but each feedback entry is linked to one specific order and user.

ORDERS & PRODUCTS: One-to-Many Relationship: An Order (Order_ID) can contain multiple Products (PRODUCT_ID), but each product is assigned to a specific order.

PRODUCTS & SUPPLIERS: Many-to-One Relationship: Multiple Products (PRODUCT_ID) can be supplied by a single Supplier (SUPPLIER_ID), but each product is associated with only one supplier

PRODUCTS & PRODUCT CATEGORIES: Many-to-One Relationship: A Product (PRODUCT_ID) belongs to a single Product Category (CATEGORY_ID), but each category can have multiple products.

5. Entity-Relationship Diagram (ERD):-



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:



Results		Messages				
	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.

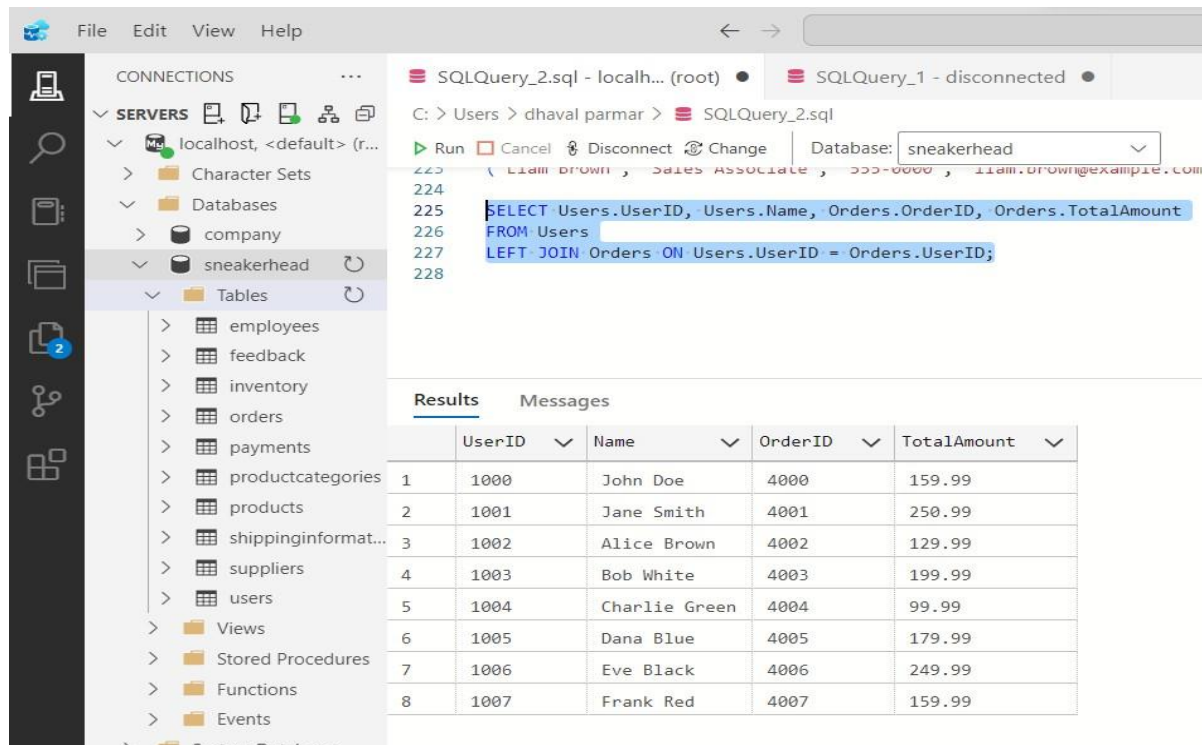
Query:

```
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 show the 'sneakerhead' database and its tables. The 'Tables' folder is selected, showing a list of tables including employees, feedback, inventory, orders, payments, productcategories, products, shippinginformat..., suppliers, users, Views, Stored Procedures, Functions, and Events. The 'SQLQuery_2.sql' file is open in the center pane, displaying 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, showing the output of the query. The results are displayed in a table with 5 columns: UserID, Name, OrderID, and TotalAmount. The table contains 8 rows of 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 show the 'sneakerhead' database. The 'SQLQuery_1 - localh...' window is active, displaying the following SQL query:

```
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;
```

The 'Results' tab is selected, showing the output of the query. The results are 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. On the left, the 'SERVERS' tree is expanded to show the 'sneakerhead' database. The 'Tables' folder is expanded, and the 'products' table is selected. The 'Columns' folder is also expanded, showing columns like ProductID, Name, CategoryID, Price, Material, Size, Color, StockQuantity, and LaunchDate. The 'SQLQuery_2.sql' window is open, showing the following query:

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

The 'Results' tab is selected, showing the output of the query. The results are displayed in a table with two columns: ProductName and SupplierName. The table contains 12 rows of data, representing the Cartesian product of the products and suppliers tables.

	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.