Databases Project Report

Module: Advanced Databases (SQLI12025)

Assessment: Database Design & SQL for Data Analysis

Consultant: Agbaakin Oluwatosin

Date: October 4, 2025

Table of Contents

Part 1: Database Design and Implementation

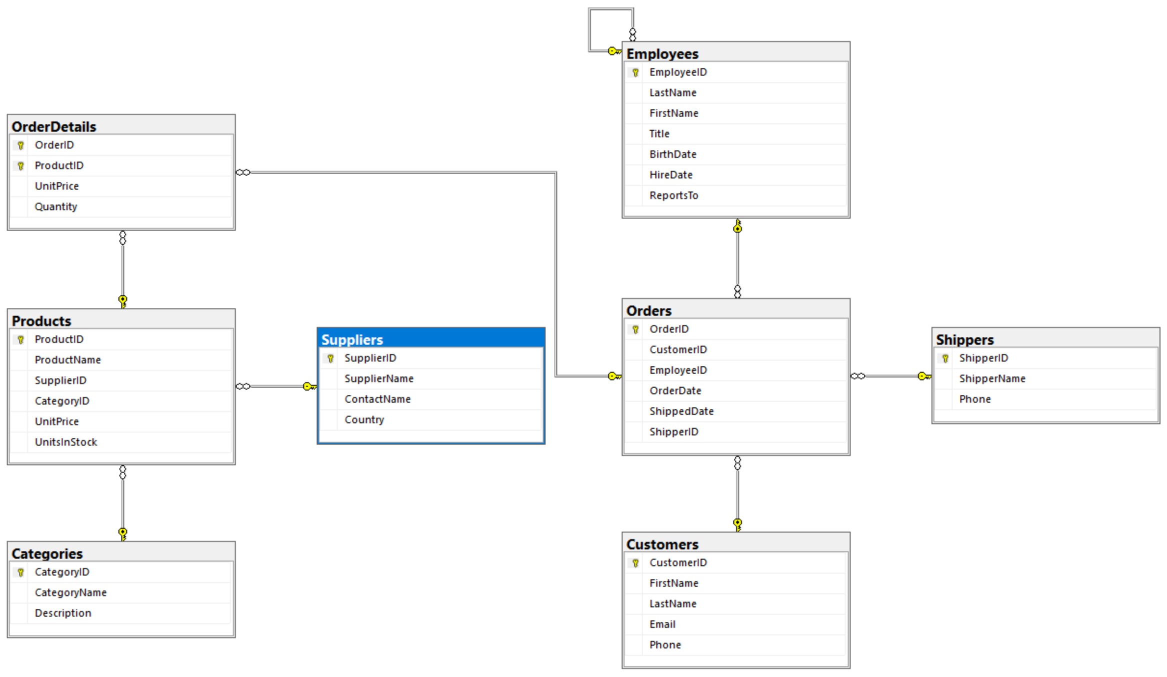
1.1 Database Design and Normalization

The design of the EcommerceDB was guided by the principles of database normalization to ensure data integrity, eliminate redundancy, and create a flexible, scalable schema. The process to achieve Third Normal Form (3NF) was as follows:

- **First Normal Form (1NF):** The initial step involved ensuring all tables have a primary key and that all column values are atomic. For instance, an order with multiple products is not stored in a single record; it is broken down into individual line items in the OrderDetails table, each linked to a single product.
- **Second Normal Form (2NF):** This step eliminates partial dependencies. All non-key attributes in tables with composite primary keys must depend on the entire key. This is demonstrated in the OrderDetails table, where Quantity is dependent on both the OrderID and the ProductID together, not just one part of the key.
- **Third Normal Form (3NF):** The final step eliminates transitive dependencies. This means that no non-key attribute is dependent on another non-key attribute. For example, SupplierName is stored in the Suppliers table and is linked to a product via SupplierID. Storing the supplier's name directly in the Products table would be a transitive dependency, leading to data anomalies if a supplier's name were to change. The final schema adheres strictly to 3NF.

1.2 Entity-Relationship Diagram

The resulting schema consists of seven interconnected tables that logically separate the core business entities. The diagram below illustrates these tables, their columns, and the relationships between them.



Screenshot of my Entity-Relationship Diagram

1.3 Table Implementation, Data Types, and Constraints

The database tables were implemented using T-SQL CREATE TABLE statements. Data types were selected to optimize for storage and data validity. For example, INT is used for keys, NVARCHAR for text, and DECIMAL(10, 2) for currency to maintain precision.

Data integrity is enforced programmatically through constraints. Task 2 of the brief requires a constraint to ensure a product's price is positive. This was implemented directly in the Products table definition:

Code:

```sql
CREATE TABLE dbo.Products (
    ProductID INT IDENTITY(1,1) PRIMARY KEY,
    -- ... other columns
    UnitPrice DECIMAL(10, 2) NOT NULL,
    CONSTRAINT CK_Products_UnitPrice CHECK (UnitPrice > 0)
);
```

The CHECK constraint CK_Products_UnitPrice automatically rejects any INSERT or UPDATE operation where the UnitPrice is not greater than zero, ensuring business rules are enforced at the database level.

1.4 Data Population

All tables were populated with a sufficient number of records (at least seven where appropriate) to allow for meaningful testing of all required queries and objects. The sample data was specifically crafted to ensure test cases existed for every task, such as creating customers older than 40 and orders with 'Premium' products.

Part 2: T-SQL Object Development and Queries

2.1 Task 3: Query for Customers Older Than 40

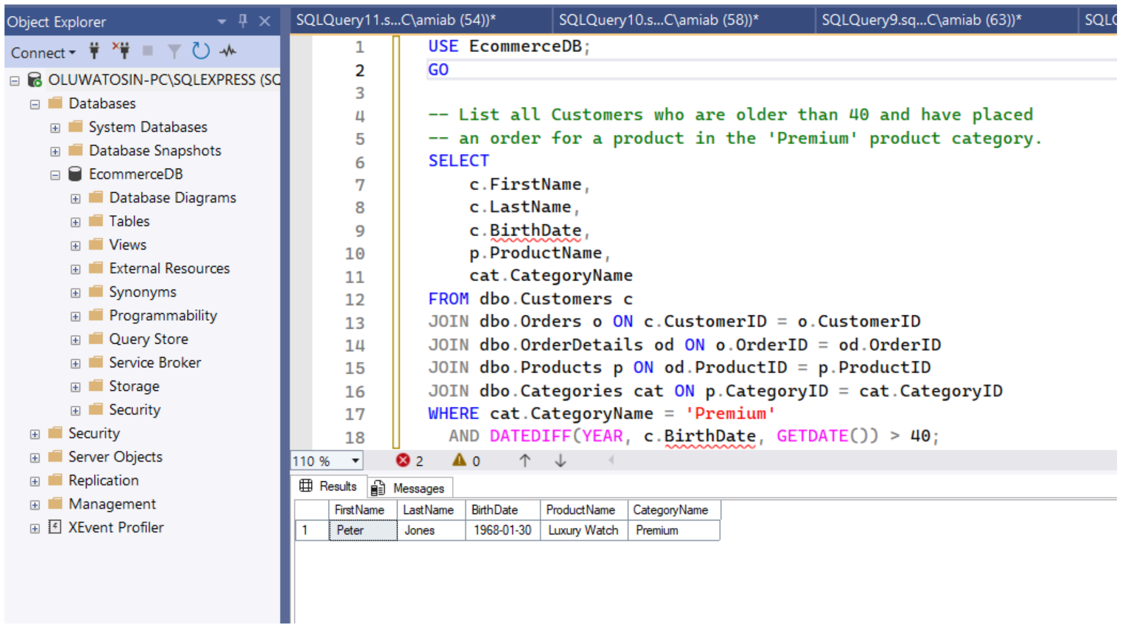
Requirement: List all customers older than 40 who have placed an order for a product in the 'Premium' category.

Code and Execution:

```sql
USE EcommerceDB;
GO

-- List all Customers who are older than 40 and have placed
-- an order for a product in the 'Premium' product category.
SELECT
    c.FirstName,
    c.LastName,
    c.BirthDate,
    p.ProductName,
    cat.CategoryName
FROM dbo.Customers c
JOIN dbo.Orders o ON c.CustomerID = o.CustomerID
JOIN dbo.OrderDetails od ON o.OrderID = od.OrderID
JOIN dbo.Products p ON od.ProductID = p.ProductID
JOIN dbo.Categories cat ON p.CategoryID = cat.CategoryID
WHERE cat.CategoryName = 'Premium'
  AND DATEDIFF(YEAR, c.BirthDate, GETDATE()) > 40;
```

Results:



```sql
USE EcommerceDB;
GO

-- List all Customers who are older than 40 and have placed
-- an order for a product in the 'Premium' product category.
SELECT
    c.FirstName,
    c.LastName,
    c.BirthDate,
    p.ProductName,
    cat.CategoryName
FROM dbo.Customers c
JOIN dbo.Orders o ON c.CustomerID = o.CustomerID
JOIN dbo.OrderDetails od ON o.OrderID = od.OrderID
JOIN dbo.Products p ON od.ProductID = p.ProductID
JOIN dbo.Categories cat ON p.CategoryID = cat.CategoryID
WHERE cat.CategoryName = 'Premium'
  AND DATEDIFF(YEAR, c.BirthDate, GETDATE()) > 40;
```

	FirstName	LastName	BirthDate	ProductName	CategoryName
1	Peter	Jones	1968-01-30	Luxury Watch	Premium

Explanation: This query joins five tables to link customers to the products they ordered. It filters the results to only include orders where the product's category is 'Premium' and uses the DATEDIFF function to calculate the customer's current age, ensuring it is greater than 40.

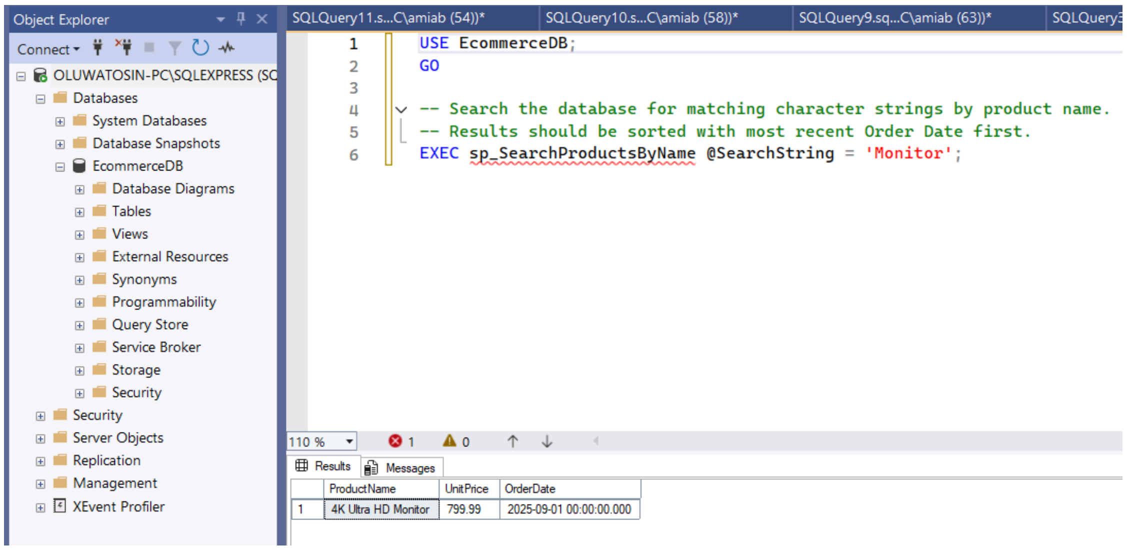
2.2 Task 4a: Stored Procedure to Search Products

Requirement: Create a stored procedure to search for products by name, sorted by most recent order date.

Code and Execution:

```sql
-- Calling the procedure
EXEC sp_SearchProductsByName @SearchString = 'Monitor';
```

Results:



```sql
USE EcommerceDB;
GO

-- Search the database for matching character strings by product name.
-- Results should be sorted with most recent Order Date first.
EXEC sp_SearchProductsByName @SearchString = 'Monitor';
```

	ProductName	UnitPrice	OrderDate
1	4K Ultra HD Monitor	799.99	2025-09-01 00:00:00.000

Explanation: The sp_SearchProductsByName procedure accepts a search string. It uses the LIKE operator to find any product whose name contains the string. The results are joined with order data and sorted by OrderDate in descending order.

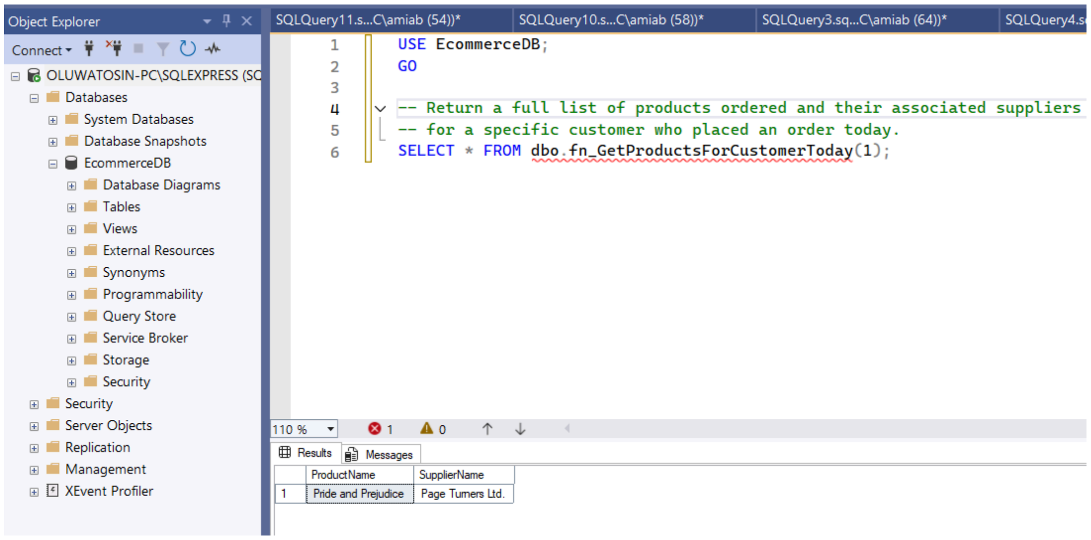
2.3 Task 4b: Function for Today's Orders

Requirement: Return a list of products and suppliers for a specific customer's orders placed today.

Code and Execution:

```sql
-- Calling the function
SELECT * FROM dbo.fn_GetProductsForCustomerToday(1);
```

Results:



```sql
USE EcommerceDB;
GO

-- Return a full list of products ordered and their associated suppliers
-- for a specific customer who placed an order today.
SELECT * FROM dbo.fn_GetProductsForCustomerToday(1);
```

	ProductName	SupplierName
1	Pride and Prejudice	Page Turners Ltd.

Explanation: This inline table-valued function takes a CustomerID as input. It filters the Orders table for records where the order date matches the current system date (GETDATE()). It returns a table of product and supplier names for that customer's orders today.

2.4 Task 4c: Stored Procedure to Update a Supplier

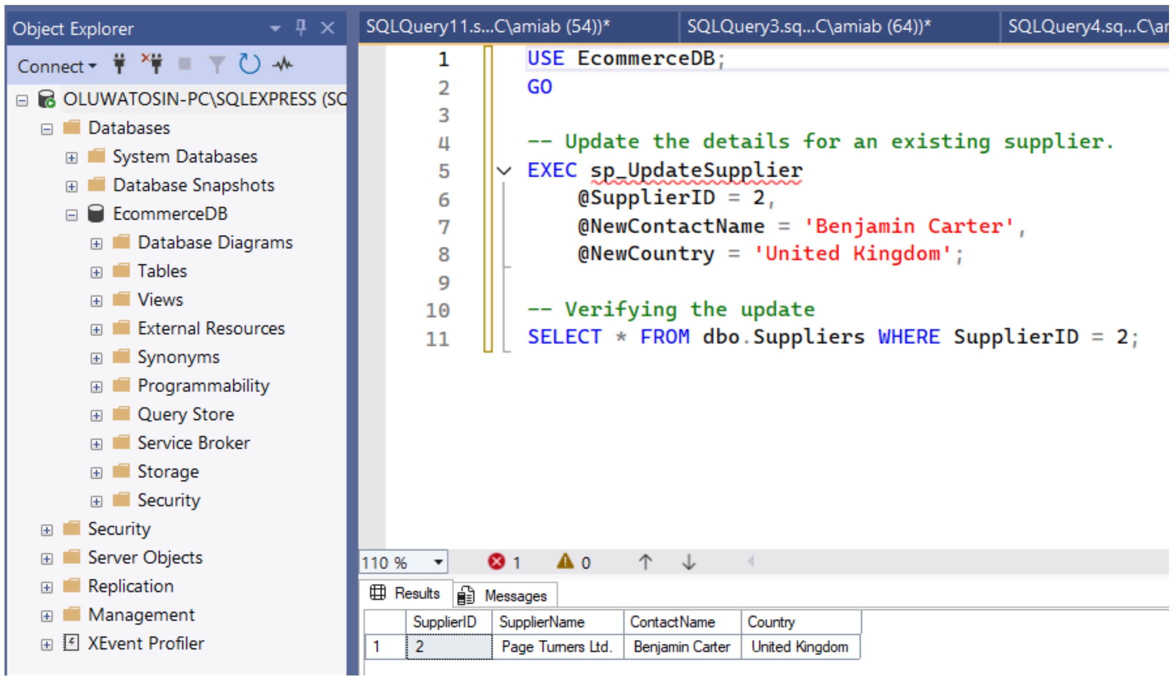
Requirement: Update the details for an existing supplier.

Code and Execution:

```
-- Calling the procedure
EXEC sp_UpdateSupplier @SupplierID = 2, @NewContactName = 'Benjamin Carter',
@NewCountry = 'United Kingdom';

-- Verifying the update
SELECT * FROM dbo.Suppliers WHERE SupplierID = 2;
```

Results:



```
USE EcommerceDB;
GO

-- Update the details for an existing supplier.
EXEC sp_UpdateSupplier
    @SupplierID = 2,
    @NewContactName = 'Benjamin Carter',
    @NewCountry = 'United Kingdom';

-- Verifying the update
SELECT * FROM dbo.Suppliers WHERE SupplierID = 2;
```

	SupplierID	SupplierName	ContactName	Country
1	2	Page Turners Ltd.	Benjamin Carter	United Kingdom

Explanation: The sp_UpdateSupplier procedure takes the SupplierID and the new details as parameters. It performs an UPDATE operation on the Suppliers table. The subsequent SELECT statement verifies that the changes were successfully applied.

2.5 Task 4d: Stored Procedure to Delete a Delivered Order

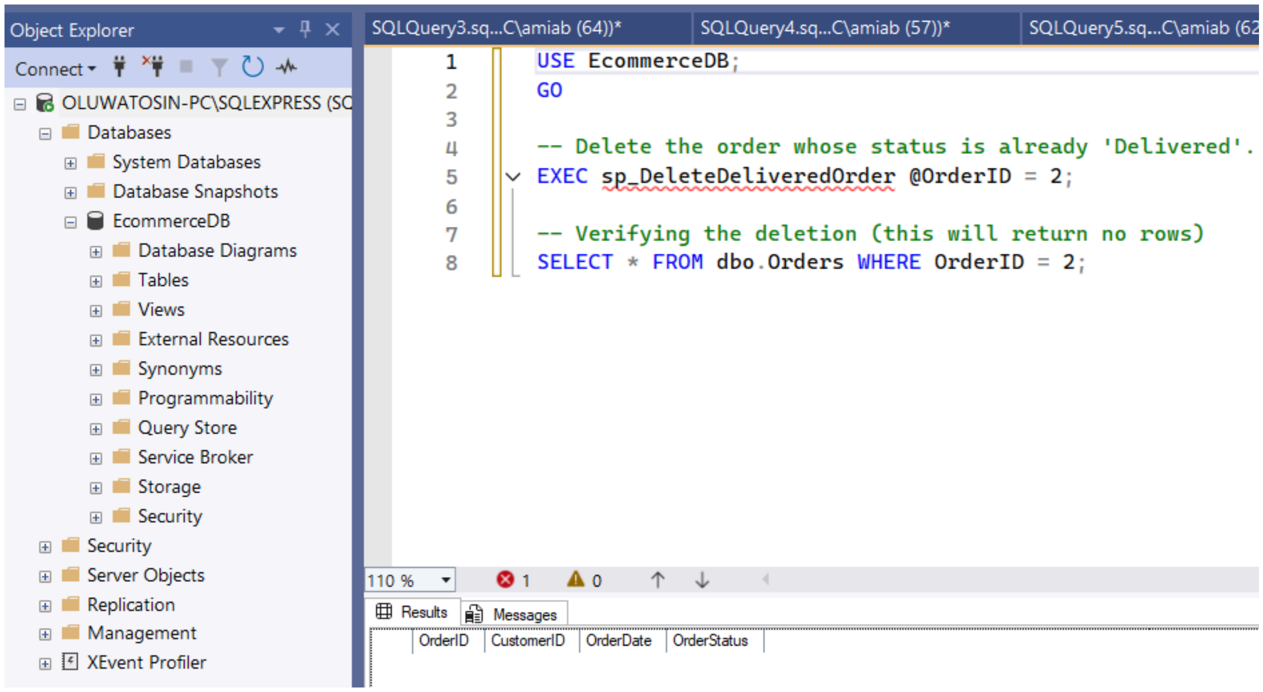
Requirement: Delete an order whose status is 'Delivered'.

Code and Execution:

```
-- Calling the procedure to delete OrderID 2
EXEC sp_DeleteDeliveredOrder @OrderID = 2;

-- Verifying the deletion
SELECT * FROM dbo.Orders WHERE OrderID = 2;
```

Results:



```
USE EcommerceDB;
GO

-- Delete the order whose status is already 'Delivered'.
EXEC sp_DeleteDeliveredOrder @OrderID = 2;

-- Verifying the deletion (this will return no rows)
SELECT * FROM dbo.Orders WHERE OrderID = 2;
```

OrderID	CustomerID	OrderDate	OrderStatus

Explanation: This procedure first checks if the specified order exists and has a status of 'Delivered'. To maintain referential integrity, it first deletes the child records from OrderDetails before deleting the parent record from the Orders table.

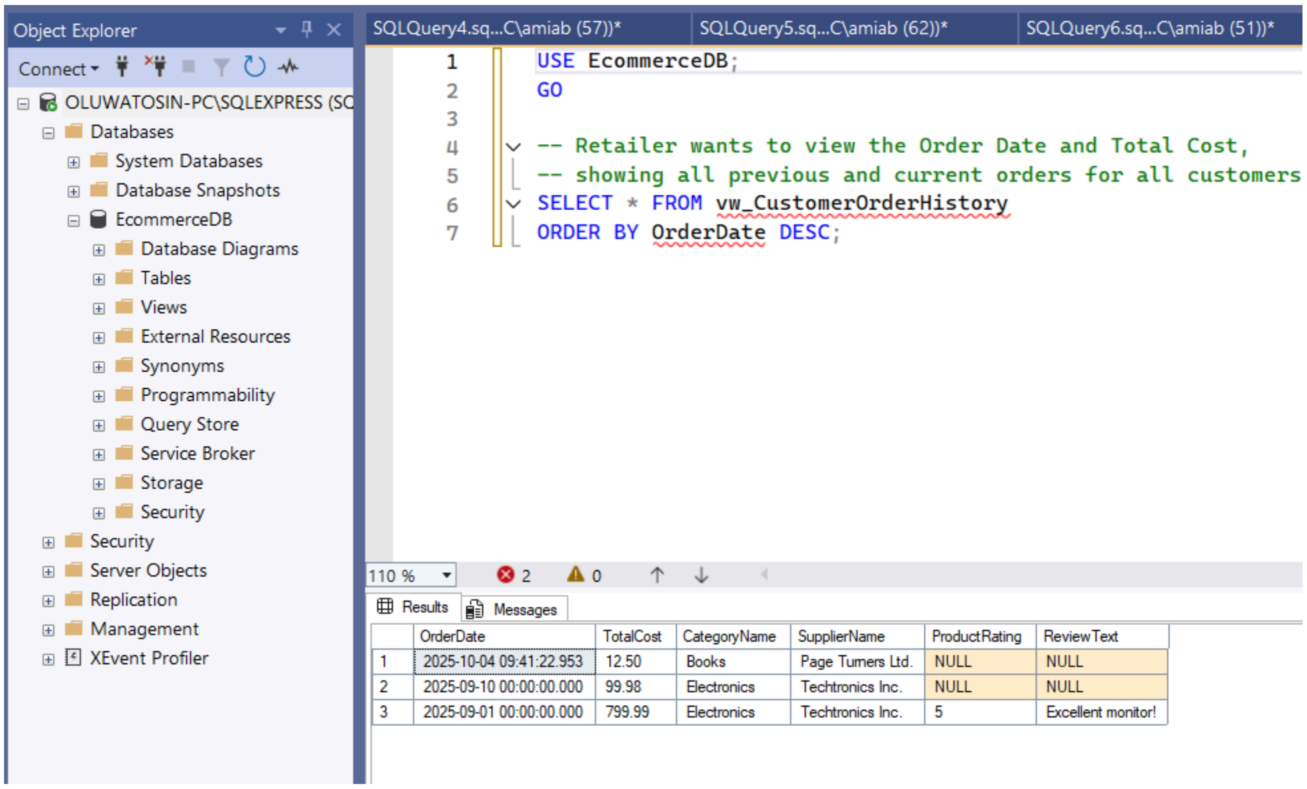
2.6 Task 5: View for Order History

Requirement: Create a view showing Order Date, Total Cost, Product Category, Supplier Name, and any associated Review/Rating.

Code and Execution:

```
-- Calling the view
SELECT * FROM vw_CustomerOrderHistory
ORDER BY OrderDate DESC;
```

Results:



```
USE EcommerceDB;
GO

-- Retailer wants to view the Order Date and Total Cost,
-- showing all previous and current orders for all customers
SELECT * FROM vw_CustomerOrderHistory
ORDER BY OrderDate DESC;
```

	OrderDate	TotalCost	CategoryName	SupplierName	ProductRating	ReviewText
1	2025-10-04 09:41:22.953	12.50	Books	Page Turners Ltd.	NULL	NULL
2	2025-09-10 00:00:00.000	99.98	Electronics	Techtronics Inc.	NULL	NULL
3	2025-09-01 00:00:00.000	799.99	Electronics	Techtronics Inc.	5	Excellent monitor!

Explanation: The vw_CustomerOrderHistory view joins six tables to consolidate all the required information into a single, queryable object. A LEFT JOIN is used for the Reviews table to ensure that orders for products that have not yet been reviewed are still included in the results.

2.7 Task 6: Trigger to Restock Inventory

Requirement: When an order is cancelled, the inventory stock level for the product(s) must be increased.

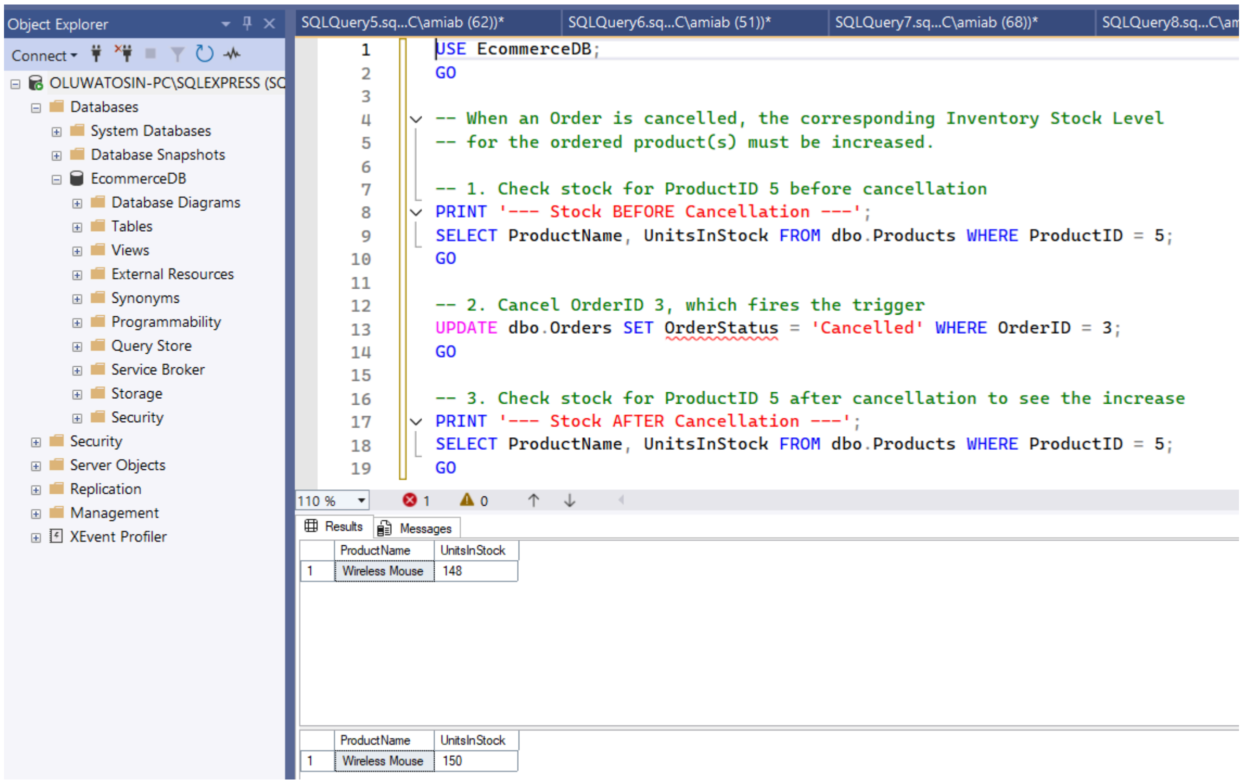
Code and Execution:

```sql
-- Check stock for ProductID 5 before cancellation
SELECT ProductName, UnitsInStock FROM dbo.Products WHERE ProductID = 5;

-- Cancel OrderID 3
UPDATE dbo.Orders SET OrderStatus = 'Cancelled' WHERE OrderID = 3;

-- Check stock for ProductID 5 after cancellation
SELECT ProductName, UnitsInStock FROM dbo.Products WHERE ProductID = 5;
```

Results:



```
USE EcommerceDB;
GO

-- When an Order is cancelled, the corresponding Inventory Stock Level
-- for the ordered product(s) must be increased.

-- 1. Check stock for ProductID 5 before cancellation
PRINT '--- Stock BEFORE Cancellation ---';
SELECT ProductName, UnitsInStock FROM dbo.Products WHERE ProductID = 5;
GO

-- 2. Cancel OrderID 3, which fires the trigger
UPDATE dbo.Orders SET OrderStatus = 'Cancelled' WHERE OrderID = 3;
GO

-- 3. Check stock for ProductID 5 after cancellation to see the increase
PRINT '--- Stock AFTER Cancellation ---';
SELECT ProductName, UnitsInStock FROM dbo.Products WHERE ProductID = 5;
GO
```

	ProductName	UnitsInStock
1	Wireless Mouse	148

	ProductName	UnitsInStock
1	Wireless Mouse	150

Explanation: The trg_RestockOnCancel trigger fires automatically after an UPDATE on the Orders table. It checks if the OrderStatus was changed to 'Cancelled'. If so, it finds all products associated with that order in OrderDetails and adds their quantities back to the UnitsInStock in the Products table.

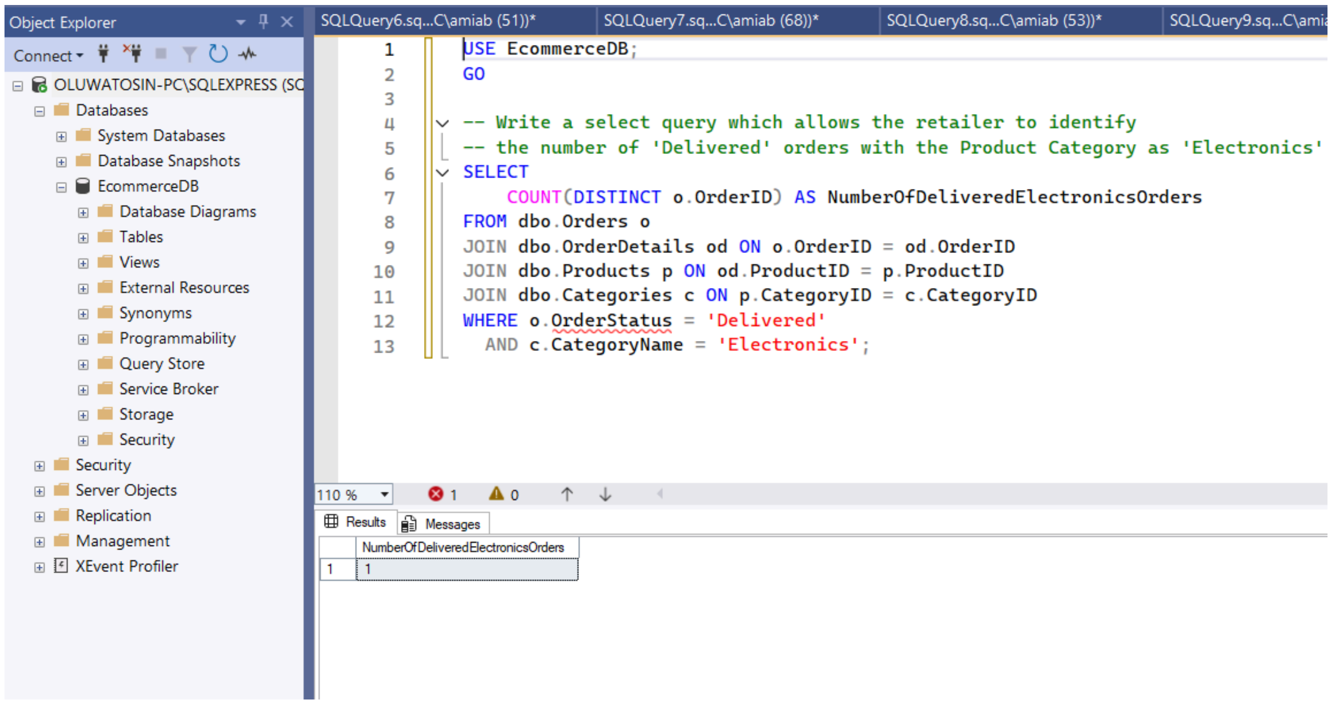
2.8 Task 7: Query for Delivered Electronics Orders

Requirement: Identify the number of 'Delivered' orders with the Product Category as 'Electronics'.

Code and Execution:

```
SELECT
    COUNT(DISTINCT o.OrderID) AS NumberOfDeliveredElectronicsOrders
FROM dbo.Orders o
JOIN dbo.OrderDetails od ON o.OrderID = od.OrderID
JOIN dbo.Products p ON od.ProductID = p.ProductID
JOIN dbo.Categories c ON p.CategoryID = c.CategoryID
WHERE o.OrderStatus = 'Delivered'
  AND c.CategoryName = 'Electronics';
```

Results:



```sql
USE EcommerceDB;
GO

-- Write a select query which allows the retailer to identify
-- the number of 'Delivered' orders with the Product Category as 'Electronics'
SELECT
    COUNT(DISTINCT o.OrderID) AS NumberOfDeliveredElectronicsOrders
FROM dbo.Orders o
JOIN dbo.OrderDetails od ON o.OrderID = od.OrderID
JOIN dbo.Products p ON od.ProductID = p.ProductID
JOIN dbo.Categories c ON p.CategoryID = c.CategoryID
WHERE o.OrderStatus = 'Delivered'
  AND c.CategoryName = 'Electronics';
```

```
   NumberOfDeliveredElectronicsOrders
1  1
```

Explanation: This query counts the distinct OrderIDs to ensure that orders with multiple electronics are only counted once. It joins the necessary tables and filters for records where the OrderStatus is 'Delivered' and the CategoryName is 'Electronics'.

Part 3: Strategic Recommendations

3.1 Data Integrity and Concurrency

Data Integrity: This database design ensures integrity through the strict enforcement of constraints: PRIMARY KEYs guarantee uniqueness, FOREIGN KEYs maintain referential integrity, and CHECK constraints enforce business rules like positive prices and valid order statuses.

Concurrency: In a high-traffic e-commerce system, managing concurrent transactions is critical. The primary risk is two customers buying the last item in stock simultaneously. This is managed using transactions with an appropriate isolation level. The sp_PlaceNewOrder procedure wraps the stock check and stock reduction logic in a single atomic transaction. This ensures that one transaction will complete fully before another can access the same data, preventing stock from being oversold.

3.2 Database Security

A role-based security model was implemented, adhering to the principle of least privilege. Two custom roles, DataAnalyst and OrderEntryClerk, are created. The DataAnalyst has read-only access to specific views for reporting. Critically, the OrderEntryClerk does not have direct access to tables; instead, they are only granted EXECUTE permissions on stored procedures. This approach provides a secure interface for data modification, prevents accidental data corruption, and significantly reduces the risk of SQL injection attacks.

3.3 Database Backup and Recovery

For a mission-critical e-commerce database, a robust backup and recovery strategy is non-negotiable. The following strategy is recommended:

- **Recovery Model:** The database must be set to the **Full Recovery Model**. This logs all transactions and is essential for point-in-time recovery, which is crucial for minimizing data loss in a transactional system.
- **Backup Schedule:**
  - **Weekly Full Backups:** A complete copy of the database should be taken once a week during off-peak hours.
  - **Daily Differential Backups:** These capture all changes made since the last full backup and are much faster to create than a full backup.
  - **Frequent Transaction Log Backups:** Log backups should be taken every 15-30 minutes. This is the key to minimizing data loss, as it allows the database to be restored to a state just moments before a failure occurred.

This multi-layered strategy ensures that in the event of a system failure, the business can recover quickly with minimal data loss.

4. Conclusion

The EcommerceDB database has been successfully designed, normalized, and implemented to meet all specified requirements. The use of advanced T-SQL objects like stored procedures and triggers provides a secure, efficient, and automated system for managing the retailer's data. The strategic recommendations offered provide a clear framework for maintaining data integrity, security, and availability as the business grows.