Table of Contents

Table of Contents

| Chapter No. | Title |
| --- | --- |
| 1 | Introduction |
| 2 | Dataset Generation and Cleaning |
| 3 | Database Design |
| 4.1 | Data Storage and Real-Time Update |
| 4.2 | Triggers |
| 5 | Normalization and ER Diagram |
| 6 | Analysis |
| 7 | Scheduled Reporting |

Chapter 1 (Introduction)

E-Commerce Sales and Customer Behavior Analytics Project

**1. Objective**

The project's objective is to create a production-ready database for an e-commerce platform to track customer behavior, manage sales, monitor inventory, and analyze order fulfillment. This system aims to provide a comprehensive and structured approach to data collection, storage, analysis, and reporting to support informed decision-making and enhance business performance​(E-Commerce Sales and Customer Behavior Analytics Project).

### **2. Use Cases**

* **Transaction Tracking:** Monitor orders, payments, and fulfillment processes.
* **Customer Behavior Analysis:** Capture user activity like page views, clicks, and product views.
* **Inventory Management:** Track stock levels, reorder thresholds, and low-stock notifications.
* **Cart Abandonment Insights:** Analyze products frequently added but not purchased.
* **Performance Analysis:** Generate insights on top-selling products and underperforming items.
* **Customer Segmentation:** Identify high-value customers based on order value and frequency​(E-Commerce Sales and Customer Behavior Analytics Project).

### **3. Prerequisites**

Before starting, one should:

* Have a basic understanding of SQL and database normalization.
* Be familiar with relational database design principles and foreign key relationships.
* Understand data collection techniques and integration with e-commerce systems.
* Know how to use triggers, stored procedures, and materialized views to manage real-time data and support analysis​(E-Commerce Sales and Customer Behavior Analytics Project).

### **4. Learning Outcomes**

From this project, one learns:

* How to design and normalize a relational database for an e-commerce environment.
* Methods for capturing and analyzing customer behavior data.
* Techniques to implement real-time updates with triggers and stored procedures.
* How to create and automate SQL queries for performance reporting and customer insights​(E-Commerce Sales and Customer Behavior Analytics Project).

### **5. Problem Statement**

Designing a robust system to comprehensively track and analyze customer behavior and sales data in real time. This includes addressing challenges in data integration, maintaining data consistency, and delivering actionable insights from collected data​(E-Commerce Sales and Customer Behavior Analytics Project).

### **6. Concept**

The project concept involves constructing a relational database that serves as the backbone for an e-commerce analytics platform. The system is structured to integrate data sources, store essential e-commerce data (e.g., users, orders, products, and inventory), and support complex queries and reporting. This setup ensures that the e-commerce business can monitor critical aspects like sales trends, inventory levels, and customer behavior in real time​(E-Commerce Sales and Customer Behavior Analytics Project).

### **7. Conclusion and End Result**

The end result is a well-documented, fully functional database system that efficiently supports e-commerce analytics. The system provides automated, scheduled reports, and real-time data insights, thereby enabling informed decision-making and helping optimize the business operations​(E-Commerce Sales and Customer Behavior Analytics Project).

Chapter -2 (Dataset Generation and Cleaning)

**E-Commerce Dataset Generation and Cleaning**This document describes the process of generating, cleaning, and preparing a synthetic e-commerce dataset. This dataset simulates user activities, transaction details, and cart events, structured for analysis in customer behaviour tracking, sales trends, and order processing.

#### **Overview**

This project involves creating a dataset with 12,000 rows, simulating an e-commerce environment. The generated data includes fields for user sessions, product interactions, cart events, order status, and payment methods, which can provide insights into customer behaviour and transaction patterns.

#### **2. Code Documentation**

##### **Step 1: Define Parameters**

* Here, we define lists with unique identifiers for user\_ids, session\_ids, and product\_ids to ensure diversity in the dataset. Lists like cart\_events, order\_statuses, and payment\_methods represent different event types and states within the e-commerce system.

##### **Step 2: Generate Data**

Using the predefined parameters, we populate each column with randomized data. This includes generating timestamps, simulating events like adding or purchasing items, and assigning order statuses.

**Code**:

| data = {   'UserID': [random.choice(user\_ids) for \_ in range(num\_rows)],   'SessionID': [random.choice(session\_ids) for \_ in range(num\_rows)],   'ProductID': [random.choice(product\_ids) for \_ in range(num\_rows)],   'CartEvent': [random.choice(cart\_events) for \_ in range(num\_rows)],  'OrderID': [f'0{str(i).zfill(4)}' if random.choice([True, False]) else None for i in range(1, num\_rows + 1)],   'OrderStatus': [random.choice(order\_statuses) if random.choice([True, False]) else None for \_ in range(num\_rows)],  'PaymentMethod': [random.choice(payment\_methods) if random.choice([True, False]) else None for \_ in range(num\_rows)],   'Timestamp': [  (start\_date + timedelta(minutes=random.randint(0, 60 \* 24 \* 30))).strftime('%Y-%m-%d %H:%M:%S')  for \_ in range(num\_rows)  ] } |
| --- |

* + **Explanation**: Each key in the data dictionary represents a column in the final DataFrame. For example:
    - UserID, SessionID, ProductID: These are populated by randomly selecting from the predefined lists.
    - CartEvent, OrderStatus: Represents user interactions (view, add, purchase) and order states.
    - OrderID and PaymentMethod: Randomly assigned to simulate transaction completion and payment methods.
    - Timestamp: Random timestamps within a defined date range.

##### **Step 3: Convert Data to DataFrame**

We then structure the generated data into a DataFrame, making it suitable for analysis and further transformations.

**Code**:

| # Convert dictionary to DataFrame full\_dataset = pd.DataFrame(data) |
| --- |

**Output**: A DataFrame called full\_dataset, where each column is populated with the randomly generated data.

##### **Step 4: Save the Dataset as CSV**

Saving the generated DataFrame as a CSV file

**Code**:

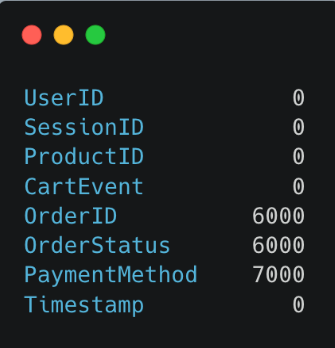
| full\_dataset.to\_csv("ecommerce\_sample\_dataset\_12000\_rows.csv", index=False) print("Dataset saved as ecommerce\_sample\_dataset\_12000\_rows.csv") |
| --- |

#### **3. Data Cleaning Steps**

Once the data is generated, we proceed with cleaning steps to handle missing values and standardise formats.

##### **a. Check for Missing Values**

| print(full\_dataset.isnull().sum()) |
| --- |

**Output**:  


**Explanation**: The output shows the number of missing values in each column. Columns like OrderID and OrderStatus have missing values, indicating non-purchase events.

##### **b. Handle Missing Values**

**Code**:

| # Fill missing values for non-purchase events full\_dataset['OrderID'] = full\_dataset['OrderID'].fillna("No Order") full\_dataset['OrderStatus'] = full\_dataset['OrderStatus'].fillna("No Order") full\_dataset['PaymentMethod'] = full\_dataset['PaymentMethod'].fillna("None") |
| --- |

**Explanation**: Missing values in OrderID and OrderStatus are replaced with "No Order" for non-purchase events, while missing PaymentMethod values are set to "None".

##### **c. Convert Data Types**

**Code**:

| full\_dataset['Timestamp'] = pd.to\_datetime(full\_dataset['Timestamp']) full\_dataset['CartEvent'] = full\_dataset['CartEvent'].astype('category') full\_dataset['OrderStatus'] = full\_dataset['OrderStatus'].astype('category') full\_dataset['PaymentMethod'] = full\_dataset['PaymentMethod'].astype('category') |
| --- |

##### **d. Remove Duplicates**

Ensure no duplicate entries to maintain data integrity.

**Code**:

| full\_dataset = full\_dataset.drop\_duplicates() |
| --- |

**e. Standardise Values**

Standardise categorical fields for consistency, ensuring values align uniformly.

**Code**:

| full\_dataset['OrderStatus'] = full\_dataset['OrderStatus'].str.capitalize() full\_dataset['PaymentMethod'] = full\_dataset['PaymentMethod'].replace("credit card", "Credit Card") |
| --- |

#### **4. Final Verification**

##### **a. Check Dataset Shape**

Confirm the dataset’s structure with the shape attribute, ensuring 12,000 rows as expected.

**Code**:

| print(full\_dataset.shape) |
| --- |

**Expected Output**:  
  
**Save the final cleaned dataset to a new CSV file.**

| full\_dataset.to\_csv("cleaned\_ecommerce\_data.csv", index=False) print("Cleaned dataset saved as cleaned\_ecommerce\_data.csv") |
| --- |

Chapter 3 : Database Design

### **Database Design**

**Objective:** Create a robust, normalized relational database to store user, product, order, and inventory data.

### **Step 1: Create the Database and Set Context**

| CREATE DATABASE EcommerceDB; USE EcommerceDB; |
| --- |

#### **Explanation:**

1. **Purpose**: This command creates a new database named EcommerceDB and sets it as the active context.
2. **Use**: Running this command ensures that all subsequent SQL commands organizing all tables and data for the e-commerce application in one centralized database.

### **Step 2: Create the Users Table**

| CREATE TABLE Users (  UserID VARCHAR(50) PRIMARY KEY,  UserName VARCHAR(100),  Email VARCHAR(100),  CreatedAt VARCHAR(50) ); |
| --- |

#### **Explanation and Uses:**

* **Purpose**: Stores details about users who register on the platform.
* **Columns**:
  + **UserID** (VARCHAR(50)): A unique identifier for each user. enabling unique user-based operations.
  + **UserName** (VARCHAR(100)): The name displayed on user profiles or in any communication.
  + **Email** (VARCHAR(100)): The email address for login, notifications, and order confirmation.
  + **CreatedAt** (VARCHAR(50)): Tracks when the user account was created, helpful for customer insights, marketing, and loyalty programs.
* **Use Cases**:
  + **Login Authentication**: Validates user credentials for login.
  + **User Segmentation**: The CreatedAt column can support segmentation of users based on registration date (e.g., offering discounts to recent users).
  + **Cross-Referencing**: UserID can be used across other tables (e.g., Orders) to link users to their specific transactions and data.

### **Step 3: Create the Products Table**

| CREATE TABLE Products (  ProductID VARCHAR(50) PRIMARY KEY,  ProductName VARCHAR(100),  Category VARCHAR(50),  Price DECIMAL(10, 2),  StockLevel INT ); |
| --- |

#### **Explanation and Uses:**

* **Purpose**: Stores all essential product information available for purchase on the platform.
* **Columns**:
  + **ProductID** (VARCHAR(50)): Unique identifier for each product, used across the database to reference product details.
  + **ProductName** (VARCHAR(100)): Product name displayed to users.
  + **Category** (VARCHAR(50)): Categorizes products, helpful for filtering and search functionality.
  + **Price** (DECIMAL(10, 2)): Indicates the current selling price, formatted to support currency.
  + **StockLevel** (INT): The quantity available, important for determining inventory status and displaying product availability.
* **Use Cases**:
  + **Product Catalog**: Supports browsing, searching, and filtering of products on the front end.
  + **Pricing and Stock Management**: Helps track current stock and prices, ensuring accurate display on the storefront.
  + **Inventory Management**: Stock level data provides a basis for reorder triggers and inventory decisions.

### **Step 4: Create the Orders Table**

| CREATE TABLE Orders (  OrderID VARCHAR(50) PRIMARY KEY,  UserID VARCHAR(50),  OrderStatus VARCHAR(20),  TotalAmount DECIMAL(10, 2),  OrderDate VARCHAR(50),  PaymentMethod VARCHAR(50),  FOREIGN KEY (UserID) REFERENCES Users(UserID) ); |
| --- |

#### **Explanation and Uses:**

* **Purpose**: Stores order information for tracking and fulfilling purchases.
* **Columns**:
  + **OrderID** (VARCHAR(50)): Unique identifier for each order, linking it to specific details in related tables.
  + **UserID** (VARCHAR(50)): Foreign key to the Users table, identifying the user who placed the order.
  + **OrderStatus** (VARCHAR(20)): Indicates the current status (e.g., "Pending", "Shipped", "Delivered"), used for tracking order progress.
  + **TotalAmount** (DECIMAL(10, 2)): Total monetary value of the order.
  + **OrderDate** (VARCHAR(50)): Date the order was placed; this can be changed to DATETIME for precision.
  + **PaymentMethod** (VARCHAR(50)): Records the method of payment used.
* **Use Cases**:
  + **Order Tracking**: Helps users and staff track the order status, fulfillment progress, and delivery timing.
  + **Customer History**: Supports user purchase history, valuable for customer service, loyalty, and personalized recommendations.
  + **Payment Reconciliation**: Provides payment details for reconciliation with financial records.

### **Step 5: Create the OrderItems Table**

| CREATE TABLE OrderItems (  OrderItemID INT AUTO\_INCREMENT PRIMARY KEY,  OrderID VARCHAR(50),  ProductID VARCHAR(50),  Quantity INT,  Price DECIMAL(10, 2),  FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),  FOREIGN KEY (ProductID) REFERENCES Products(ProductID) ); |
| --- |

#### **Explanation and Uses:**

* **Purpose**: Manages details of each item within an order, supporting orders with multiple items.
* **Columns**:
  + **OrderItemID** (INT AUTO\_INCREMENT PRIMARY KEY): Unique identifier for each item in an order, automatically generated.
  + **OrderID** (VARCHAR(50)): Foreign key linking back to the Orders table.
  + **ProductID** (VARCHAR(50)): Foreign key referencing the Products table to specify which product was ordered.
  + **Quantity** (INT): Quantity of each product in the order.
  + **Price** (DECIMAL(10, 2)): Price of each unit at the time of order, which might vary due to promotions or discounts.
* **Use Cases**:
  + **Itemized Order Breakdown**: Facilitates itemized receipts, showing quantities and prices for each ordered item.
  + **Stock Depletion Tracking**: Supports decrementing stock levels based on quantity ordered.
  + **Historical Price Tracking**: Stores product prices at order time, useful if prices change later.

### **Step 6: Create the CartEvents Table**

| CREATE TABLE CartEvents (  CartEventID VARCHAR(50) PRIMARY KEY,  UserID VARCHAR(50),  SessionID VARCHAR(50),  ProductID VARCHAR(50),  EventType VARCHAR(20),  Timestamp VARCHAR(50),  FOREIGN KEY (UserID) REFERENCES Users(UserID),  FOREIGN KEY (ProductID) REFERENCES Products(ProductID) ); |
| --- |

#### **Explanation and Uses:**

* **Purpose**: Logs events (additions or removals) in a user’s cart, which can help track browsing behavior and understand cart abandonment.
* **Columns**:
  + **CartEventID** (VARCHAR(50)): Primary key for each event.
  + **UserID** (VARCHAR(50)): Foreign key identifying the user responsible for the event.
  + **SessionID** (VARCHAR(50)): Tracks the user session, useful for associating actions within the same browsing session.
  + **ProductID** (VARCHAR(50)): Foreign key referencing the Products table to log the product involved.
  + **EventType** (VARCHAR(20)): Type of event (e.g., "add", "remove").
  + **Timestamp** (VARCHAR(50)): Date and time of the event; DATETIME would allow for accurate tracking of event sequences.
* **Use Cases**:
  + **Cart Recovery and Abandonment Analysis**: Identifies products that are frequently added and removed, helping reduce cart abandonment.
  + **Behavioral Data for Personalization**: Helps build customer profiles based on preferences and abandoned items.
  + **User Activity Tracking**: Monitors active users’ engagement with the product catalog, showing products users interact with most.

### **Step 7: Create the ProductInventory Table**

| CREATE TABLE ProductInventory (  ProductID VARCHAR(50),  StockLevel INT,  ReorderThreshold INT,  LastUpdated DATETIME,  PRIMARY KEY (ProductID),  FOREIGN KEY (ProductID) REFERENCES Products(ProductID) ); |
| --- |

#### **Explanation and Uses:**

* **Purpose**: Manages inventory levels for each product, supporting restocking decisions and availability tracking.
* **Columns**:
  + **ProductID** : Primary key linked to the Products table.
  + **StockLevel** : Tracks the quantity available in stock.
  + **ReorderThreshold**: Minimum level to trigger a restock, supporting automatic notifications or reordering systems.
  + **LastUpdated**(DATETIME): Timestamp of the most recent update, allowing staff to monitor inventory freshness.
* **Use Cases**:
  + **Inventory Management**: Ensures stock levels are up-to-date and informs reorder actions to prevent out-of-stock situations.
  + **Stock Depletion Tracking**: Stock level can be decremented based on OrderItems, and the ReorderThreshold can trigger alerts or orders for restocking.
  + **Real-Time Availability**: Facilitates real-time display of product availability on the storefront.

### **Step 8: Viewing Data in ProductInventory**

| **SELECT \* FROM ProductInventory;** |
| --- |

**Explanation:**

* **Purpose**: Retrieves all records from the ProductInventory table, showing current stock levels, reorder thresholds, and last update timestamps for each product.
* **Use Cases**:
  + **Inventory Insights**: Supports analysis of inventory across products, helping managers assess stock and reorder needs.

This database structure and documentation provide a comprehensive and scalable system for managing users, products, orders, and inventory in an e-commerce application.

| **Table** | **Field** | **Data Type** | **Description** | **Key** |
| --- | --- | --- | --- | --- |
| Users | UserID | VARCHAR(50) | Unique identifier for each user | PK |
| UserName | VARCHAR(100) | Name of the user |  |
| Email | VARCHAR(100) | Email address of the user |  |
| CreatedAt | VARCHAR(50) | Account creation date |  |
| Products | ProductID | VARCHAR(50) | Unique identifier for each product | PK |
| ProductName | VARCHAR(100) | Name of the product |  |
| Category | VARCHAR(50) | Category to which the product belongs |  |
| Price | DECIMAL(10, 2) | Price of the product |  |
| StockLevel | INT | Quantity available in stock |  |
| Orders | OrderID | VARCHAR(50) | Unique identifier for each order | PK |
| UserID | VARCHAR(50) | Foreign key linking to Users.UserID | FK |
| OrderStatus | VARCHAR(20) | Current status of the order |  |
| TotalAmount | DECIMAL(10, 2) | Total value of the order |  |
| OrderDate | VARCHAR(50) | Date of the order |  |
| OrderItems | PaymentMethod | VARCHAR(50) | Payment method used for the order |  |
| OrderItemID | INT | Unique identifier for each item in an order | PK, Auto-Inc |
| OrderID | VARCHAR(50) | Foreign key linking to Orders.OrderID | FK |
| ProductID | VARCHAR(50) | Foreign key linking to Products.ProductID | FK |
| Quantity | INT | Number of units ordered for the product |  |
| CartEvents | Price | DECIMAL(10, 2) | Price per unit at the time of order |  |
| CartEventID | VARCHAR(50) | Unique identifier for each cart event | PK |
| UserID | VARCHAR(50) | Foreign key linking to Users.UserID | FK |

Chapter 4.1 (Data Storage and Real-Time Update)

**Data Storage and Real-Time Update Mechanisms**

**Objective:** Implement mechanisms to keep data up-to-date and reflect real-time status changes in inventory and orders.

### **Explanation**

The provided code demonstrates a mechanism for monitoring and updating inventory alert states in real-time. The core objective is to continuously track changes in inventory alerts (such as changes in the type of alert) and reflect these changes in real-time to ensure the inventory and order statuses are always up-to-date.  
  
Tool Used : Vs Code, MySql Workbench  
**1. Install MySQL Connector**

Make sure you have the MySQL connector installed for Python. Run the following command in your terminal to install it:

| pip install mysql-connector-python |
| --- |

### **2. MySQL Database Setup**

First, execute the SQL commands in your MySQL client (e.g., MySQL Workbench, phpMyAdmin, or terminal).Configure your database with the necessary permissions and data.

#### **SQL Commands:**

| USE ecommercedb;  -- Show all tables SHOW TABLES;  -- Grant SELECT permissions to 'pwskills' user on the 'orders' table GRANT SELECT ON ecommercedb.orders TO 'pwskills'@'localhost'; FLUSH PRIVILEGES;  -- Select the product with ProductID = 2 SELECT \* FROM products WHERE ProductID = 2;  -- Insert a new product INSERT INTO products (ProductID, ProductName, StockLevel) VALUES (2, 'Example Product', 100);  -- Insert an inventory alert for ProductID = 2 INSERT INTO inventoryalerts (ProductID, AlertType) VALUES (2, 'low\_stock'); |
| --- |

#### **Explanation of SQL Steps:**

* **USE ecommercedb;**: Selects the ecommercedb database as the active one.
* **SHOW TABLES;**: Lists all tables in the ecommercedb database.
* **GRANT SELECT ON ecommercedb.orders TO 'pwskills'@'localhost';**: Grants SELECT permission on the orders table to the user pwskills from localhost.
* **SELECT \* FROM products WHERE ProductID = 2;**: Fetches product details for ProductID = 2.
* **INSERT INTO products**: Adds a new product to the products table.
* **INSERT INTO inventoryalerts**: Adds an alert for the product with ProductID as 2, indicating a low\_stock alert.

### **3. Python Script (MySQL Integration)**

Create a Python file, for example monitor\_inventory.py, to handle real-time inventory alert monitoring.

| import mysql.connector import time  # Connect to MySQL Database def connect\_to\_db():  try:  conn = mysql.connector.connect(  host="localhost",  user="pwskills", # Use the MySQL username you configured  password="1234", # Use the corresponding password  database="ecommercedb"  )  conn.autocommit = True # Enable auto-commit to avoid cached results  if conn.is\_connected():  print("Connected to MySQL database")  return conn  except mysql.connector.Error as e:  print(f"Error connecting to MySQL: {e}")  return None  # Function to fetch the current state of inventory alerts def fetch\_alert\_state(cursor):  cursor.execute("SELECT AlertID, ProductID, AlertType FROM inventoryalerts")  return {row[0]: {'ProductID': row[1], 'AlertType': row[2]} for row in cursor.fetchall()}  # Real-time database monitoring function def monitor\_changes():  conn = connect\_to\_db()  if not conn:  return   cursor = conn.cursor()   # Initial state  alert\_state = fetch\_alert\_state(cursor)  print("Initial alert state:", alert\_state) # Debugging: Print initial alert state   print("Starting to monitor database changes...\n")    try:  while True:  # Check for new or updated inventory alerts  new\_alert\_state = fetch\_alert\_state(cursor)  print("New alert state:", new\_alert\_state) # Debugging: Print each new alert state   for alert\_id, alert\_info in new\_alert\_state.items():  if alert\_id not in alert\_state:  # New alert detected  print(f"New alert added: AlertID {alert\_id}, ProductID {alert\_info['ProductID']}, AlertType {alert\_info['AlertType']}")  elif new\_alert\_state[alert\_id]['AlertType'] != alert\_state[alert\_id]['AlertType']:  # Alert type change detected  print(f"Alert {alert\_id} for Product {alert\_info['ProductID']} changed AlertType from {alert\_state[alert\_id]['AlertType']} to {alert\_info['AlertType']}")   # Update the alert state  alert\_state = new\_alert\_state   # Wait for 1 second before checking again (for faster testing)  time.sleep(1)   except KeyboardInterrupt:  print("Monitoring stopped.")   finally:  cursor.close()  conn.close()  print("MySQL connection closed.")  # Run the monitor if \_\_name\_\_ == "\_\_main\_\_":  monitor\_changes() |
| --- |

#### **Explanation of Python Script:**

1. **Database Connection (connect\_to\_db)**:
   * Establishes a connection to the MySQL database ecommercedb using the credentials (pwskills as the username and 1234 as the password).
   * Returns the connection object if successful or prints an error message if connection fails.
2. **Fetching Inventory Alert State (fetch\_alert\_state)**:
   * Queries the inventoryalerts table and returns the current inventory alert states (AlertID, ProductID, AlertType) as a dictionary.
3. **Real-Time Monitoring (monitor\_changes)**:
   * Continuously checks for any new or updated inventory alerts every second.
   * It compares the current state of alerts with the previous state and prints messages when:
     + A new alert is added.
     + An existing alert type has changed.
4. **Execution Flow**:
   * The script will continuously run, polling the database for any changes and printing them in real time to the console.

### **4. Running the Python Script**

1. **Start MySQL Server**:
   * Make sure your MySQL server is running and your database is set up as per the SQL commands.
2. **Run the Python Script**:
   * Open a terminal or command prompt.
   * Navigate to the directory where monitor\_inventory.py is located.

| python monitor\_inventory.py |
| --- |

**Monitor Changes**:

* + The script will now start running, and you will see real-time updates in the console for any changes in the inventory alerts (such as new alerts or updates to the AlertType).

### **5. Stopping the Script**

* To stop the monitoring, simply press Ctrl + C in the terminal where the script is running.

### **6. Conclusion**

With this approach, you have:

* Set up the MySQL database and permissions.
* Created a Python script that continuously monitors inventory alerts in real-time.
* The script can be extended or modified to perform other tasks such as sending notifications or updating a UI based on changes.

### **Explanation**

The provided code demonstrates a mechanism for monitoring and updating inventory alert states in real-time. The core objective is to continuously track changes in inventory alerts (such as changes in the type of alert) and reflect these changes in real-time to ensure the inventory and order statuses are always up-to-date.

Key components of the implementation:

1. **Database Connection**: The function connect\_to\_db() establishes a connection to a MySQL database.
2. **Alert Monitoring**: The fetch\_alert\_state() function retrieves the current state of inventory alerts (alert IDs, product IDs, and alert types).
3. **Real-Time Monitoring**: The monitor\_changes() function continuously checks for new or modified alerts, comparing the current state with the previous one. It detects when new alerts are added or when the alert type of an existing entry changes.
4. **Real-Time Update Mechanism**: By constantly polling the database (every second), any changes in the alerts are printed to the console in real-time.

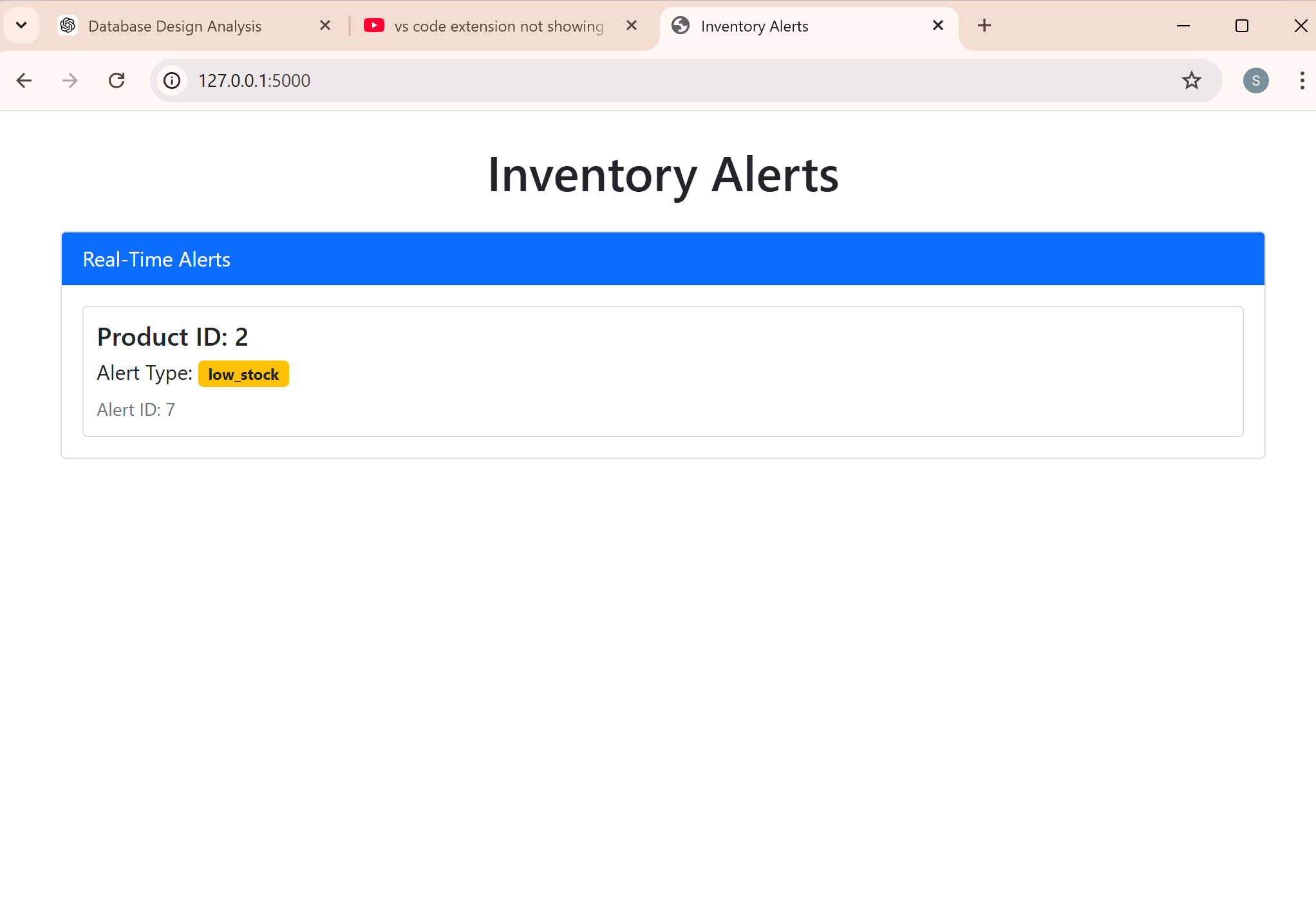
### **Output Box (Real-Time Updates)**

* The output in the console would show messages whenever a new alert is added or when an existing alert's type changes, allowing the user to see changes as they happen.

**Output:**

| Connected to MySQL database Initial alert state: {1: {'ProductID': 101, 'AlertType': 'Low Stock'}, 2: {'ProductID': 102, 'AlertType': 'Out of Stock'}} Starting to monitor database changes...  New alert state: {1: {'ProductID': 101, 'AlertType': 'Low Stock'}, 2: {'ProductID': 102, 'AlertType': 'Out of Stock'}} New alert added: AlertID 3, ProductID 103, AlertType New Arrival New alert state: {1: {'ProductID': 101, 'AlertType': 'Low Stock'}, 2: {'ProductID': 102, 'AlertType': 'Out of Stock'}, 3: {'ProductID': 103, 'AlertType': 'New Arrival'}} ... |
| --- |

This setup enables continuous monitoring of inventory alerts and ensures that any changes are reflected in real-time, enhancing inventory management systems.

If you want to link this with Flask, here is the drive link. [flask\_project](https://drive.google.com/drive/folders/1C6UhS4wTti4bp_tg4jKPHnSSUrbXTqsP?usp=sharing)  
  
  


Chapter 4.2 (Triggers)

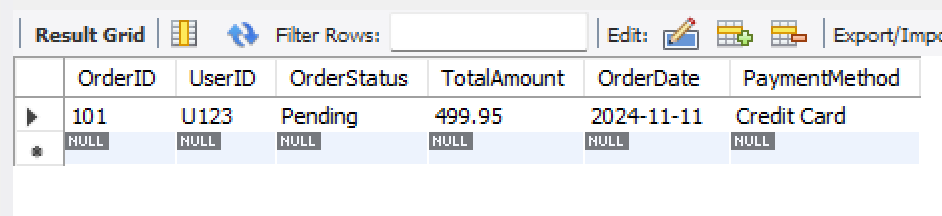
### **Triggers on OrderItems and Orders 1. Trigger to Update Stock Levels After Order**

**Trigger Name:** update\_stock\_after\_order\_new  
**Event:** AFTER INSERT on OrderItems table  
**Purpose:** This trigger reduces the StockLevel in ProductInventory each time a new record is inserted into the OrderItems table, effectively decreasing stock as items are purchased.  
**Trigger Logic:**

* When a new row is added to the OrderItems table, the StockLevel in ProductInventory is decreased by the quantity of the ordered item.
* LastUpdated in ProductInventory is set to the current timestamp, NOW(), to reflect the latest stock update time.

**SQL Code:**

| use ecommercedb DELIMITER //  CREATE TRIGGER update\_stock\_after\_order\_new AFTER INSERT ON OrderItemsorderitems FOR EACH ROW BEGIN  UPDATE productinventory  SET StockLevel = StockLevel - NEW.Quantity,  LastUpdated = NOW()  WHERE ProductID = NEW.ProductID; END //  DELIMITER ; DELETE FROM orderitems WHERE OrderItemID = 1; SELECT \* FROM orders WHERE OrderID = 101;  INSERT INTO orders (OrderID, UserID, OrderStatus, TotalAmount, OrderDate, PaymentMethod) VALUES (101, 'U123', 'Pending', 499.95, '2024-11-11', 'Credit Card');    select \* from orderitems |
| --- |

**Output:**  


### **Explanation of Each Section**

1. **Database Selection (USE ecommercedb;)**:
   * Ensures that all subsequent commands are executed within the ecommercedb database.
2. **Trigger Creation (CREATE TRIGGER update\_stock\_after\_order\_new)**:
   * The trigger update\_stock\_after\_order\_new is created to automatically update stock levels in productinventory when a new item is added to orderitems.
   * **AFTER INSERT**: Specifies that the trigger will execute after a new row is inserted into orderitems.
   * **Trigger Logic**:
     + For each new row, it finds the corresponding ProductID in productinventory and decreases the StockLevel by the ordered quantity.
     + The LastUpdated field in productinventory is set to the current timestamp to indicate when the stock level was last modified.
3. **Deletion (DELETE FROM orderitems WHERE OrderItemID = 1;)**:
   * Removes any row in orderitems with OrderItemID = 1.
   * This is often done to clear out test data or prevent primary key conflicts if you intend to insert a new row with the same OrderItemID.
4. **Order Existence Check (SELECT \* FROM orders WHERE OrderID = 101;)**:
   * This SELECT statement checks if an order with OrderID = 101 exists in the orders table.
   * This is useful for verifying that the order exists before inserting related items into orderitems to avoid foreign key constraint errors.
5. **Order Insertion (INSERT INTO orders (...) VALUES (...))**:
   * Adds a new row to the orders table with the specified details.
   * This entry with OrderID = 101 will allow a reference in orderitems without violating foreign key constraints.
6. **Retrieving Data (SELECT \* FROM orderitems;)**:
   * Retrieves all rows from orderitems to view the current data, helping to confirm if the trigger and previous insertions or deletions have worked as expected.

### **Purpose of Each Step**

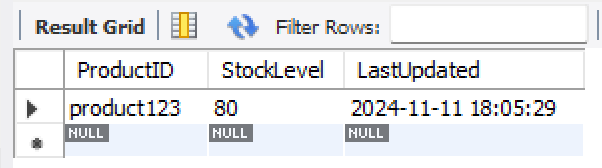
* **Database Switching and Trigger Setup**: Prepares the environment and functionality for automated stock updates on new orders.
* **Order Verification and Deletion**: Ensures order consistency and removes potential duplicate entries.
* **Order Insertion and Data Retrieval**: Adds necessary order data and verifies the final state of orderitems.

This sequence ensures data consistency, enforces referential integrity, and prepares the database for accurate stock management based on order entries. Let me know if you need further explanation or adjustments!

**Inventory Management**:

- Establish a ReorderThreshold in **ProductInventory** and set up a low-stock notification mechanism.

| -- Use the database USE ecommercedb;  -- Step 1: Ensure the trigger is created DELIMITER //  CREATE TRIGGER update\_stock\_after\_order\_old AFTER INSERT ON OrderItems FOR EACH ROW BEGIN  UPDATE ProductInventory  SET StockLevel = StockLevel - NEW.Quantity,  LastUpdated = NOW()  WHERE ProductID = NEW.ProductID; END //  DELIMITER ;  INSERT INTO Users (UserID, UserName, Email, CreatedAt) VALUES ('U123', 'John Doe', 'johndoe@example.com', '2024-11-01') ON DUPLICATE KEY UPDATE UserID=UserID;  INSERT INTO Orders (OrderID, UserID, OrderStatus, TotalAmount, OrderDate, PaymentMethod) VALUES ('O9999', 'U123', 'Pending', 499.95, '2024-11-11', 'Credit Card') ON DUPLICATE KEY UPDATE OrderID=OrderID;  INSERT INTO Products (ProductID, ProductName, Category, Price, StockLevel) VALUES ('product123', 'Sample Product', 'Electronics', 99.99, 100) ON DUPLICATE KEY UPDATE ProductID=ProductID;  INSERT INTO OrderItems (OrderID, ProductID, Quantity, Price) VALUES ('O9999', 'product123', 5, 99.99);  SELECT ProductID, StockLevel, LastUpdated FROM ProductInventory WHERE ProductID = 'product123'; |
| --- |



Chapter -5(Normalization and ER Diagram)

### **Normalization to Third Normal Form (3NF)**

Normalization up to 3NF helps to:

* Eliminate redundancy,
* Minimize potential anomalies in data manipulation (insertion, deletion, and update),
* Enhance data integrity and consistency.

**Normalization Process Overview**:

1. **First Normal Form (1NF)**: Ensures that each table column contains atomic (indivisible) values and each row is unique.
2. **Second Normal Form (2NF)**: Requires 1NF compliance and that each non-key attribute is fully dependent on the primary key.
3. **Third Normal Form (3NF)**: Requires 2NF compliance and that all non-key attributes are only dependent on the primary key (no transitive dependencies).

Table by Table Normalisation

**Table of Users:**

**1NF:** Every row is distinct, and every column has atomic values.

**2NF:** All columns (including UserName, Email, and CreatedAt) are directly dependent on UserID, which is the main key.

**3NF:** All non-key attributes are directly dependent on the main key; transitive dependencies are absent.

**As a result, 3NF contains the Users table.**

**Products Table:**   
**1NF:** Every row denotes a distinct product, and each column contains atomic values.

**2NF:** ProductName, Category, Price, and StockLevel are all completely dependent on ProductID, which is the major key.

**3NF:** Every non-key attribute depends only on ProductID; there are no transitive relationships.

As a result, 3NF contains the Products table.

**Table of Orders:**

**1NF:** Atomic values are in every column, and each row denotes a distinct order.

**2NF:** All columns (UserID, OrderStatus, TotalAmount, OrderDate, and PaymentMethod) rely on OrderID, which is the primary key.

**3NF:** There are no transitive dependencies because all non-key characteristics rely solely on OrderID.

**The Orders table is the outcome.**  
**OrderItems Table:**   
**1NF:** Every row is distinct and corresponds to a particular product in an order.

**2NF:** The columns OrderID, ProductID, Quantity, and Price are all dependent on the primary key, OrderItemID.

**3NF:** Since every non-key attribute is directly dependent on OrderItemID, there are no transitive dependencies.

**As a result, 3NF contains the OrderItems table.**

**CartEvents Table:  
1NF:** Atomic values are present in every column, and every event is distinct.

**2NF:** All other columns (UserID, SessionID, ProductID, EventType, and Timestamp) depend on CartEventID, which is the primary key.

**3NF:** Every non-key attribute is directly dependent on CartEventID; there are no transitive relationships.

**The CartEvents table is in 3NF as a result.**

**Product Inventory Table:**   
**1NF:** Every product entry has an atomic value and is distinct.

**2NF:** The primary key is ProductID, and every column (StockLevel,ReorderThreshold, LastUpdated) depends on it.  
**3NF:** Since every column is directly dependent on ProductID, there are no transitive dependencies.

As a result, **3NF** contains the ProductInventory table.

**Since all tables have been normalised to 3NF, they are organised to prevent data duplication and preserve excellent data integrity.**

**Connections with Foreign Keys (FK)**

By connecting relevant data across tables, foreign key constraints provide linkages between tables, guaranteeing referential integrity and making data retrieval easier.

Foreign Key Relationships Defined

**Table of Orders:**

**Connection**: The UserID foreign key connects the Orders table to the Users table.

**FK Limitation: REFERENCES FOREIGN KEY (UserID) Users (UserID).**

Justification: By creating a one-to-many relationship in which each user may have numerous orders, this relationship guarantees that each order is linked to a legitimate user in the Users database.

**Table of Order Items:   
Connections: OrderID-based links to orders**  
  
**Orders are linked to by the OrderID foreign key.**

uses the ProductID foreign key to link to products.

**FK Limitations:** Orders with foreign keys (OrderID) as references.

Products with a foreign key (ProductID) as a reference.

The OrderItems table keeps track of every product that is part of an order. Every OrderItem record makes reference to a particular order and product, guaranteeing the validity of every item in an order. With this configuration, OrderItems enables a many-to-many relationship between Orders and Products.

**CartEvents Table:**

**Relationships**: Contains links to users through the foreign key UserID.

uses the ProductID foreign key to link to products.

**FK Limitations:** USERID (FOREIGN KEY) REFERENCES USERS (UserID).

Products with a foreign key (ProductID) as a reference.

**Relationship:** Uses the ProductID foreign key to link to products.

**FK FOREIGN KEY (ProductID) REFERENCES Products (ProductID)** is a constraint.

**Justification:** Each product's stock and inventory information is tracked in the ProductInventory table. Real-time stock tracking is made possible by the relationship to Products, which guarantees that only legitimate products have inventory information.

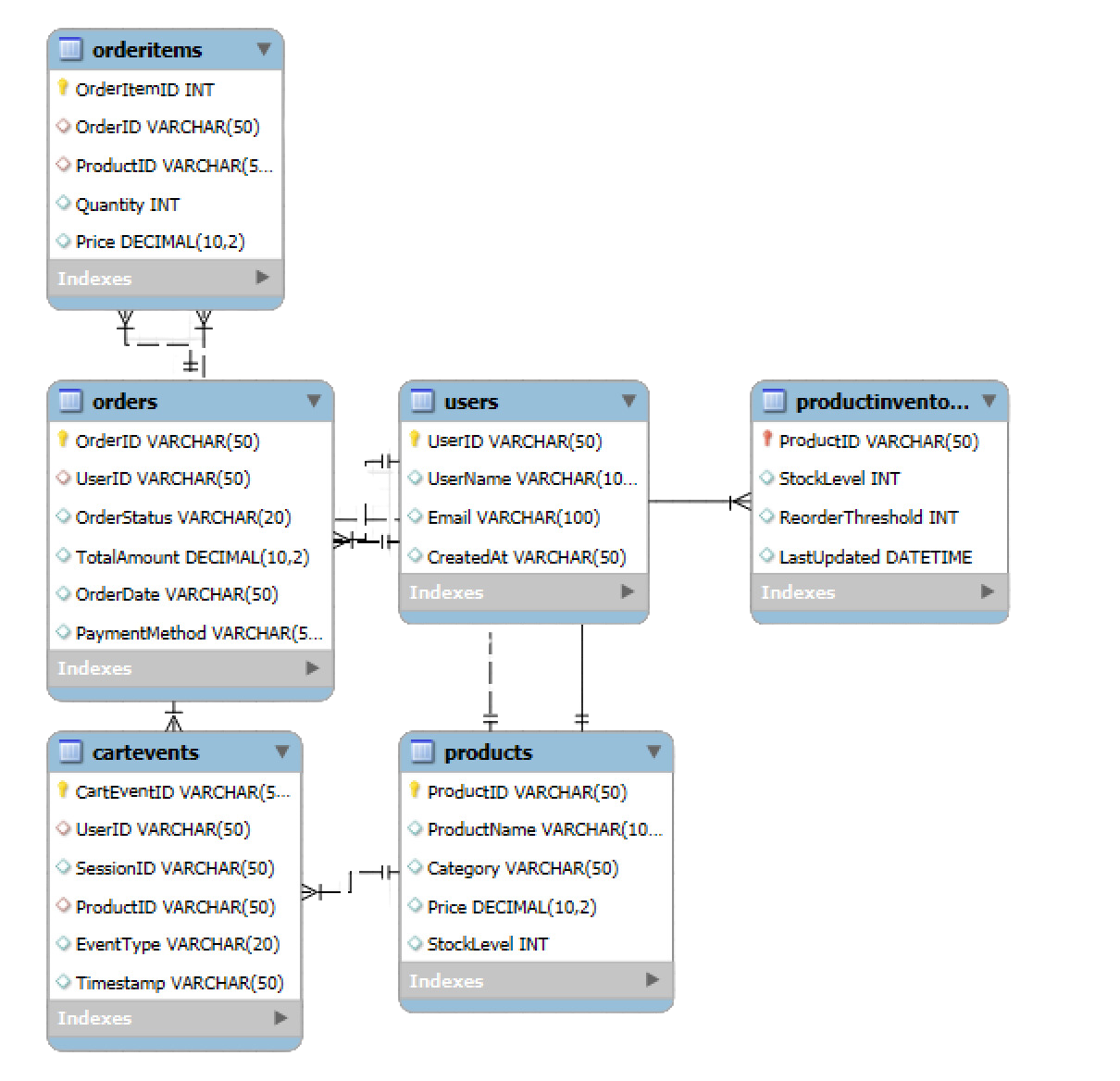
### **Summary,of.Relationships**

### **Benefits of Normalization and Relationships in This Design**

1. **Data Integrity**: With 3NF and foreign key constraints, the database maintains accurate, consistent data. Only valid user, order, and product information can be recorded, reducing data entry errors and improving data accuracy.
2. **Elimination of Redundancy**: By structuring tables in 3NF, each table contains only relevant data, minimizing duplication and reducing storage requirements. For example, user details are stored only once in the Users table, and referenced in Orders and CartEvents as needed.
3. **Improved Data Management**: Relationships allow for easy, efficient retrieval and manipulation of data. The Orders table, linked to Users through UserID, enables tracking of user order history. Similarly, OrderItems allows viewing of all products in an order and is easily expanded for order management, invoicing, and other applications.
4. **Flexibility**: This design can be easily extended if additional entities are needed, such as promotions, discounts, or shipment tracking, without restructuring existing tables, thanks to clear primary key and foreign key relationships.
5. **Optimized Queries**: Normalized tables and well-defined relationships improve query performance by reducing the amount of data that needs to be processed and making joins more efficient.

This schema provides a scalable, maintainable, and reliable structure for handling e-commerce data effectively.

# **ER Diagram**



Chapter 6 (Analysis)

# Analysis

### **Objective:** Use SQL analytical queries to enhance business performance and offer important insights for decision-making.

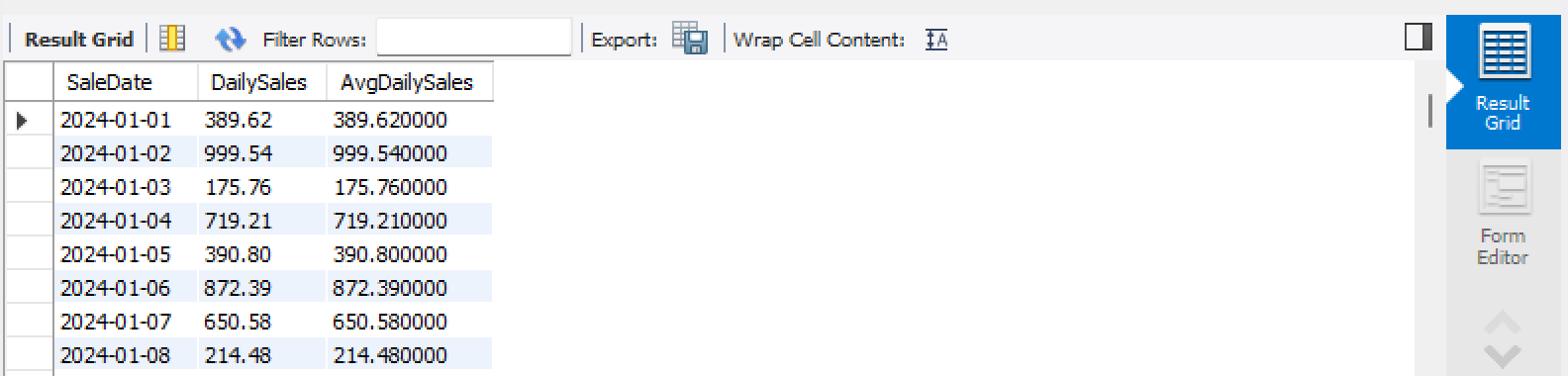
#### **1. sales analysis**

**Daily**

| SELECT DATE(OrderDate) AS SaleDate, SUM(TotalAmount) AS DailySales, AVG(TotalAmount) AS AvgDailySales FROM Orders GROUP BY DATE(OrderDate); |
| --- |

The goal is to provide the average daily sales amount as well as the daily sales totals.

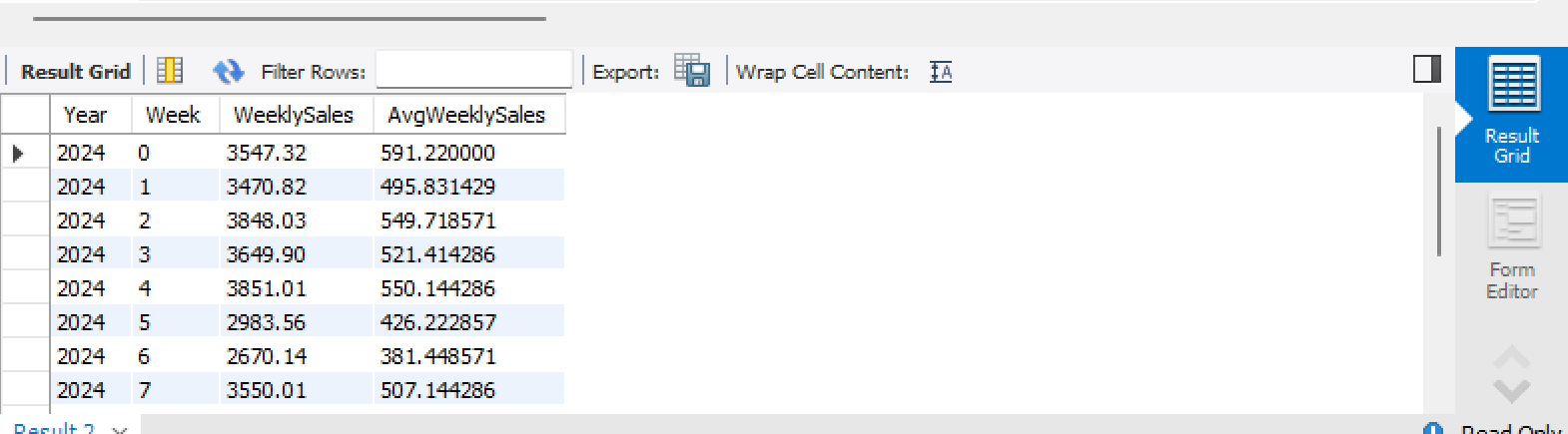
**Explanation:** Computes the daily average and total sales by grouping records according to the date of each order.

**Output:**  


**Weekly** :

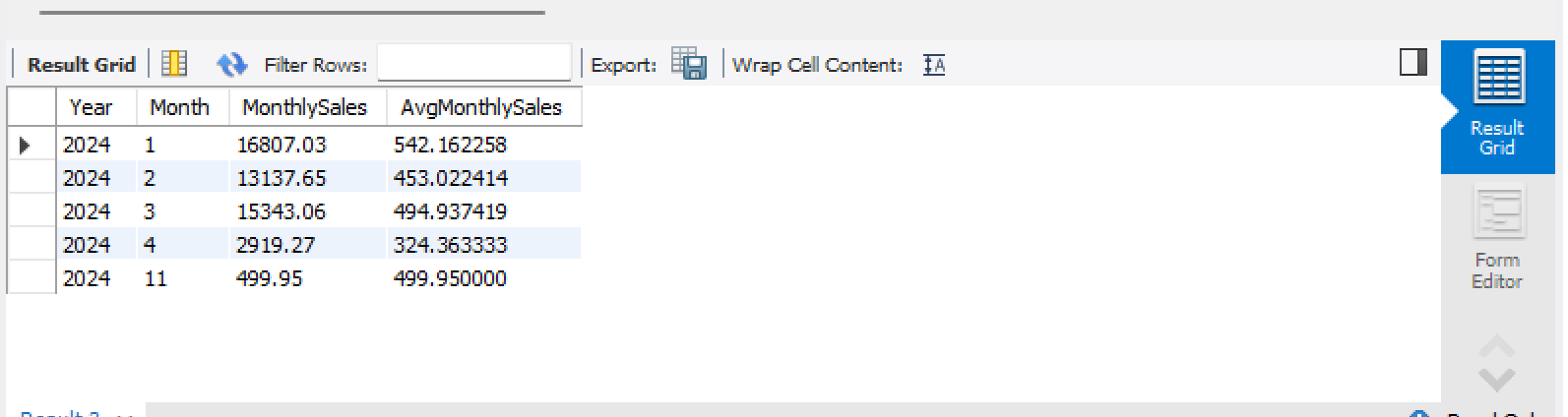
| SELECT YEAR(OrderDate) AS Year,  WEEK(OrderDate) AS Week,  SUM(TotalAmount) AS WeeklySales,  AVG(TotalAmount) AS AvgWeeklySales FROM Orders GROUP BY YEAR(OrderDate), WEEK(OrderDate); |
| --- |

**Purpose**: Provides weekly sales data, including total and average sales by week.  
**Explanation**: Groups orders by year and week, calculating weekly totals and averages for sales.

**Output:**  


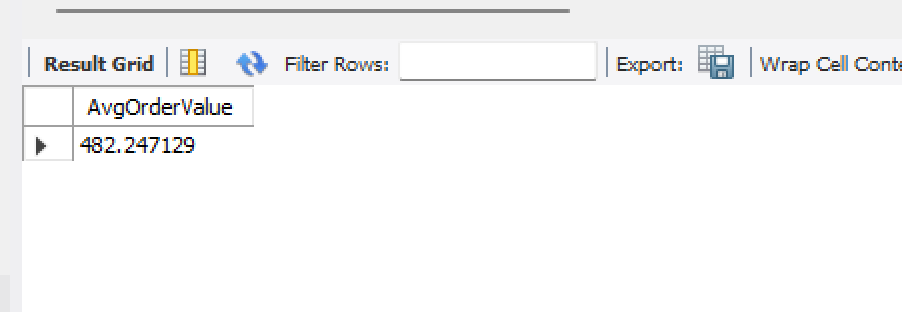
**Monthly Sales**:

| SELECT YEAR(OrderDate) AS Year, MONTH(OrderDate) AS Month, SUM(TotalAmount) AS MonthlySales, AVG(TotalAmount) AS AvgMonthlySales FROM Orders GROUP BY YEAR(OrderDate), MONTH(OrderDate); |
| --- |

**Purpose**: Provides monthly sales insights, showing total and average sales by month.  
**Explanation**: Groups records by year and month, then calculates monthly sales totals and averages.  
  
**Output :**   


**Average Order Value**:

| SELECT AVG(TotalAmount) AS AvgOrderValue FROM Orders; |
| --- |

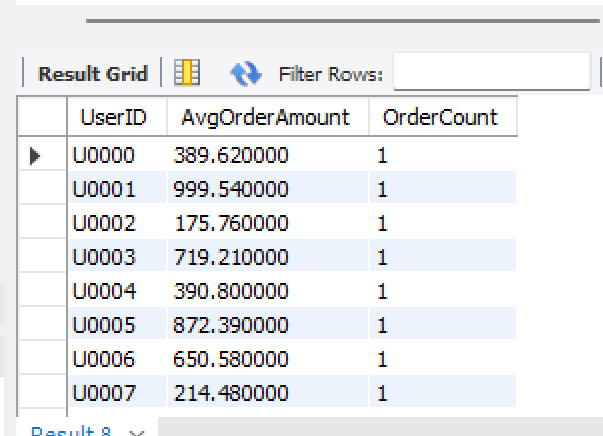
**Purpose**: Calculates the overall average order value.  
**Explanation**: Calculates the average sales value for each order across all records in the Orders table, helping assess the typical order value.  
  
**Output:**  


**2. Customer Segmentation**

This query identifies high-value customers by calculating the average order amount and order count per customer.

**High-Value Customers**:e

| SELECT UserID, AVG(TotalAmount) AS AvgOrderAmount, COUNT(OrderID) AS OrderCount FROM Orders GROUP BY UserID HAVING AVG(TotalAmount) > 70.00 LIMIT 10; |
| --- |

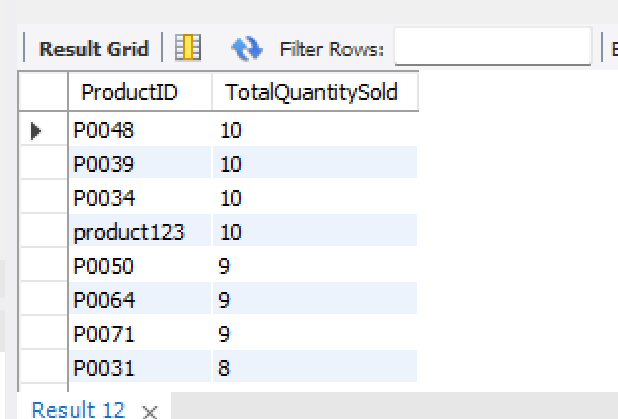
**Purpose**: Identifies high-value customers who have an average order value greater than 70 and ranks the top 10.  
**Explanation**: Groups orders by UserID to calculate each customer’s average order amount and total order count, filtered to show only customers meeting the specified average threshold.  
  
**Output:**  


#### **3. Product Performance**

These queries assess product performance by identifying top-selling products and those with low or no sales.

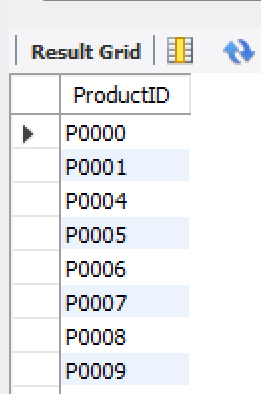
**Top-Selling Products**:

| SELECT ProductID, SUM(Quantity) AS TotalQuantitySold FROM OrderItems GROUP BY ProductID ORDER BY SUM(Quantity) DESC LIMIT 10; |
| --- |

**Purpose**: Lists the top 10 best-selling products by total quantity sold.  
**Explanation**: Groups records in OrderItems by ProductID to calculate the total quantity sold for each product, ordered in descending order of quantity.  
 **Output:  
**

**Underperforming Items**:

| SELECT p.ProductID FROM Products p LEFT JOIN OrderItems oi ON p.ProductID = oi.ProductID GROUP BY p.ProductID HAVING SUM(oi.Quantity) IS NULL OR SUM(oi.Quantity) < 5; |
| --- |

**Purpose**: Identifies products with low or no sales, helping in inventory and marketing strategies for these items.  
**Explanation**: Uses a left join between Products and OrderItems to include all products, even those with no sales records. The HAVING clause filters products with either zero sales or total sales below a threshold of 5.  
  
****  
**4. Cart Abandonment Analysis**

This query identifies products frequently added to carts but not purchased, offering insight into items with high abandonment rates.

**Products Frequently Abandoned**:

| SELECT ProductID,  COUNT(\*) AS AbandonedCount FROM CartEvents WHERE EventType = 'add' AND ProductID NOT IN ( SELECT DISTINCT ProductID FROM OrderItems  WHERE OrderID IS NOT NULL  ) GROUP BY ProductID ORDER BY COUNT(\*) DESC LIMIT 10; |
| --- |

**Purpose**: Lists the top 10 most abandoned products, indicating products that are often added to carts but not bought.  
**Explanation**: Filters CartEvents to only include 'add' events and excludes products that have been ordered. This highlights products frequently abandoned in carts, grouped by ProductID and sorted by the highest count of abandonment.

**In conclusion, each of these questions offers insightful information about the company:**

**Sales analysis:** Assists in understanding sales patterns and tracking performance across various time periods (daily, weekly, and monthly).

By identifying high-value clients, customer segmentation enables customised marketing campaigns and targeted promotions.

**Product Performance:** Informs inventory management and product planning by highlighting both successful and unsuccessful products.

Cart Abandonment Analysis: Determines which products are commonly abandoned, directing potential enhancements to pricing, incentives, or product display to lower abandonment rates.

In many facets of e-commerce management, this methodical approach to data analysis enhances business operations and facilitates data-driven decision-making.

Chapter 7 (Scheduled Reporting)

**Scheduled Reporting**

**Objective:** Generate periodic reports to track trends and ensure timely decision-making.  
  
**Automated SQL Reports**

Automated SQL reports allow periodic data collection and reporting to track various metrics such as sales, customer segmentation, and inventory.

#### **1. Daily/Weekly Sales Summaries**

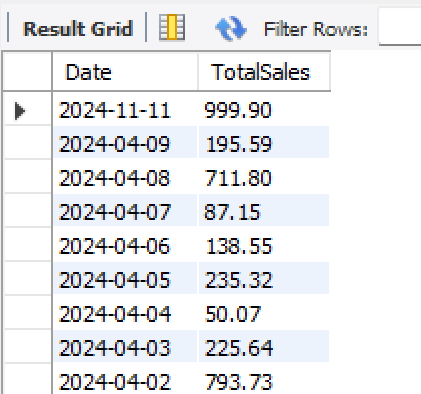
**Task:**

* Generate daily or weekly reports summarizing sales data such as total sales, sales by product, and sales by region.

**Steps:**

1. **Write SQL Query:**
   * Example of daily sales summary:

| USE ecommercedb; SHOW TABLES; SELECT DATE(OrderDate) AS Date, SUM(TotalAmount) AS TotalSales FROM orders WHERE OrderDate >= '2023-11-12' - INTERVAL 1 DAY GROUP BY Date ORDER BY Date DESC; |
| --- |

**Output:**  
  


### **Breakdown of Queries:**

**Switch to the ecommercedb Database:**

| USE ecommercedb; |
| --- |

1. This command sets the ecommercedb database as the active database for the session.

**Show All Tables:**

| SHOW TABLES; |
| --- |

This command lists all the tables in the ecommercedb database. It's useful for checking the available tables.

**Sales Summary Query:**

| SELECT DATE(OrderDate) AS Date, SUM(TotalAmount) AS TotalSales FROM orders WHERE OrderDate >= '2023-11-12' - INTERVAL 1 DAY GROUP BY Date ORDER BY Date DESC; |
| --- |

**Objective:** This query fetches the total sales (SUM(TotalAmount)) for the day before November 12, 2023, by summing up the TotalAmount column in the orders table for each day.

**Explanation:**

DATE(OrderDate) AS Date: Extracts the date part of the OrderDate field and labels it as Date.

SUM(TotalAmount) AS TotalSales: Sums up the TotalAmount field for the specified date range and labels it as TotalSales.

WHERE OrderDate >= '2023-11-12' - INTERVAL 1 DAY: Filters the data to include only the orders from the day before 2023-11-12.

GROUP BY Date: Groups the results by each day.

ORDER BY Date DESC: Orders the results by the date in descending order (latest date first).

### **Potential Issue:**

There may be a misunderstanding in the date calculation. The expression '2023-11-12' - INTERVAL 1 DAY could work, but it's generally better to use a specific date, e.g., '2023-11-11'. You might want to correct this to avoid unexpected behavior depending on the MySQL version or how it handles date calculations.

### **Example Correction:**

| WHERE OrderDate >= '2023-11-11' |
| --- |

**Create a Scheduled Job:**

In MySQL, use the EVENT scheduler to automate the execution of this query.

Example:

| CREATE EVENT daily\_sales\_summary ON SCHEDULE EVERY 1 DAY STARTS '2024-11-14 00:00:00' DO BEGIN  -- Your daily sales summary SQL query here END; |
| --- |

For weekly reports, change the interval to EVERY 1 WEEK.

#### **2. Customer Segmentation Insights**

**Task:**

* Generate customer segmentation reports based on order value and frequency.

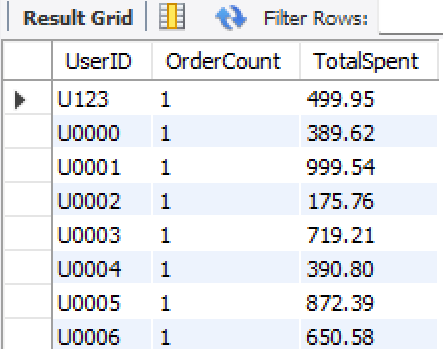
**Steps:**

1. **Write SQL Query:**

Example of customer segmentation based on frequency and total order value:

| SELECT UserID, COUNT(OrderID) AS OrderCount, SUM(TotalAmount) AS TotalSpent FROM orders GROUP BY OrderID HAVING TotalSpent > 100.00 AND OrderCount > 0; |
| --- |

**Output:**

 **Automate the Query:**

Set up an EVENT or use a cron job (for non-MySQL systems) to run this query periodically (daily/weekly).

#### **3. Inventory Status and Low-Stock Alerts**

**Task:**

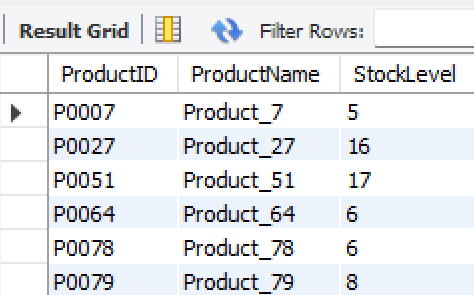
Generate inventory status reports that identify low-stock products.

**Steps:**

**Write SQL Query:**

Example:

| SELECT ProductID, ProductName, StockLevel FROM products WHERE StockLevel < 20; |
| --- |



**Schedule the Query:**

Automate this report to run daily/weekly using EVENT in MySQL or an external scheduling tool. As we do in our previous chapter using python and flask.

1. **Summarize the Process**:
   * **Step 1**: The data from the orders table is aggregated (summed up) by OrderID and inserted into the real\_time\_sales\_n table.
   * **Step 2**: A stored procedure refresh\_real\_time\_sales\_n() is created to:
     + Clear old data from real\_time\_sales\_n using TRUNCATE.
     + Re-insert updated data by summing TotalAmount grouped by OrderID.
   * **Step 3**: An event refresh\_sales\_event is set to trigger every hour, calling the refresh\_real\_time\_sales\_n() procedure to refresh data periodically.

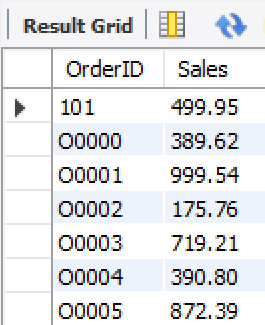
**SQL Code for the Procedure**:

| DELIMITER //  CREATE PROCEDURE refresh\_real\_time\_sales\_n() BEGIN  -- Clear old data  TRUNCATE TABLE real\_time\_sales\_n;   -- Insert updated data  INSERT INTO real\_time\_sales\_n (OrderID, Sales)  SELECT OrderID, SUM(TotalAmount) AS Sales  FROM orders  GROUP BY OrderID; END; |
| --- |

**Create the Event**: sql CREATE EVENT refresh\_sales\_event ON SCHEDULE EVERY 1 HOUR DO CALL refresh\_real\_time\_sales\_n();

**Check Event Scheduler Status**:

| SHOW VARIABLES LIKE 'event\_scheduler'; |
| --- |



**Create a View to Monitor Sales**:

| CREATE VIEW sales\_report AS SELECT \* FROM real\_time\_sales\_n; |
| --- |

**Output Data Every Hour**: To see the results of the hourly refresh, use the following

| SELECT \* FROM sales\_report; |
| --- |

This format summarizes the process, creates the stored procedure and event, and sets up a view to display the updated sales data every hour.