ONLINE SHOPPING DATA

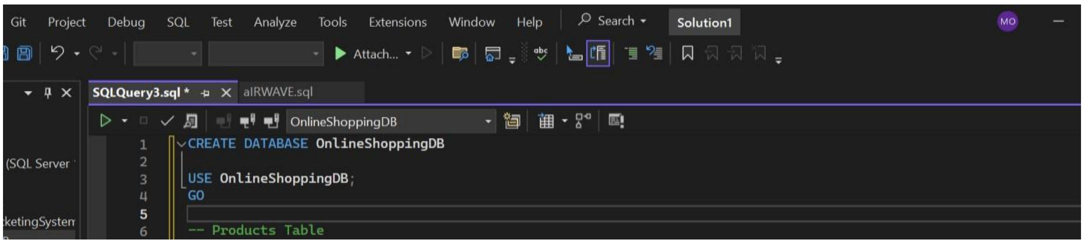
• Introduction

This report presents the design and querying of a database system for online shopping platform using SQL Server. Working with five CSV files customers, products, orders, order items, and payments. I created the OnlineShoppingDB and established table relationships with proper constraints. The task involved writing SQL queries to extract insights, update records, and create stored procedures, views, and functions. Each solution is backed by explanation and outputs, showing how structured data can be transformed into meaningful business insights.

.1 Database Creation (OnlineShoppingDB)

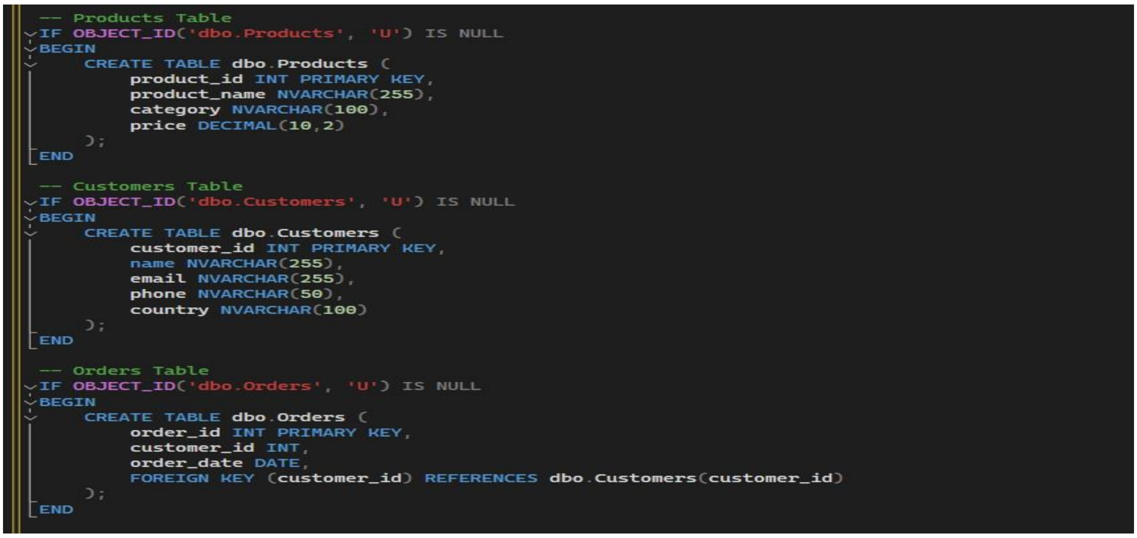
Before building the database, it's important to understand the dataset and its purpose. To help with this, think about these simple questions:

- How can you organize the entities into separate tables?
- What kind of relationships exist between the entities—one-to-one, one-to-many, or many-to-many?
- What are the main entities in the dataset?
- How do these entities relate to each other?

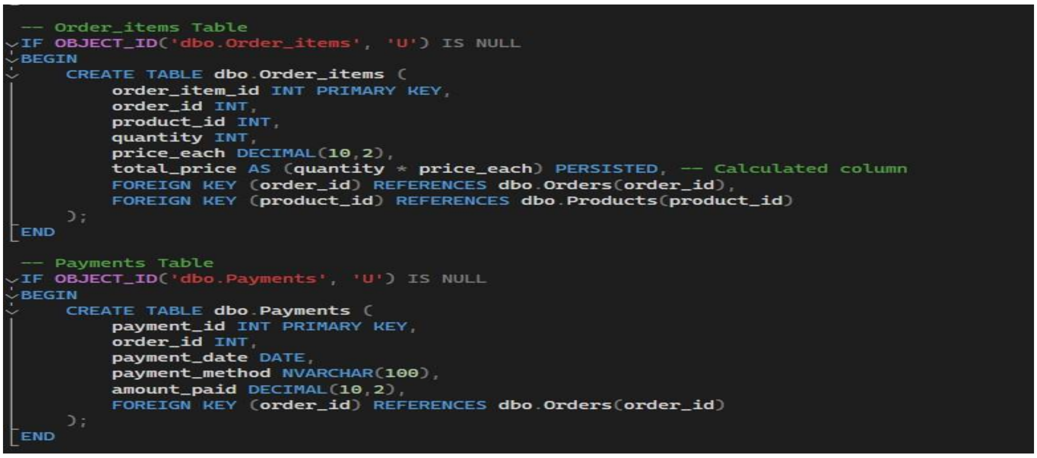


```
CREATE DATABASE OnlineShoppingDB

USE OnlineShoppingDB;
GO

-- Products Table
```

After creating the database, I proceeded to create the respective tables before importing the csv files containing data for the individual tables



```
-- Products Table
IF OBJECT_ID('dbo.Products', 'U') IS NULL
BEGIN
    CREATE TABLE dbo.Products (
        product_id INT PRIMARY KEY,
        product_name NVARCHAR(255),
        category NVARCHAR(100),
        price DECIMAL(10,2)
    );
END

-- Customers Table
IF OBJECT_ID('dbo.Customers', 'U') IS NULL
BEGIN
    CREATE TABLE dbo.Customers (
        customer_id INT PRIMARY KEY,
        name NVARCHAR(255),
        email NVARCHAR(255),
        phone NVARCHAR(50),
        country NVARCHAR(100)
    );
END

-- Orders Table
IF OBJECT_ID('dbo.Orders', 'U') IS NULL
BEGIN
    CREATE TABLE dbo.Orders (
        order_id INT PRIMARY KEY,
        customer_id INT,
        order_date DATE,
        FOREIGN KEY (customer_id) REFERENCES dbo.Customers(customer_id)
    );
END
```



```
-- Order_items Table
IF OBJECT_ID('dbo.Order_items', 'U') IS NULL
BEGIN
    CREATE TABLE dbo.Order_items (
        order_item_id INT PRIMARY KEY,
        order_id INT,
        product_id INT,
        quantity INT,
        price_each DECIMAL(10,2),
        total_price AS (quantity * price_each) PERSISTED, -- Calculated column
        FOREIGN KEY (order_id) REFERENCES dbo.Orders(order_id),
        FOREIGN KEY (product_id) REFERENCES dbo.Products(product_id)
    );
END

-- Payments Table
IF OBJECT_ID('dbo.Payments', 'U') IS NULL
BEGIN
    CREATE TABLE dbo.Payments (
        payment_id INT PRIMARY KEY,
        order_id INT,
        payment_date DATE,
        payment_method NVARCHAR(100),
        amount_paid DECIMAL(10,2),
        FOREIGN KEY (order_id) REFERENCES dbo.Orders(order_id)
    );
END
```

- *Table Adjustments*

On reviewing the data in the respective files certain adjustments were made in respect of the data types which were incorporated at the time of creating the tables.

Customers Table

- **customer_id** is an INT for a simple numeric ID.
- **name**, **email**, **phone**, and **country** are NVARCHAR to store text with varying lengths and support different languages and special characters.

Products Table

- **product_id** is an INT to uniquely identify each product.
- **product_name** and **category** are NVARCHAR to store descriptive text.
- **price** is DECIMAL(10,2) for accurate currency representation.

Orders Table

- **order_id** and **customer_id** are INT for IDs.
- **order_date** uses the DATE type to store just the date of the order.

Order_items Table

- **order_item_id**, **order_id**, and **product_id** are INT as they are identifiers.
- **quantity** is INT because you can't order partial items.
- **price_each**, **Total_price**, and **total_amount** are DECIMAL(10,2) to handle monetary values accurately.

Payments Table

- **payment_id** and **order_id** are INT for identifiers.
- **payment_date** uses the DATE type to record the transaction date.
- **payment_method** is NVARCHAR to store different text-based payment options.
- **amount_paid** is DECIMAL(10,2) for accurate currency storage without rounding errors.

.2 Importing CSV files

Working with Visual Studio Enterprise I load the data into the created tables using BULK INSERT code used to import large data files into SQL Server.

BULK INSERT 'Table name'

This tells SQL Server to load data into the named table in the dbo schema.

FROM 'C:\’Path’\’filename’

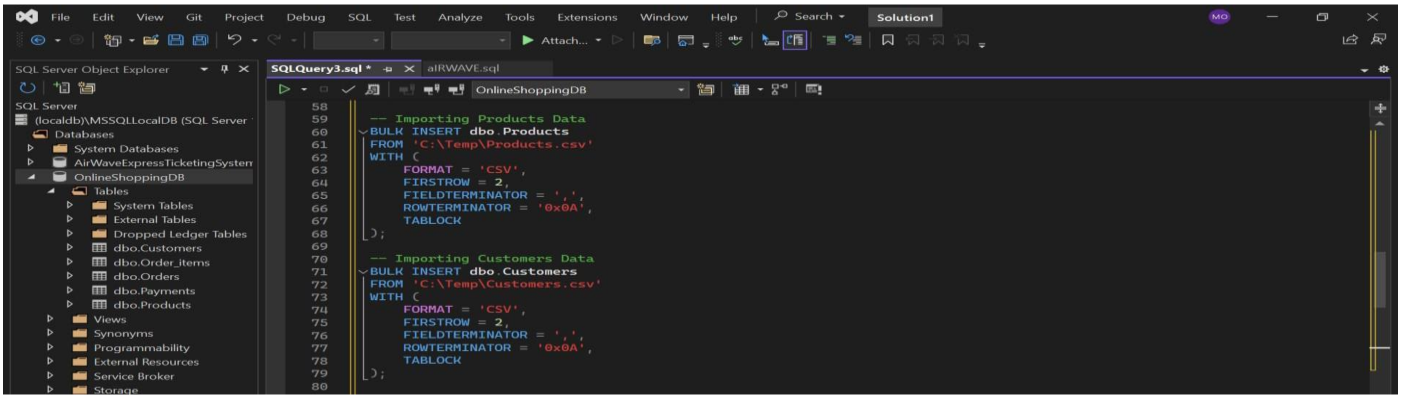
It specifies the full path to the CSV file on your PC. This must be a folder that SQL Server has permission to access.

Tip: It is recommended to move files to a neutral location like C:\Temp before importing.

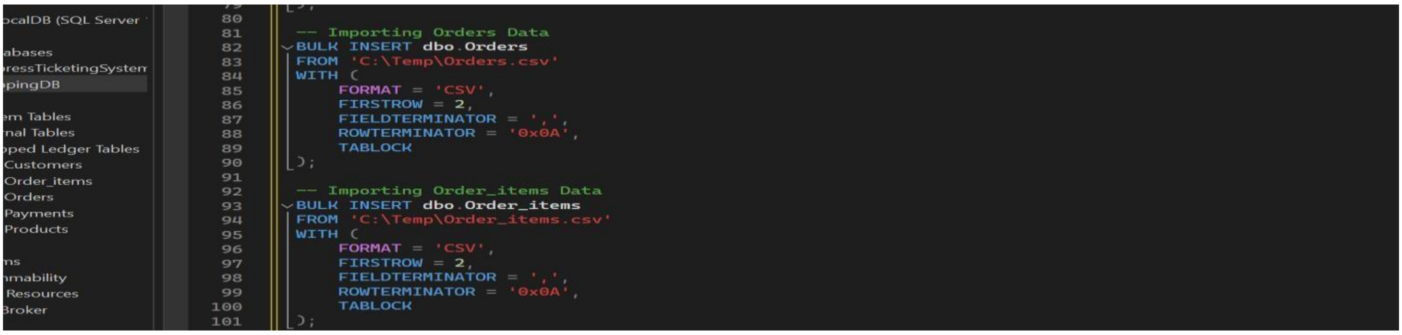
WITH (...)

This clause sets options for how the file is read.

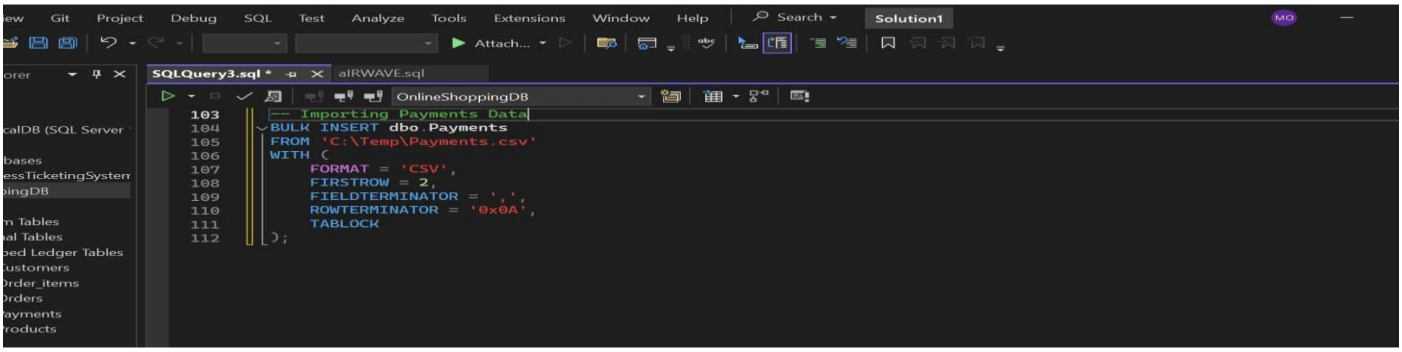
FORMAT = 'CSV'	Declares that the file is in CSV format (important for SQL 2022+).
FIRSTROW = 2	Skips the header row; starts inserting from the second line.
FIELDTERMINATOR = ','	Defines that fields in each row are separated by commas.
ROWTERMINATOR = '0x0A'	Defines a line break (LF) at the end of each row (Unix-style).
TABLOCK	Applies a table-level lock for better performance during import.



```sql
-- Importing Products Data
BULK INSERT dbo.Products
FROM 'C:\Temp\Products.csv'
WITH (
    FORMAT = 'CSV',
    FIRSTROW = 2,
    FIELDTERMINATOR = ',',
    ROWTERMINATOR = '0x0A',
    TABLOCK
);

-- Importing Customers Data
BULK INSERT dbo.Customers
FROM 'C:\Temp\Customers.csv'
WITH (
    FORMAT = 'CSV',
    FIRSTROW = 2,
    FIELDTERMINATOR = ',',
    ROWTERMINATOR = '0x0A',
    TABLOCK
);
```



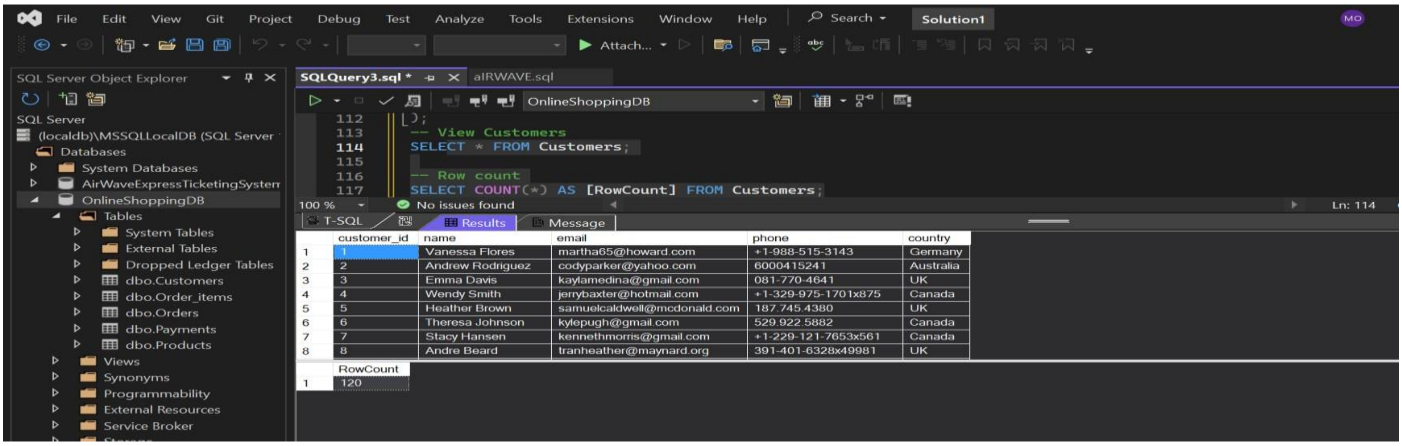
```sql
-- Importing Orders Data
BULK INSERT dbo.Orders
FROM 'C:\Temp\Orders.csv'
WITH (
    FORMAT = 'CSV',
    FIRSTROW = 2,
    FIELDTERMINATOR = ',',
    ROWTERMINATOR = '0x0A',
    TABLOCK
);

-- Importing Order_items Data
BULK INSERT dbo.Order_items
FROM 'C:\Temp\Order_items.csv'
WITH (
    FORMAT = 'CSV',
    FIRSTROW = 2,
    FIELDTERMINATOR = ',',
    ROWTERMINATOR = '0x0A',
    TABLOCK
);
```



```sql
-- Importing Payments Data
BULK INSERT dbo.Payments
FROM 'C:\Temp\Payments.csv'
WITH (
    FORMAT = 'CSV',
    FIRSTROW = 2,
    FIELDTERMINATOR = ',',
    ROWTERMINATOR = '0x0A',
    TABLOCK
);
```

1. Querying the Tables

Use **SELECT * FROM table_name**; to get all data from the table. To count the rows, use **SELECT COUNT(*) FROM table_name.** With the following statement we can view all the data in the customer table.



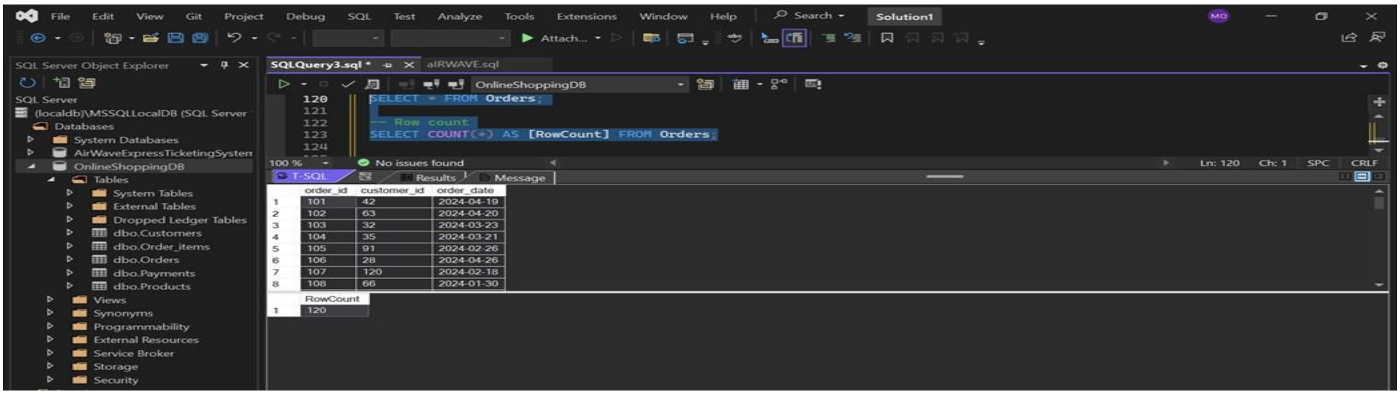
```sql
);
-- View Customers
SELECT * FROM Customers;

-- Row count
SELECT COUNT(*) AS [RowCount] FROM Customers;
```

	customer_id	name	email	phone	country
1	1	Vanessa Flores	martha65@howard.com	+1-988-515-3143	Germany
2	2	Andrew Rodriguez	codyparker@yahoo.com	6000415241	Australia
3	3	Emma Davis	kaylamedina@gmail.com	081-770-4641	UK
4	4	Wendy Smith	jerrybaxter@hotmail.com	+1-329-975-1701x875	Canada
5	5	Heather Brown	samuelcaldwell@mcdonald.com	187.745.4380	UK
6	6	Theresa Johnson	kylepugh@gmail.com	529.922.5882	Canada
7	7	Stacy Hansen	kennethmorris@gmail.com	+1-229-121-7653x561	Canada
8	8	Andre Beard	tranheather@maynard.org	391-401-6328x49981	UK

	RowCount
1	120

Customer table:

A total of 120 rows



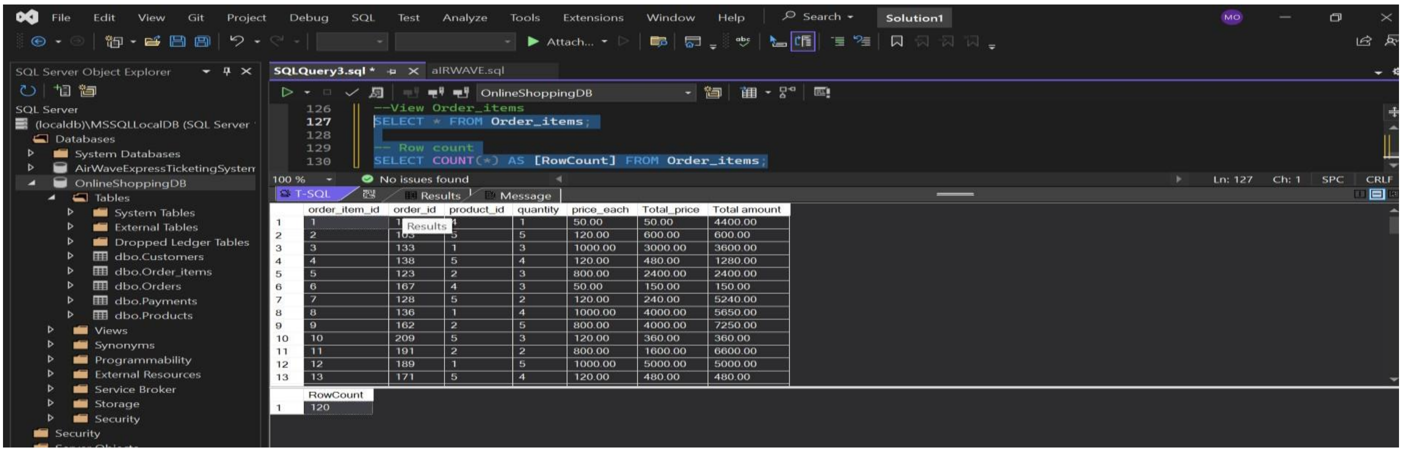
```sql
SELECT * FROM Orders;

-- Row count
SELECT COUNT(*) AS [RowCount] FROM Orders;
```

	order_id	customer_id	order_date
1	101	42	2024-04-19
2	102	63	2024-04-20
3	103	32	2024-03-23
4	104	35	2024-03-21
5	105	91	2024-02-26
6	106	28	2024-04-26
7	107	120	2024-02-18
8	108	66	2024-01-30

	RowCount
1	120

Order table:

A total of 120 rows.



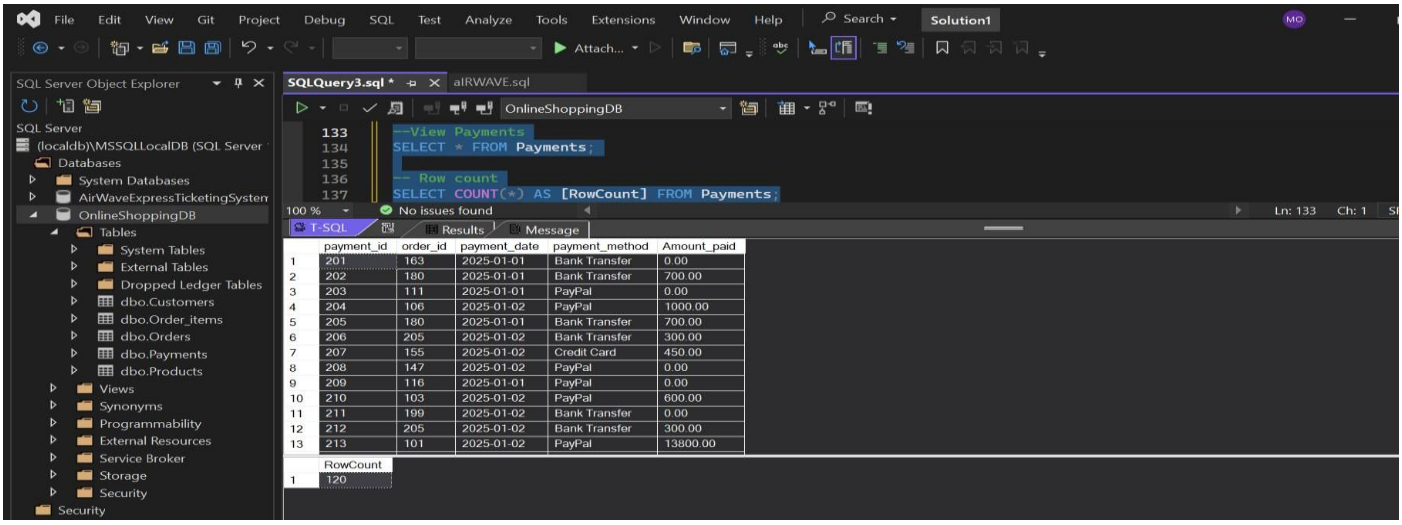
```sql
--View Order_items
SELECT * FROM Order_items;

-- Row count
SELECT COUNT(*) AS [RowCount] FROM Order_items;
```

	order_item_id	order_id	product_id	quantity	price_each	Total_price	Total amount
1	1	[illegible]	4	1	50.00	50.00	4400.00
2	2	103	5	5	120.00	600.00	600.00
3	3	133	1	3	1000.00	3000.00	3600.00
4	4	138	5	4	120.00	480.00	1280.00
5	5	123	2	3	800.00	2400.00	2400.00
6	6	167	4	3	50.00	150.00	150.00
7	7	128	5	2	120.00	240.00	5240.00
8	8	136	1	4	1000.00	4000.00	5650.00
9	9	162	2	5	800.00	4000.00	7250.00
10	10	209	5	3	120.00	360.00	360.00
11	11	191	2	2	800.00	1600.00	6600.00
12	12	189	1	5	1000.00	5000.00	5000.00
13	13	171	5	4	120.00	480.00	480.00

	RowCount
1	120

Order_items table:

A total of 120 rows.



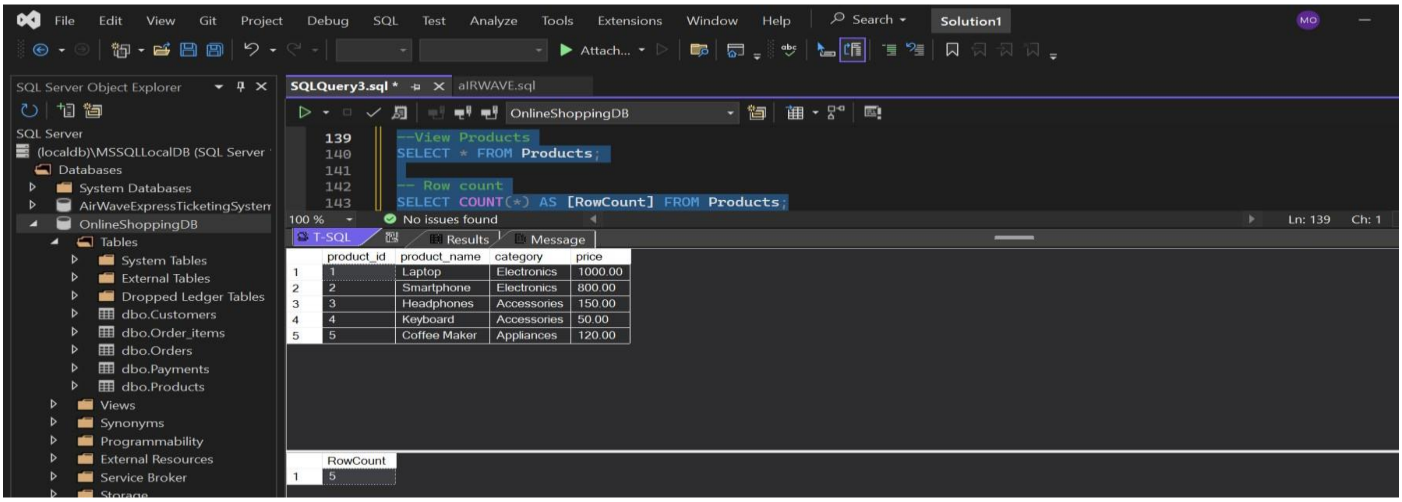
```sql
--View Payments
SELECT * FROM Payments;

-- Row count
SELECT COUNT(*) AS [RowCount] FROM Payments;
```

	payment_id	order_id	payment_date	payment_method	Amount_paid
1	201	163	2025-01-01	Bank Transfer	0.00
2	202	180	2025-01-01	Bank Transfer	700.00
3	203	111	2025-01-01	PayPal	0.00
4	204	106	2025-01-02	PayPal	1000.00
5	205	180	2025-01-01	Bank Transfer	700.00
6	206	205	2025-01-02	Bank Transfer	300.00
7	207	155	2025-01-02	Credit Card	450.00
8	208	147	2025-01-02	PayPal	0.00
9	209	116	2025-01-01	PayPal	0.00
10	210	103	2025-01-02	PayPal	600.00
11	211	199	2025-01-02	Bank Transfer	0.00
12	212	205	2025-01-02	Bank Transfer	300.00
13	213	101	2025-01-02	PayPal	13800.00

	RowCount
1	120

Payment table:

A total of 120 rows.



```sql
--View Products
SELECT * FROM Products;

-- Row count
SELECT COUNT(*) AS [RowCount] FROM Products;
```

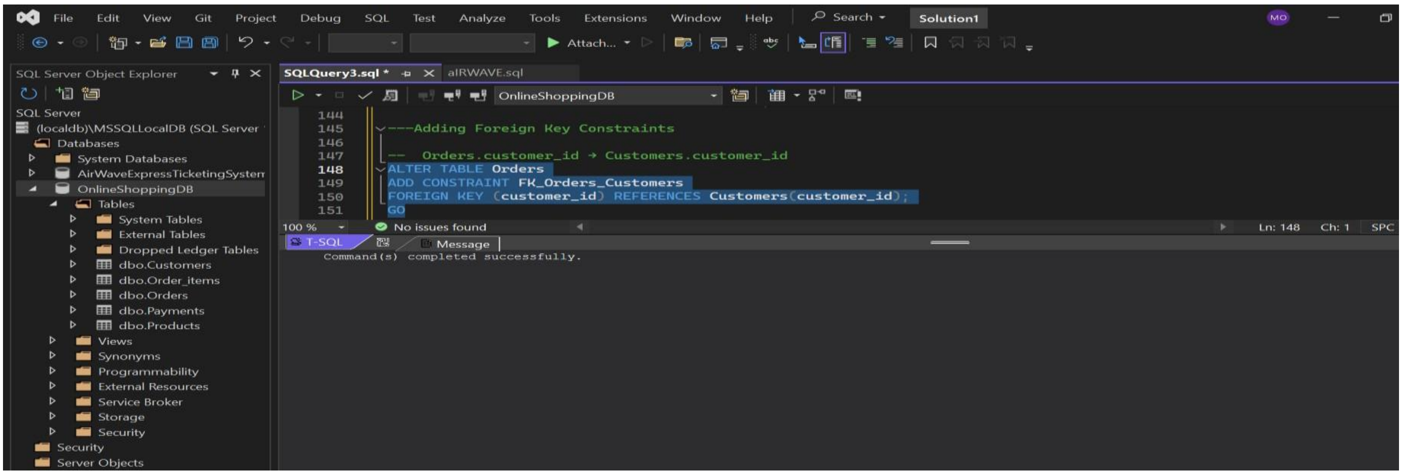
	product_id	product_name	category	price
1	1	Laptop	Electronics	1000.00
2	2	Smartphone	Electronics	800.00
3	3	Headphones	Accessories	150.00
4	4	Keyboard	Accessories	50.00
5	5	Coffee Maker	Appliances	120.00

	RowCount
1	5

Product table:

A total of 5 rows.

3. Relationships

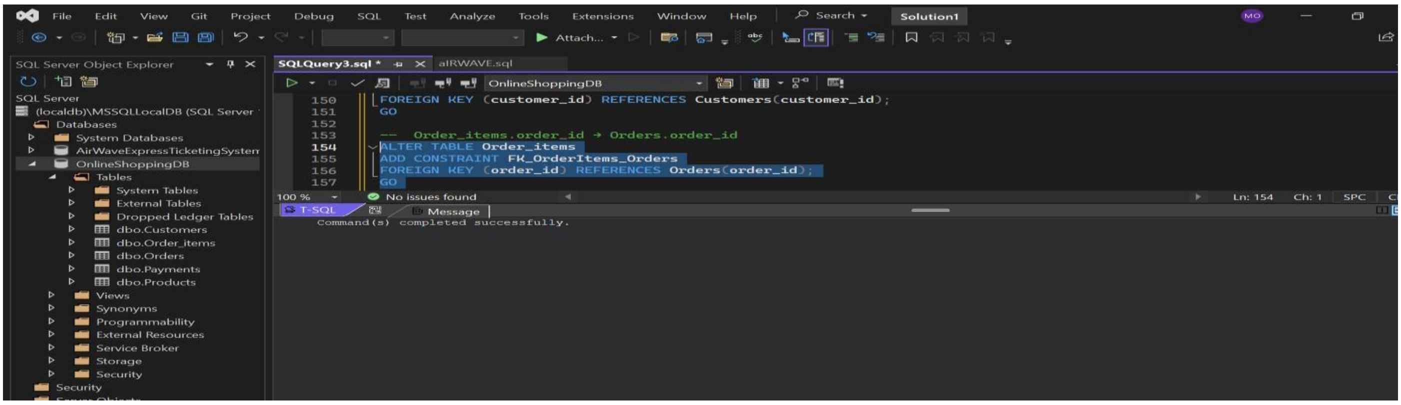


```
---Adding Foreign Key Constraints

-- Orders.customer_id → Customers.customer_id
ALTER TABLE Orders
ADD CONSTRAINT FK_Orders_Customers
FOREIGN KEY (customer_id) REFERENCES Customers(customer_id);
GO
```

```
Command(s) completed successfully.
```

- **FK_Orders_Customers**

This constraint makes sure that each order in the Orders table is connected to a valid customer in the Customers table. It creates a one-to-many relationship, meaning one customer can place many orders.

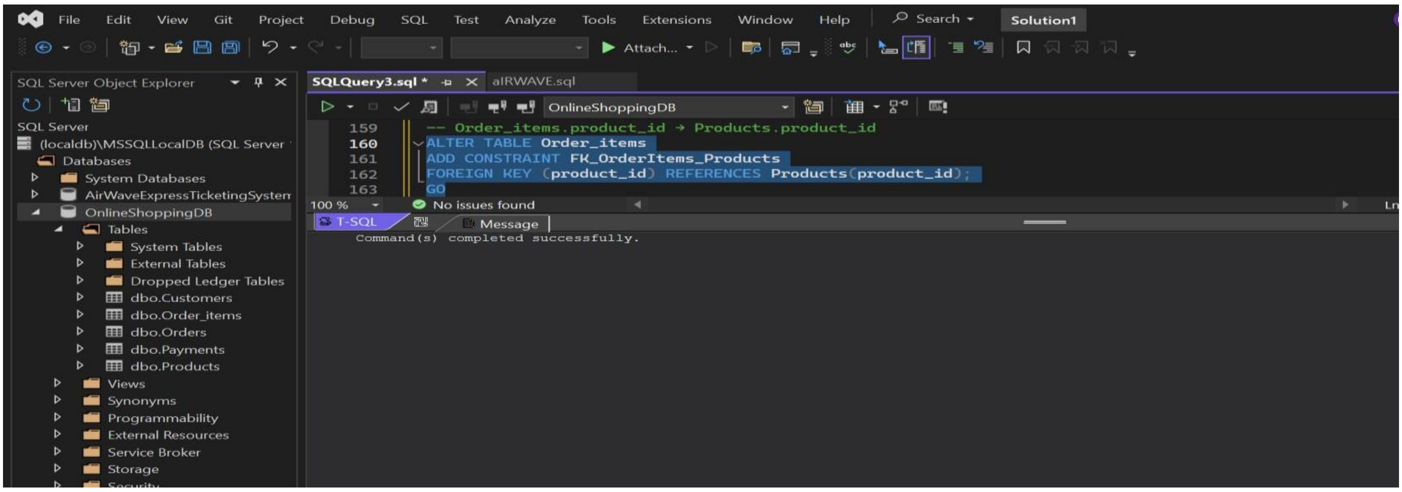


```
FOREIGN KEY (customer_id) REFERENCES Customers(customer_id);
GO

-- Order_items.order_id → Orders.order_id
ALTER TABLE Order_items
ADD CONSTRAINT FK_OrderItems_Orders
FOREIGN KEY (order_id) REFERENCES Orders(order_id);
GO
```

```
Command(s) completed successfully.
```

- **FK_OrderItems_Orders**

This ensures that every item in the Order_items table is linked to an actual order, helping to keep the data consistent and preventing items from existing without a related order.

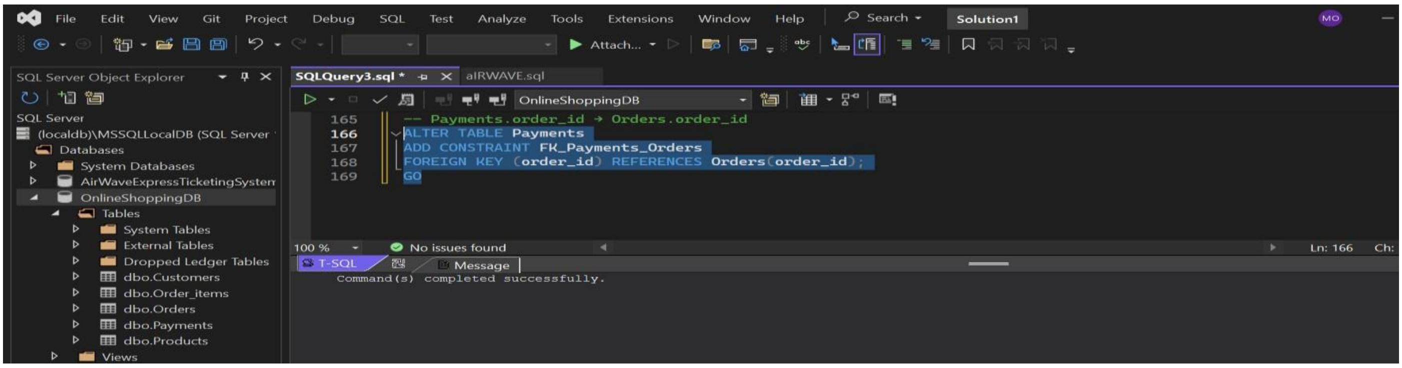


```
-- Order_items.product_id → Products.product_id
ALTER TABLE Order_items
ADD CONSTRAINT FK_OrderItems_Products
FOREIGN KEY (product_id) REFERENCES Products(product_id);
GO
```

```
Command(s) completed successfully.
```

- **FK_OrderItems_Products**

This constraint makes sure that every product in an order must already be listed in the Products table. It helps maintain accuracy by preventing the inclusion of products that don't exist.



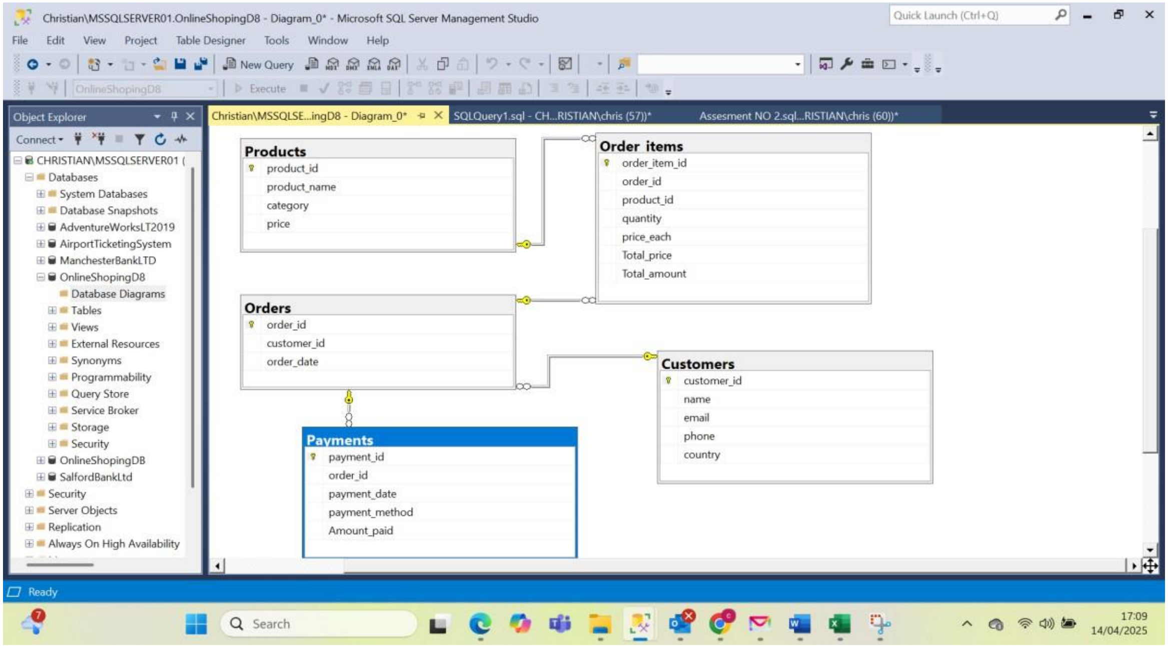
```sql
-- Payments.order_id → Orders.order_id
ALTER TABLE Payments
ADD CONSTRAINT FK_Payments_Orders
FOREIGN KEY (order_id) REFERENCES Orders(order_id);
GO
```

```
Command(s) completed successfully.
```

- **FK_Payments_Orders**

This ensures that every payment is linked to a valid order, helping with financial tracking and preventing payments from being recorded without a corresponding purchase.

Database Diagram



Products: product_id, product_name, category, price

Order_items: order_item_id, order_id, product_id, quantity, price_each, Total_price, Total_amount

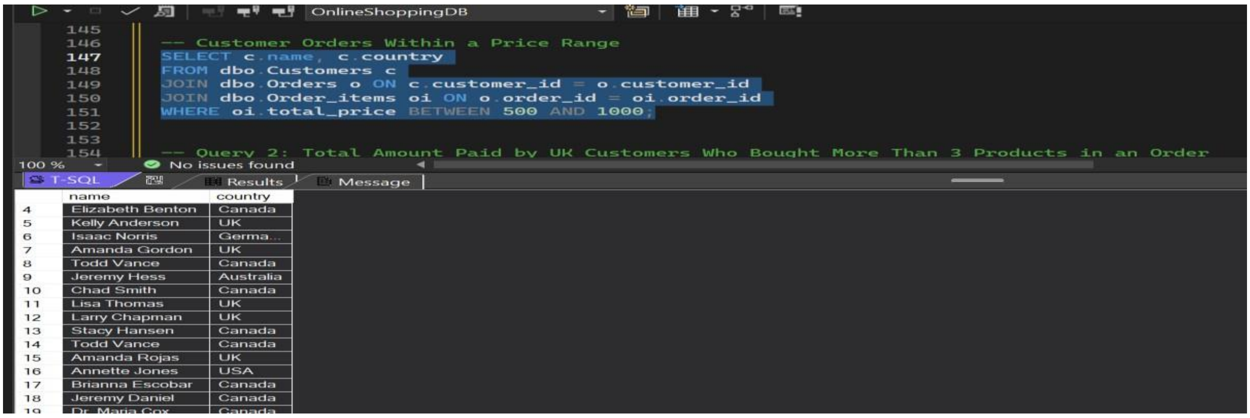
Orders: order_id, customer_id, order_date

Customers: customer_id, name, email, phone, country

Payments: payment_id, order_id, payment_date, payment_method, Amount_paid

4. Answers to Questions 2 – 4

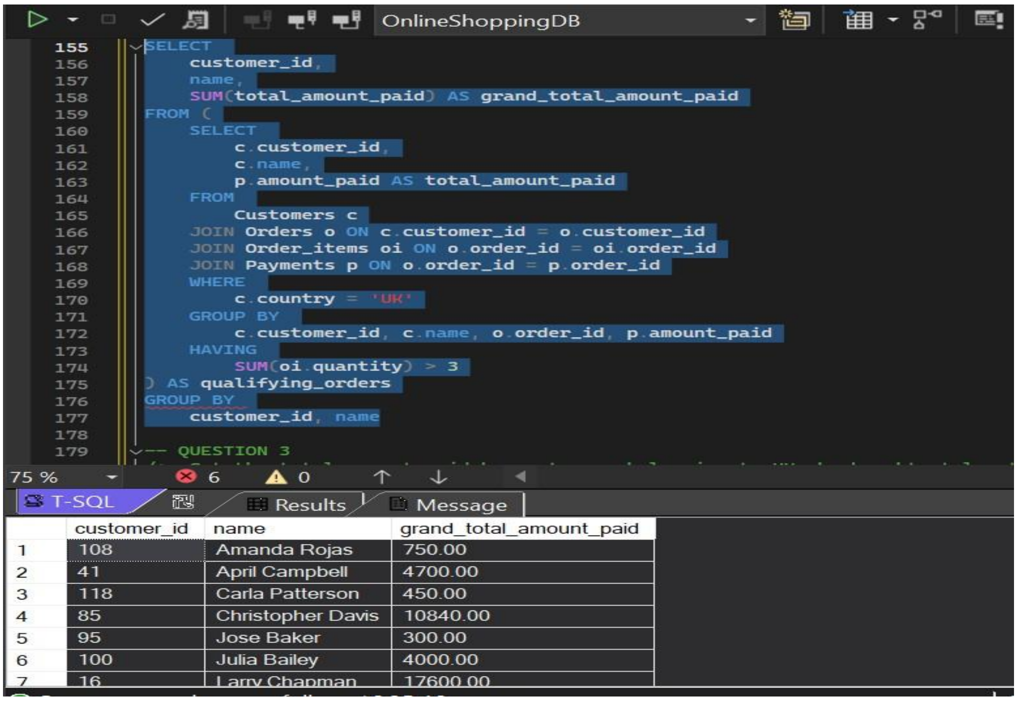
- Write a query that returns the names and countries of customers who made orders with a total amount between £500 and £1000.



```sql
-- Customer Orders Within a Price Range
SELECT c.name, c.country
FROM dbo.Customers c
JOIN dbo.Orders o ON c.customer_id = o.customer_id
JOIN dbo.Order_items oi ON o.order_id = oi.order_id
WHERE oi.total_price BETWEEN 500 AND 1000;

-- Query 2: Total Amount Paid by UK Customers Who Bought More Than 3 Products in an Order
```

	name	country
4	Elizabeth Benton	Canada
5	Kelly Anderson	UK
6	Isaac Norris	Germa...
7	Amanda Gordon	UK
8	Todd Vance	Canada
9	Jeremy Hess	Australia
10	Chad Smith	Canada
11	Lisa Thomas	UK
12	Larry Chapman	UK
13	Stacy Hansen	Canada
14	Todd Vance	Canada
15	Amanda Rojas	UK
16	Annette Jones	USA
17	Brianna Escobar	Canada
18	Jeremy Daniel	Canada
19	Dr. Maria Cox	Canada

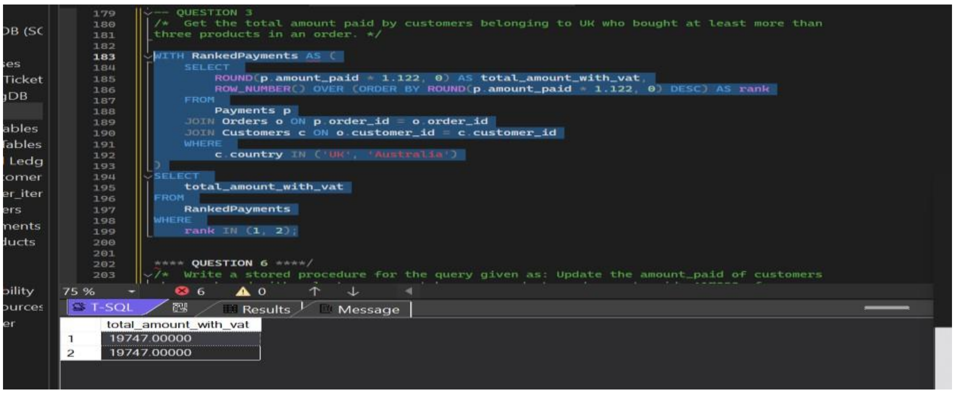
- Get the total amount paid by customers belonging to UK who bought at least more than three products in an order.



```sql
SELECT
    customer_id,
    name,
    SUM(total_amount_paid) AS grand_total_amount_paid
FROM (
    SELECT
        c.customer_id,
        c.name,
        p.amount_paid AS total_amount_paid
    FROM
        Customers c
    JOIN Orders o ON c.customer_id = o.customer_id
    JOIN Order_items oi ON o.order_id = oi.order_id
    JOIN Payments p ON o.order_id = p.order_id
    WHERE
        c.country = 'UK'
    GROUP BY
        c.customer_id, c.name, o.order_id, p.amount_paid
    HAVING
        SUM(oi.quantity) > 3
) AS qualifying_orders
GROUP BY
    customer_id, name

-- QUESTION 3
```

	customer_id	name	grand_total_amount_paid
1	108	Amanda Rojas	750.00
2	41	April Campbell	4700.00
3	118	Carla Patterson	450.00
4	85	Christopher Davis	10840.00
5	95	Jose Baker	300.00
6	100	Julia Bailey	4000.00
7	16	Larry Chapman	17600.00

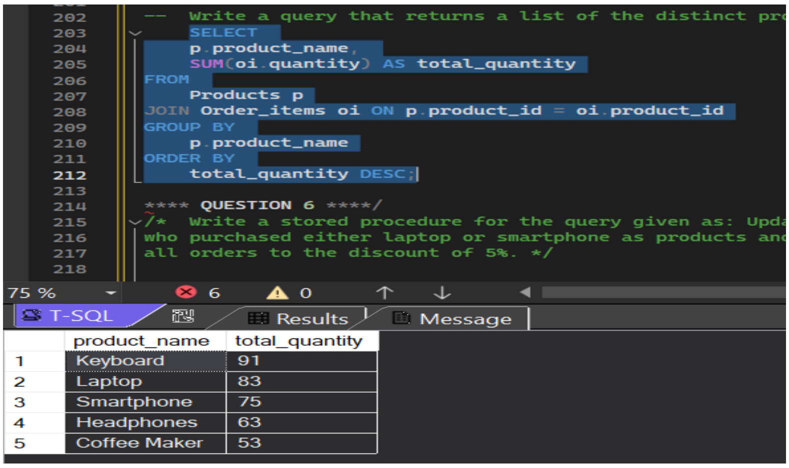
- Write a query to find the top two highest payments from UK or Australia after applying a 12.2% VAT, rounding the results to the nearest whole number.



```sql
-- QUESTION 3
/*  Get the total amount paid by customers belonging to UK who bought at least more than
three products in an order. */

WITH RankedPayments AS (
    SELECT
        ROUND(p.amount_paid * 1.122, 0) AS total_amount_with_vat,
        ROW_NUMBER() OVER (ORDER BY ROUND(p.amount_paid * 1.122, 0) DESC) AS rank
    FROM
        Payments p
    JOIN Orders o ON p.order_id = o.order_id
    JOIN Customers c ON o.customer_id = c.customer_id
    WHERE
        c.country IN ('UK', 'Australia')
)
SELECT
    total_amount_with_vat
FROM
    RankedPayments
WHERE
    rank IN (1, 2);


**** QUESTION 6 ****/
/*  Write a stored procedure for the query given as: Update the amount_paid of customers
```

	total_amount_with_vat
1	19747.00000
2	19747.00000

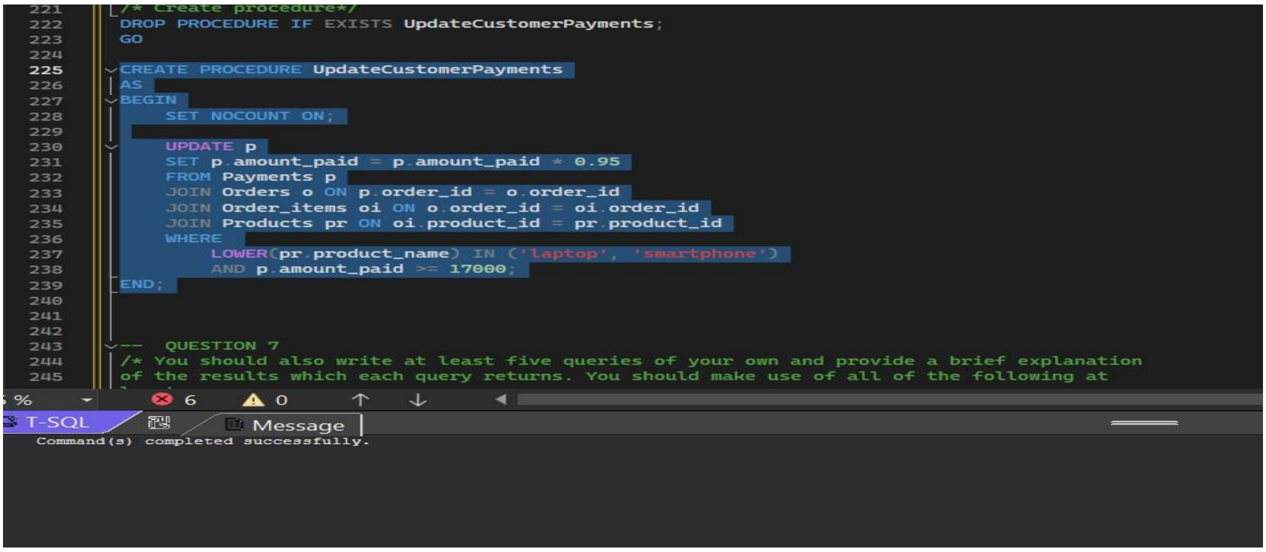
- Write a query that returns a list of the distinct product_name and the total quantity purchased for each product called as total_quantity. Sort by total_quantity.



```sql
-- Write a query that returns a list of the distinct pr
SELECT
    p.product_name,
    SUM(oi.quantity) AS total_quantity
FROM
    Products p
JOIN Order_items oi ON p.product_id = oi.product_id
GROUP BY
    p.product_name
ORDER BY
    total_quantity DESC;

**** QUESTION 6 ****/
/*  Write a stored procedure for the query given as: Upda
who purchased either laptop or smartphone as products an
all orders to the discount of 5%. */
```

	product_name	total_quantity
1	Keyboard	91
2	Laptop	83
3	Smartphone	75
4	Headphones	63
5	Coffee Maker	53

- Write a stored procedure for the query given as: Update the amount_paid of customers who purchased either laptop or smartphone as products and amount_paid>=£17000 of all orders to the discount of 5%.

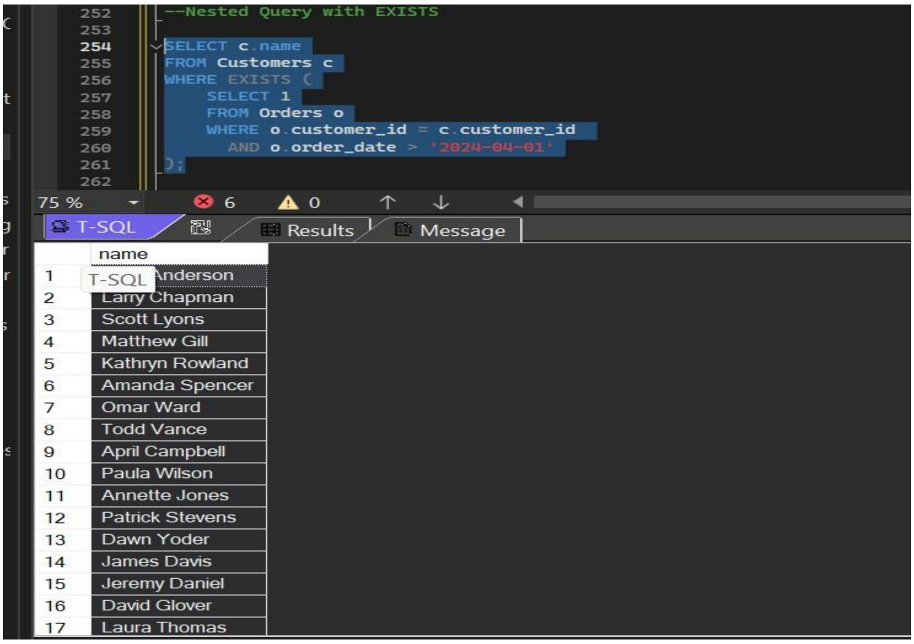


```
/* Create procedure*/
DROP PROCEDURE IF EXISTS UpdateCustomerPayments;
GO

CREATE PROCEDURE UpdateCustomerPayments
AS
BEGIN
    SET NOCOUNT ON;

    UPDATE p
    SET p.amount_paid = p.amount_paid * 0.95
    FROM Payments p
    JOIN Orders o ON p.order_id = o.order_id
    JOIN Order_items oi ON o.order_id = oi.order_id
    JOIN Products pr ON oi.product_id = pr.product_id
    WHERE
        LOWER(pr.product_name) IN ('laptop', 'smartphone')
        AND p.amount_paid >= 17000;
END;



-- QUESTION 7
/* You should also write at least five queries of your own and provide a brief explanation
of the results which each query returns. You should make use of all of the following at
```

```
Command(s) completed successfully.
```

- Write at least five queries of your own and provide a brief explanation of the results which each query returns. You should make use of all the following at least once:

i. Nested query including use of EXISTS or IN
ii. Joins
iii. System functions
iv. Use of GROUP BY, HAVING and ORDER BY clauses.

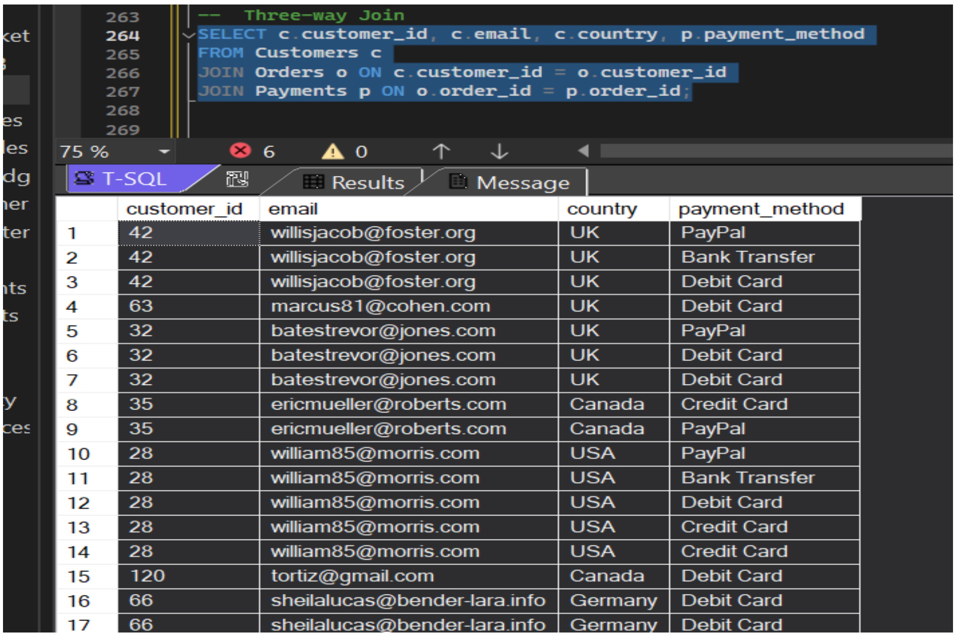


```
--Nested Query with EXISTS

SELECT c.name
FROM Customers c
WHERE EXISTS (
    SELECT 1
    FROM Orders o
    WHERE o.customer_id = c.customer_id
      AND o.order_date > '2024-04-01'
);
```

	name
1	T-SQL nderson
2	Larry Chapman
3	Scott Lyons
4	Matthew Gill
5	Kathryn Rowland
6	Amanda Spencer
7	Omar Ward
8	Todd Vance
9	April Campbell
10	Paula Wilson
11	Annette Jones
12	Patrick Stevens
13	Dawn Yoder
14	James Davis
15	Jeremy Daniel
16	David Glover
17	Laura Thomas

Nested query including use of EXISTS

Returns names of customers who placed at least one order after April 1, 2024.

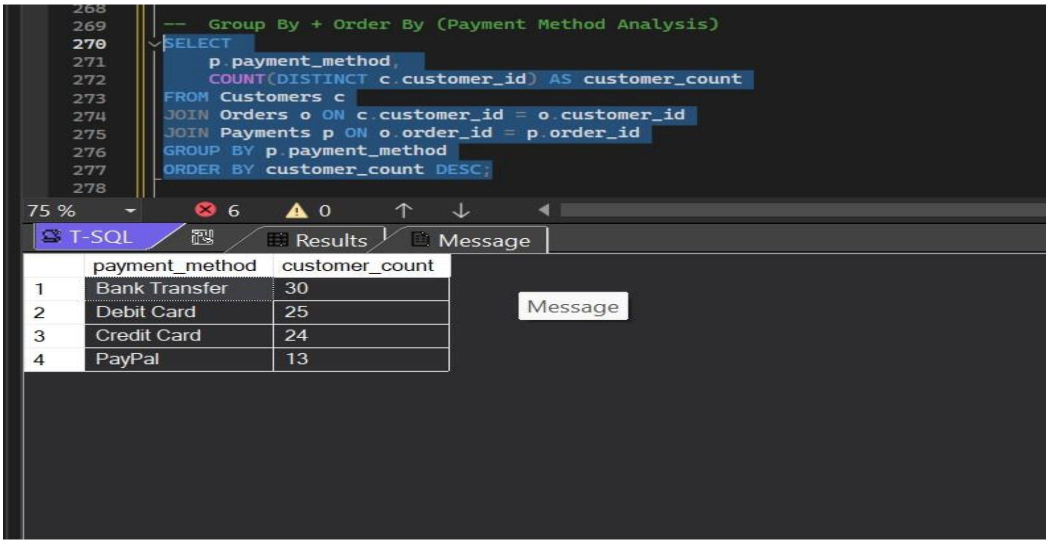


```sql
-- Three-way Join
SELECT c.customer_id, c.email, c.country, p.payment_method
FROM Customers c
JOIN Orders o ON c.customer_id = o.customer_id
JOIN Payments p ON o.order_id = p.order_id;
```

	customer_id	email	country	payment_method
1	42	willisjacob@foster.org	UK	PayPal
2	42	willisjacob@foster.org	UK	Bank Transfer
3	42	willisjacob@foster.org	UK	Debit Card
4	63	marcus81@cohen.com	UK	Debit Card
5	32	batestrevor@jones.com	UK	PayPal
6	32	batestrevor@jones.com	UK	Debit Card
7	32	batestrevor@jones.com	UK	Debit Card
8	35	ericmueller@roberts.com	Canada	Credit Card
9	35	ericmueller@roberts.com	Canada	PayPal
10	28	william85@morris.com	USA	PayPal
11	28	william85@morris.com	USA	Bank Transfer
12	28	william85@morris.com	USA	Debit Card
13	28	william85@morris.com	USA	Credit Card
14	28	william85@morris.com	USA	Credit Card
15	120	tortiz@gmail.com	Canada	Debit Card
16	66	sheilalucas@bender-lara.info	Germany	Debit Card
17	66	sheilalucas@bender-lara.info	Germany	Debit Card

3-way Join

Joins Customers, Orders, and Payments tables to display each customer's email, country, and how they made their payment.

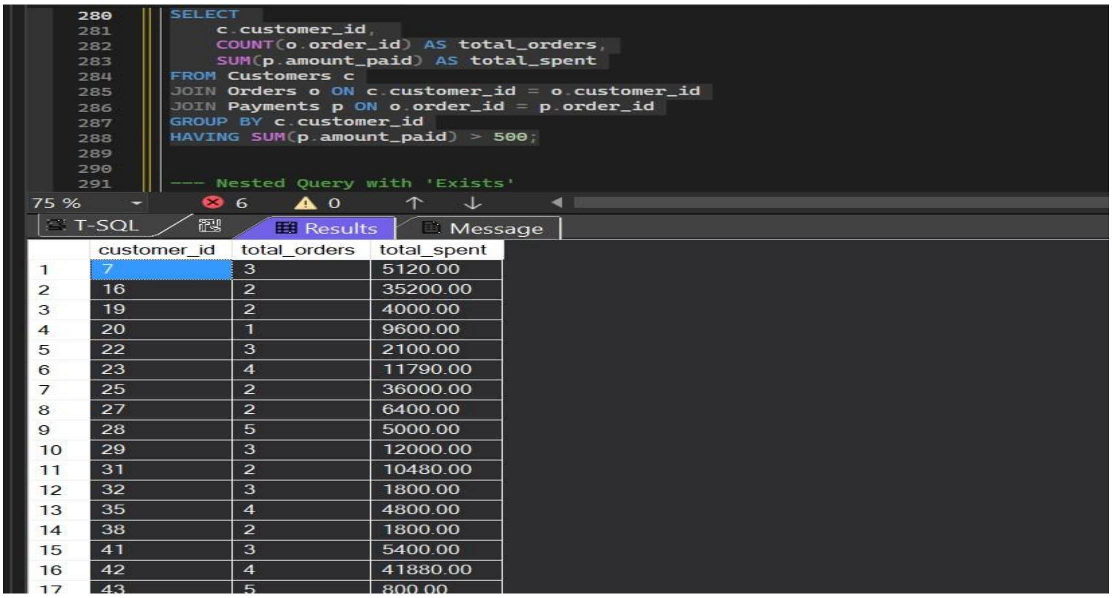


```sql
-- Group By + Order By (Payment Method Analysis)
SELECT
    p.payment_method,
    COUNT(DISTINCT c.customer_id) AS customer_count
FROM Customers c
JOIN Orders o ON c.customer_id = o.customer_id
JOIN Payments p ON o.order_id = p.order_id
GROUP BY p.payment_method
ORDER BY customer_count DESC;
```

	payment_method	customer_count
1	Bank Transfer	30
2	Debit Card	25
3	Credit Card	24
4	PayPal	13

Group By + Order By (Payment Method Analysis)

It shows how many customers used each payment method, helping identify the most popular options.

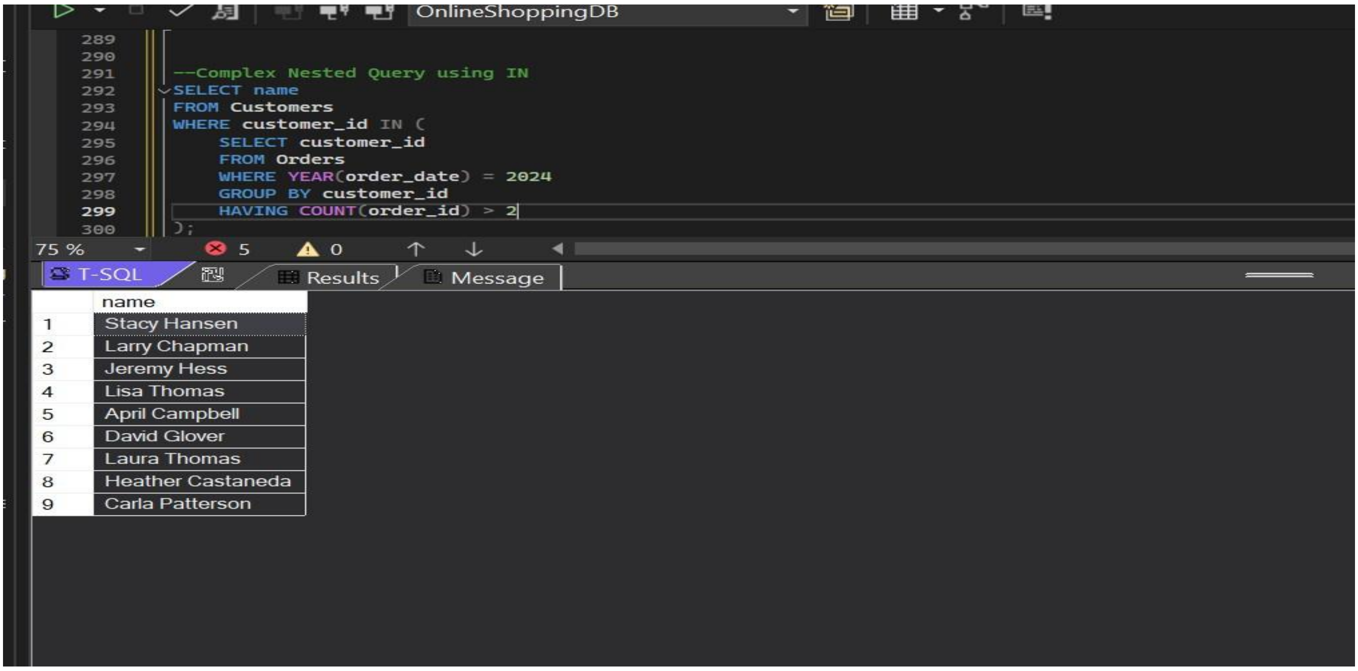


```sql
SELECT
    c.customer_id,
    COUNT(o.order_id) AS total_orders,
    SUM(p.amount_paid) AS total_spent
FROM Customers c
JOIN Orders o ON c.customer_id = o.customer_id
JOIN Payments p ON o.order_id = p.order_id
GROUP BY c.customer_id
HAVING SUM(p.amount_paid) > 500;


--- Nested Query with 'Exists'
```

	customer_id	total_orders	total_spent
1	7	3	5120.00
2	16	2	35200.00
3	19	2	4000.00
4	20	1	9600.00
5	22	3	2100.00
6	23	4	11790.00
7	25	2	36000.00
8	27	2	6400.00
9	28	5	5000.00
10	29	3	12000.00
11	31	2	10480.00
12	32	3	1800.00
13	35	4	4800.00
14	38	2	1800.00
15	41	3	5400.00
16	42	4	41880.00
17	43	5	800.00

System Functions – Total Spending

This query uses aggregate functions (COUNT and SUM) to calculate how many orders each customer made and their total spending. It then filters to show only customers who spent more than 500.

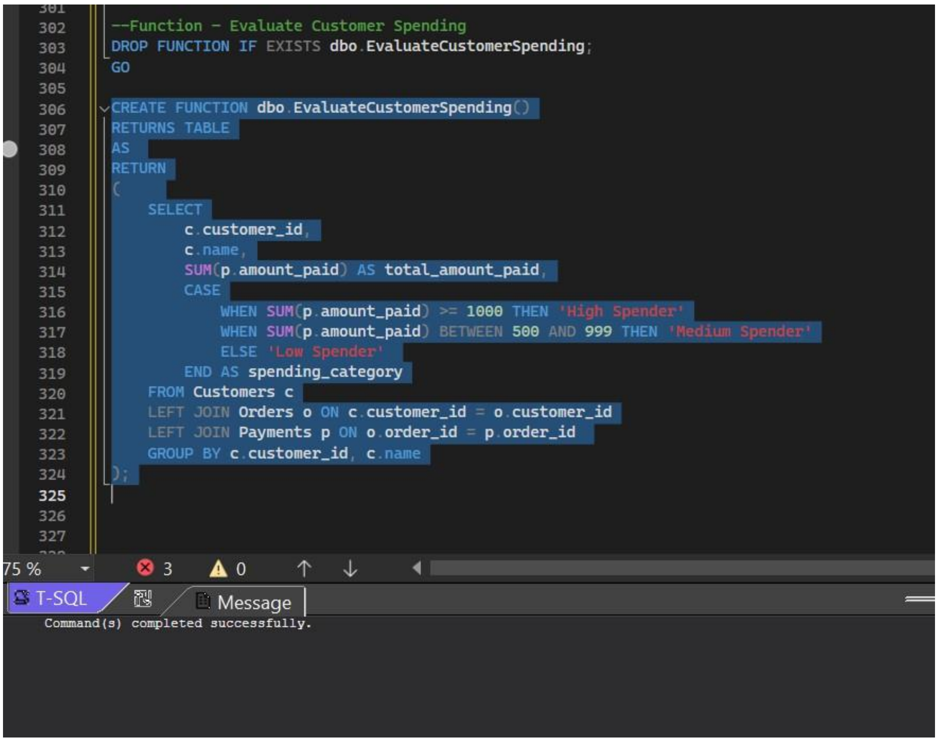


```sql

--Complex Nested Query using IN
SELECT name
FROM Customers
WHERE customer_id IN (
    SELECT customer_id
    FROM Orders
    WHERE YEAR(order_date) = 2024
    GROUP BY customer_id
    HAVING COUNT(order_id) > 2
);
```

	name
1	Stacy Hansen
2	Larry Chapman
3	Jeremy Hess
4	Lisa Thomas
5	April Campbell
6	David Glover
7	Laura Thomas
8	Heather Castaneda
9	Carla Patterson

Nested Query using IN

Finds customers who placed more than 3 orders in 2024.



```sql
--Function - Evaluate Customer Spending
DROP FUNCTION IF EXISTS dbo.EvaluateCustomerSpending;
GO

CREATE FUNCTION dbo.EvaluateCustomerSpending()
RETURNS TABLE
AS
RETURN
(
    SELECT
        c.customer_id,
        c.name,
        SUM(p.amount_paid) AS total_amount_paid,
        CASE
            WHEN SUM(p.amount_paid) >= 1000 THEN 'High Spender'
            WHEN SUM(p.amount_paid) BETWEEN 500 AND 999 THEN 'Medium Spender'
            ELSE 'Low Spender'
        END AS spending_category
    FROM Customers c
    LEFT JOIN Orders o ON c.customer_id = o.customer_id
    LEFT JOIN Payments p ON o.order_id = p.order_id
    GROUP BY c.customer_id, c.name
);
```

```
Command(s) completed successfully.
```

Function – Evaluate Customer Spending

• Conclusion

The OnlineShoppingDB is a solid solution for e-commerce platforms. It ensures data security, integrity, and performance while supporting important features like user management, order processing, and inventory tracking. With good design and best practices, it provides a reliable base for smooth operations and future growth.