**Project Description: Textile Management System**

The Textile Management System is a database management project designed to streamline the operations of a textile retail business. This system is aimed at managing the key aspects of textile production, sales, inventory, and customer relations. It provides an efficient way to handle various functions such as tracking product details, managing customer orders, maintaining inventory levels, and generating sales reports. By utilizing a relational database management system (RDBMS), this project enables the storage, retrieval, and manipulation of data in an organized manner, ensuring that the business operates smoothly and effectively.

**Key Objectives:**

* Inventory Management: Keep track of available textile products, including details like SKU, name, category, brand, price, and quantity.
* Order Management: Manage customer orders, including order details such as customer information, order dates, quantities, and total prices.
* Customer Relationship Management: Store and manage customer data to enhance sales strategies and customer service.
* Sales Reporting: Generate reports to analyze sales performance, inventory turnover, and customer preferences.

**Tables Overview**

**Products Table:**

Description: Stores information about textile products.

Columns:

* sku: Unique identifier for each product (Primary Key).
* name: Name of the product.
* category: Category of the product.
* brand: Brand associated with the product.
* price: Price of the product.
* quantity: Available stock quantity.

**Customers Table**:

Description: Contains customer details.

Columns:

* customer\_id: Unique identifier for each customer (Primary Key).
* customer\_name: Name of the customer.
* email: Customer's email address.
* phone: Contact number.

**Orders Table:**

Description: Tracks orders made by customers.

Columns:

* order\_id: Unique identifier for each order (Primary Key).
* customer\_id: Reference to the customer placing the order (Foreign Key).
* sku: Reference to the product being ordered (Foreign Key).
* order\_date: Date when the order was placed.
* quantity: Number of items ordered.
* total\_price: Total cost of the order.

**Creation of the Textile Management Database**

The creation of the Textile Management Database involves a series of structured steps, encompassing planning, design, implementation, and testing. This process ensures that the database is robust, efficient, and capable of meeting the needs of the textile management system.

**1. Planning Phase**

Before creating the database, it is crucial to outline the objectives and requirements. During this phase, the following considerations are made:

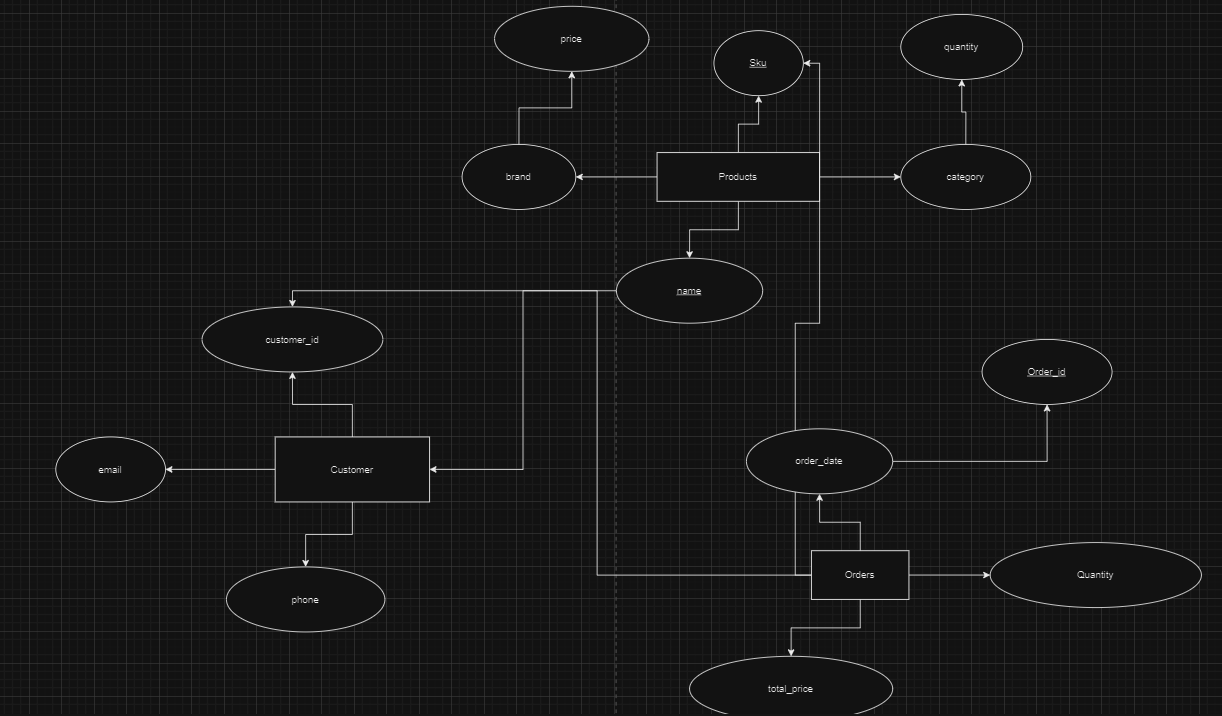
* Identify Users: Determine who will use the database, such as sales representatives, inventory managers, and customers.
* Define Scope: Establish the features and functionalities required, including inventory tracking, order processing, and reporting.
* Data Requirements: Identify the types of data that will be stored, such as product information, customer details, and order records.

**2. Design Phase**

Once the planning phase is complete, the next step is designing the database schema. This includes defining the tables, relationships, and constraints. The key components include:

* Entity-Relationship Diagram (ERD): Create an ERD to visualize the entities (tables) and their relationships. The main entities for this project are Products, Customers, and Orders.
* Define Tables: Each entity translates into a table with specific columns. For example:
* Products Table: Stores product details such as SKU, name, category, brand, price, and quantity.
* Customers Table: Contains customer information including customer ID, name, email, and phone number.
* Orders Table: Tracks order data, including order ID, customer ID, SKU, order date, quantity, and total price.

**ER -Diagram**

****

**3. Implementation Phase**

In this phase, the actual creation of the database and its tables occurs using SQL commands. Below is a step-by-step process for creating the database:

The Textile Management System incorporates several core database concepts, which are detailed below:

**Data Definition Language (DDL):**

DDL is used to define and manage all database objects, including tables and relationships. Commands like CREATE, ALTER, and DROP are essential for setting up the database schema.

**Creation of the Database :**

**Queries**:

mysql> create database Textile\_Management\_System;

mysql> use Textile\_Management\_System;

**Output:**

Database changed

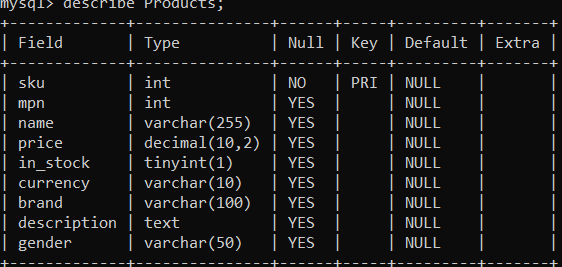
**Creation of Tables :**

Queries:

**Products Table**

mysql> CREATE TABLE Products ( sku INT PRIMARY KEY, mpn INT, name VARCHAR(255), price DECIMAL(10, 2), in\_stock BOOLEAN, currency VARCHAR(10), brand VARCHAR(100), description TEXT, gender VARCHAR(50));

Schema:



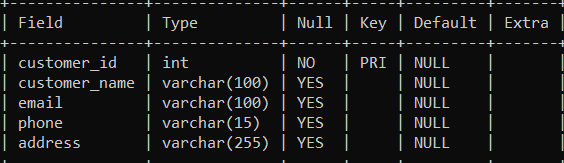
**Customers Tabl**e

Queries:

mysql> CREATE TABLE Customers ( customer\_id INT PRIMARY KEY, customer\_name VARCHAR(100), email VARCHAR(100), phone VARCHAR(15), address VARCHAR(255));

Query OK, 0 rows affected (0.05 sec)

Schema:

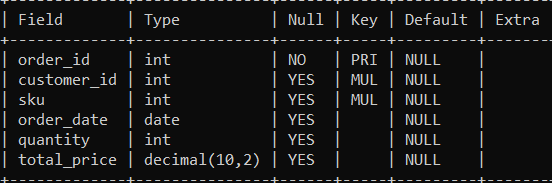


**Orders Table**

Queries:

mysql> CREATE TABLE Orders ( order\_id INT PRIMARY KEY, customer\_id INT, sku INT, order\_date DATE, quantity INT, total\_price DECIMAL(10, 2), FOREIGN KEY (customer\_id) REFERENCES Customers(customer\_id), FOREIGN KEY (sku) REFERENCES Products(sku));

Schema:



**Data Manipulation Language (DML):**

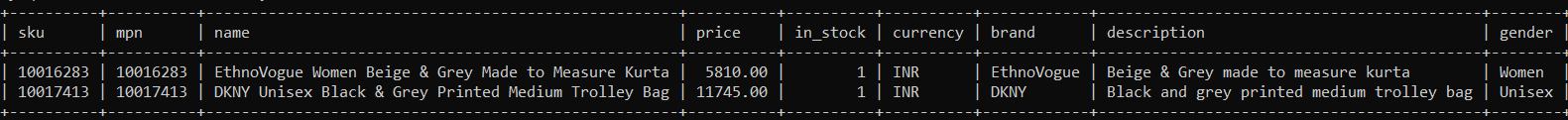
DML commands facilitate the manipulation of data within the tables, enabling operations such as inserting, updating, deleting, and retrieving records.

**Insertion of Values :**

**Queries:**

**Products Table**

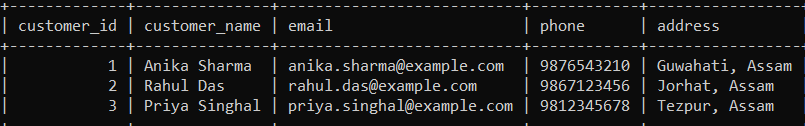
mysql> INSERT INTO Products (sku, mpn, name, price, in\_stock, currency, brand, description, gender)VALUES (10017413, 10017413, 'DKNY Unisex Black & Grey Printed Medium Trolley Bag', 11745, TRUE, 'INR', 'DKNY', 'Black and grey printed medium trolley bag', 'Unisex'), (10016283, 10016283, 'EthnoVogue Women Beige & Grey Made to Measure Kurta', 5810, TRUE, 'INR', 'EthnoVogue', 'Beige & Grey made to measure kurta', 'Women');



**Customers Table**

**Queries:**

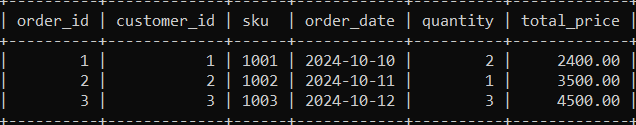
mysql> INSERT INTO Customers (customer\_id, customer\_name, email, phone, address)VALUES (1, 'Anika Sharma', 'anika.sharma@example.com', '9876543210', 'Guwahati, Assam'), (2, 'Rahul Das', 'rahul.das@example.com', '9867123456', 'Jorhat, Assam'), (3, 'Priya Singhal', 'priya.singhal@example.com', '9812345678', 'Tezpur, Assam');



**Orders Table**

**Queries:**

mysql> INSERT INTO Products (sku, name, price, in\_stock, currency, brand, description, gender)VALUES (1001, 'EthnoVogue Kurta', 1200.00, TRUE, 'INR', 'EthnoVogue', 'Traditional cotton kurta', 'Female'),(1002, 'Raymond Men’s Shirt', 3500.00, TRUE, 'INR', 'Raymond', 'Formal office shirt', 'Male'),(1003, 'Levi\'s Women Jeans', 1500.00, TRUE, 'INR', 'Levi\'s', 'Denim jeans for women', 'Female');

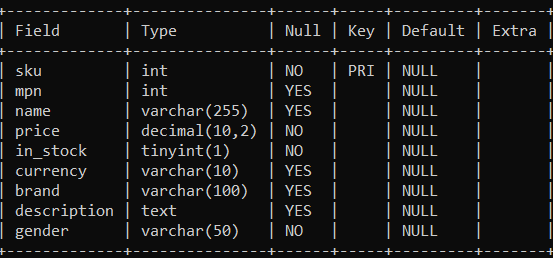


**Constraints:**

Constraints are rules applied to table columns to enforce data integrity. Common constraints include PRIMARY KEY, FOREIGN KEY, NOT NULL, UNIQUE, and CHECK.

Queries:

mysql> ALTER TABLE Products MODIFY price DECIMAL(10, 2) NOT NULL CHECK (price > 0),MODIFY in\_stock BOOLEAN NOT NULL, MODIFY gender VARCHAR(50) NOT NULL;



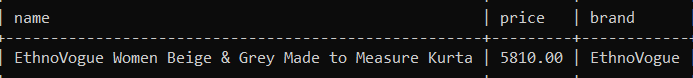
**WHERE Clause:**

The WHERE clause is used to filter records based on specific conditions.

In this Project , it is used for filtering Products by Gender and Price

Queries:

mysql> SELECT name, price, brand FROM Products WHERE gender = 'Women' AND price > 1000;



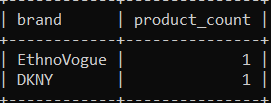
**GROUP BY Clause**:

This clause is utilized to aggregate data based on one or more columns.

In this project it is used for grouping by Brand and Counting Products

Queries:

mysql> SELECT brand, COUNT(\*) AS product\_count FROM Products GROUP BY brand;



6. **HAVING Clause**:

The HAVING clause is used in conjunction with GROUP BY to filter groups based on aggregate functions.

In this project it is used for filtering brands with More Than 10 Products

Queries:

mysql> SELECT brand, COUNT(\*) AS product\_count FROM Products GROUP BY brand HAVING COUNT(\*) > 10;

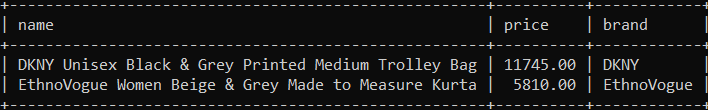
7. **ORDER BY Clause**:

This clause sorts the result set based on one or more columns.

In this project it is used for ordering Products by Price (Descending)

Queries:

mysql> SELECT name, price, brand FROM Products ORDER BY price DESC;



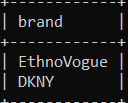
**DISTINCT and LIMIT**:

The DISTINCT keyword eliminates duplicate records, while LIMIT restricts the number of records returned by a query.

In this project it is used for retrieving Distinct Brands and Limiting Results

Queries:

mysql> SELECT DISTINCT brand FROM Products LIMIT 5;



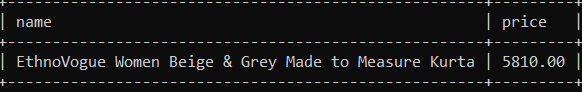
**Pattern Matching Using LIKE**:

The LIKE operator is used to search for a specified pattern in a column.

In this project it is used for finding Products Related to "Kurta"

Queries:

mysql> SELECT name, price FROM Products WHERE name LIKE '%Kurta%';



**Joins:**

Joins combine rows from two or more tables based on related columns.

Types of Joins:

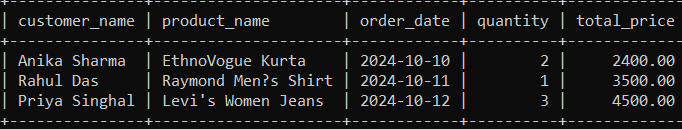
* INNER JOIN: Returns records with matching values in both tables.
* LEFT JOIN: Returns all records from the left table and matched records from the right table.
* RIGHT JOIN: Returns all records from the right table and matched records from the left table.
* FULL OUTER JOIN: Returns all records when there is a match in either left or right table records.

Here ,

a. **Inner Join**: Retrieve customers who have placed orders along with product details.

Queries:

mysql> SELECT C.customer\_name, P.name AS product\_name, O.order\_date, O.quantity, O.total\_price FROM Customers C JOIN Orders O ON C.customer\_id = O.customer\_id JOIN Products P ON O.sku = P.sku;

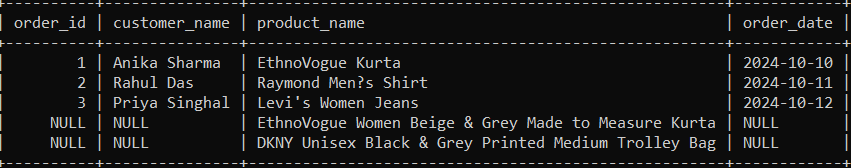


This will return only the customers who have placed an order, showing their name, the product they ordered, the order date, quantity, and total price.

b. **Left Join**: Retrieve all customers, including those who haven’t placed an order.

Queries:

mysql> SELECT O.order\_id, C.customer\_name, P.name AS product\_name, O.order\_date FROM Orders O RIGHT JOIN Customers C ON O.customer\_id = C.customer\_id RIGHT JOIN Products P ON O.sku = P.sku;

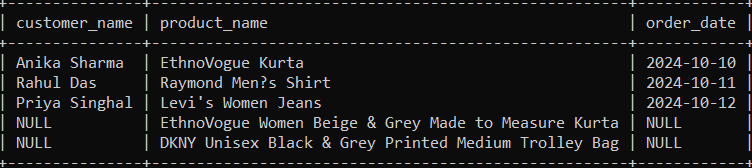


This query will return all customers. If a customer has not placed an order, the order\_id, product\_name, and order\_date will be NULL.

c. **Right Join**: Retrieve all orders, even if the customer is missing

Queries:

mysql> SELECT O.order\_id, C.customer\_name, P.name AS product\_name, O.order\_date FROM Orders O RIGHT JOIN Customers C ON O.customer\_id = C.customer\_id RIGHT JOIN Products P ON O.sku = P.sku;

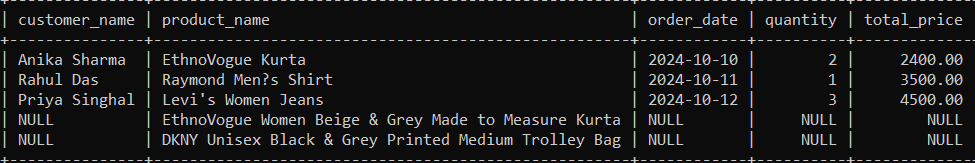


This query will return all orders and the corresponding customer information. If a customer doesn’t exist in the Customers table, the customer name will be NULL

d. **Full Outer Join** (Emulated in MySQL using UNION):

Queries:

mysql> SELECT C.customer\_name, P.name AS product\_name, O.order\_date, O.quantity, O.total\_price FROM Customers C LEFT JOIN Orders O ON C.customer\_id = O.customer\_id LEFT JOIN Products P ON O.sku = P.sku UNION SELECT C.customer\_name, P.name AS product\_name, O.order\_date, O.quantity, O.total\_price FROM Customers C RIGHT JOIN Orders O ON C.customer\_id = O.customer\_id RIGHT JOIN Products P ON O.sku = P.sku;



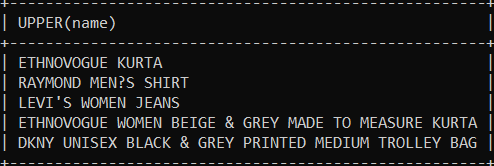
This will return all customers and all orders, regardless of whether there’s a match between the two.

**String, Numeric, and Date-Time Functions**:

* These functions manipulate string data, perform calculations, and handle date-time values.

Queries:

mysql> SELECT UPPER(name) FROM Products;

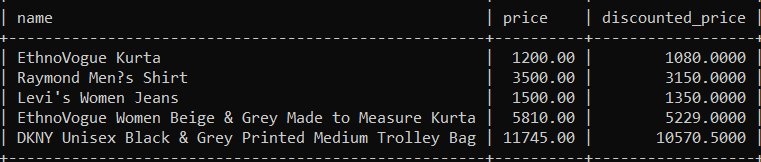


Numeric Functions:

To find the discounted Price:

Queries:

SELECT name, price, (price \* 0.90) AS discounted\_price FROM Products;

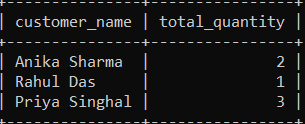


**Total Products Sold by Each Customer**

Get the total quantity of products sold to each customer.

Queries:

mysql> SELECT C.customer\_name, SUM(O.quantity) AS total\_quantity FROM Customers C JOIN Orders O ON C.customer\_id = O.customer\_id GROUP BY C.customer\_name;

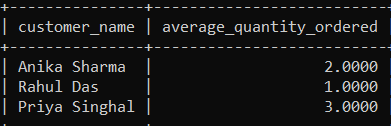


### **Average Quantity Ordered Per Customer**

Calculate the average quantity of items ordered by each customer.

Queries:

mysql> SELECT C.customer\_name, AVG(O.quantity) AS average\_quantity\_ordered FROM Customers C JOIN Orders O ON C.customer\_id = O.customer\_id GROUP BY C.customer\_name;



### **Stock Availability Check**

List products that are low in stock (assuming in\_stock is a numeric field representing the quantity in stock).

Queries:

mysql> SELECT P.name AS product\_name, P.in\_stock FROM Products P WHERE P.in\_stock < 10;

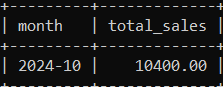


### **Sales Over Time**

Calculate total sales per month.

Queries:

mysql> SELECT DATE\_FORMAT(O.order\_date, '%Y-%m') AS month, SUM(O.total\_price) AS total\_sales FROM Orders O GROUP BY month ORDER BY month;



**Nested Queries and Subqueries**:

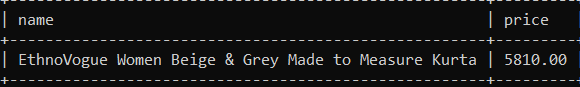
* Nested queries are queries within another query, allowing for complex data retrieva

**Customers with Above Average Spending**

Find customers whose total spending exceeds the average spending across all customers.

Queries:

mysql> SELECT name, price FROM Products WHERE price = (SELECT MAX(price) FROM Products WHERE gender = 'Women');

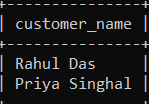


### **Total Orders by Each Customer**

Get a count of orders placed by each customer.

Queries:

mysql> SELECT C.customer\_name FROM Customers C WHERE (SELECT SUM(O.total\_price) FROM Orders O WHERE O.customer\_id = C.customer\_id) > (SELECT AVG(total\_spent) FROM (SELECT SUM(O.total\_price) AS total\_spent FROM Orders O GROUP BY O.customer\_id) AS customer\_spending);

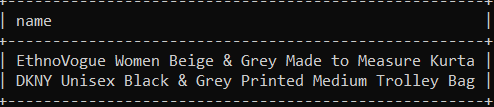


### **Products with Higher Average Price than All Others**

Find products that have a higher average price than the average price of all other products.

Queries:

mysql> SELECT P.name FROM Products P WHERE P.price > (SELECT AVG(price) FROM Products);



**(VDL)**

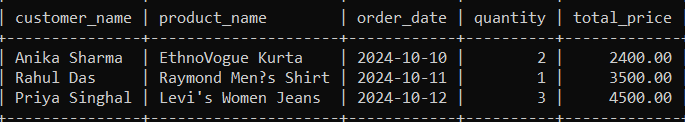
Creating a view in SQL allows you to simplify complex queries by encapsulating them in a virtual table that can be treated like a regular table. Here creating a view for the Textile Management System project and performing queries on it.

Step 1: Create a View

Creation of a view that summarizes order details along with customer and product information. This view will include customer names, product names, order dates, quantities, and total prices

Queries:’

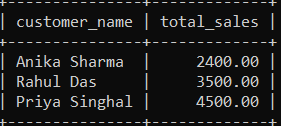
mysql> CREATE VIEW OrderSummary AS SELECT C.customer\_name, P.name AS product\_name, O.order\_date, O.quantity, O.total\_price FROM Customers C JOIN Orders O ON C.customer\_id = O.customer\_id JOIN Products P ON O.sku = P.sku;



Sum of the total Sales :[Performing Queries in View]

Queries:

mysql> SELECT customer\_name, SUM(total\_price) AS total\_sales FROM OrderSummary GROUP BY customer\_name;



Transaction Control Language:

TCL commands manage transactions in the database, ensuring data integrity. Commands like COMMIT, ROLLBACK, and SAVEPOINT are used to manage transactions.

Queries:

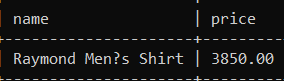
mysql>begin;

mysql> COMMIT;

mysql> BEGIN;UPDATE Products SET price = price \* 1.20 WHERE brand = 'Parx';ROLLBACK;

mysql> UPDATE Products SET price = price \* 1.10 WHERE brand = 'Raymond';

mysql> SELECT name, price FROM Products WHERE brand = 'Raymond';



**Insights About the Project**

* Data Integrity: The use of primary key and foreign key constraints significantly enhances data integrity. This ensures that relationships between entities are maintained and that invalid data entries are prevented.
* Efficient Data Retrieval: The ability to perform complex JOIN operations allows for efficient data retrieval, enabling users to gather insights from multiple tables without redundancy. This is especially useful for generating reports and analyzing customer behavior.
* Improved Query Performance: By utilizing aggregate functions and indexing on key columns, the database can provide quick responses to data retrieval requests, making it more user-friendly.
* Scalability and Maintenance: The structured approach in designing the database allows for scalability. As the business grows, more tables and relationships can be added with minimal disruption to the existing database schema.
* Business Insights: The database enables businesses to analyze their inventory, customer purchasing patterns, and sales performance. For example, queries to calculate total sales revenue can help in understanding market trends and making informed decisions regarding inventory management and marketing strategies.

Conclusion

In summary, the Textile Management Database utilizes various constraints and queries to ensure data integrity, facilitate efficient data retrieval, and provide valuable business insights. By implementing a structured approach to database design and leveraging SQL capabilities, the system is well-equipped to support the operations and decision-making processes within the textile industry.

Github Link: