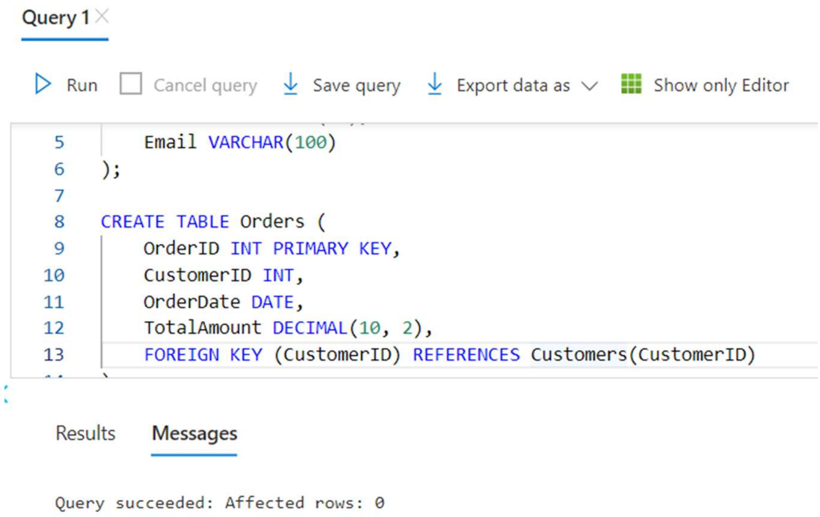
Context

Joins

1. In this example we are going to use the Customers and Orders table.
2. Creaboth tables by using the below queries.

```
Query: CREATE TABLE Customers (
  CustomerID INT PRIMARY KEY,
  FirstName VARCHAR(50),
  LastName VARCHAR(50),
  Email VARCHAR(100)
);

CREATE TABLE Orders (
  OrderID INT PRIMARY KEY,
  CustomerID INT,
  OrderDate DATE,
  TotalAmount DECIMAL(10, 2),
  FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```



```
    Email VARCHAR(100)
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    TotalAmount DECIMAL(10, 2),
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
```

```
Query succeeded: Affected rows: 0
```

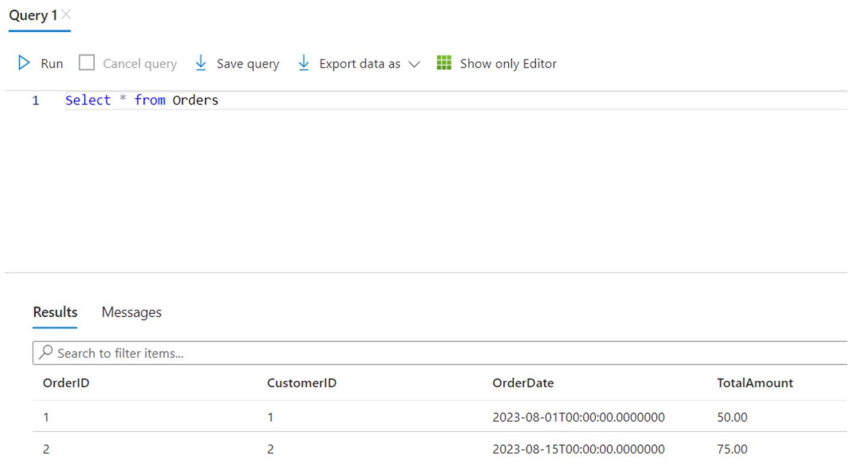
3. Insert some sample data into both tables by using the below query.

```
Query: -- Insert data into the Customers table
INSERT INTO Customers (CustomerID, FirstName, LastName, Email)
VALUES
  (1, 'John', 'Doe', 'john.doe@example.com'),
  (2, 'Jane', 'Smith', 'jane.smith@example.com')

-- Insert data into the Orders table
INSERT INTO Orders (OrderID, CustomerID, OrderDate, TotalAmount)
VALUES
  (1, 1, '2023-08-01', 50.00),
  (2, 2, '2023-08-15', 75.00)
```

4. To check the Use below Query.

Query: Select * from Orders



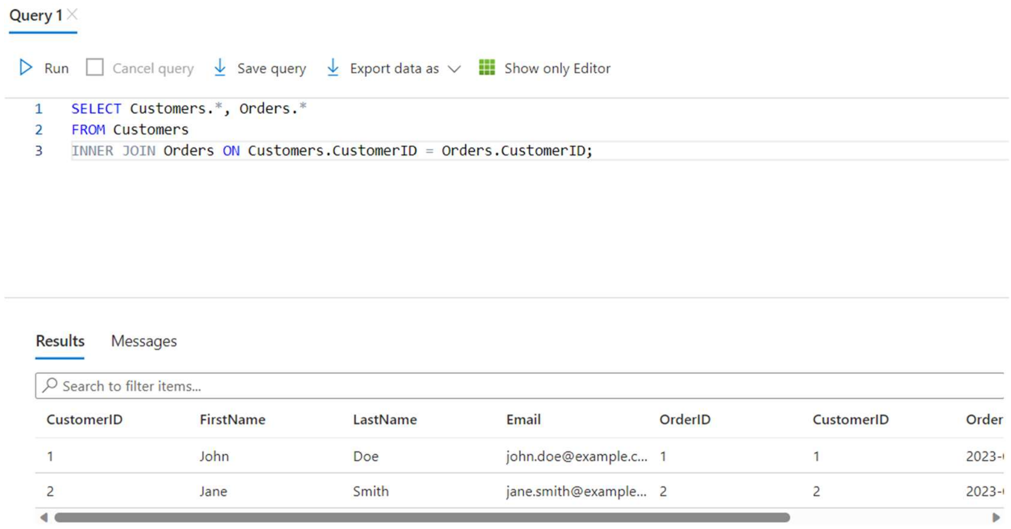
```
Select * from Orders
```

OrderID	CustomerID	OrderDate	TotalAmount
1	1	2023-08-01T00:00:00.0000000	50.00
2	2	2023-08-15T00:00:00.0000000	75.00

Inner Join

5. An inner join retrieves only the records that have matching values in both tables.

Query: SELECT Customers.*, Orders.*
FROM Customers
INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID;



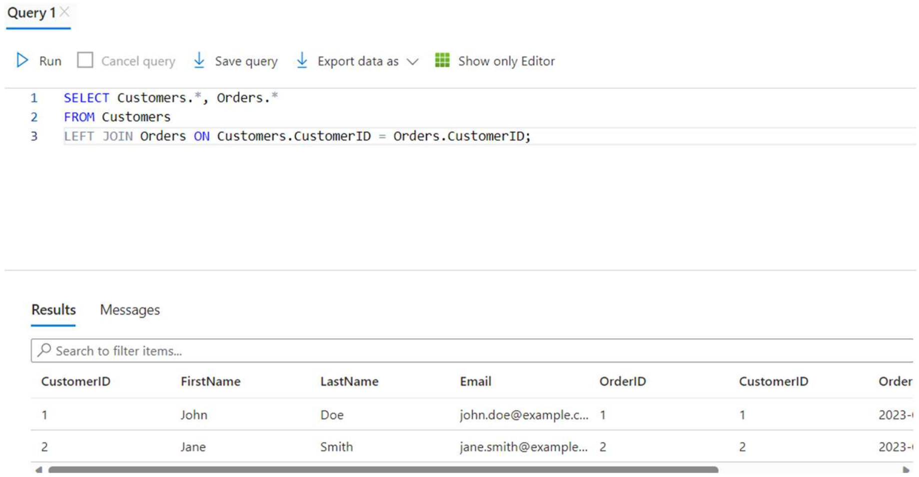
```
SELECT Customers.*, Orders.*
FROM Customers
INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

CustomerID	FirstName	LastName	Email	OrderID	CustomerID	Order
1	John	Doe	john.doe@example.c...	1	1	2023-
2	Jane	Smith	jane.smith@example...	2	2	2023-

Left Join

6. A left join retrieves all records from the left table (Customers) and matching records from the right table (Orders).

Query: SELECT Customers.*, Orders.*
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;



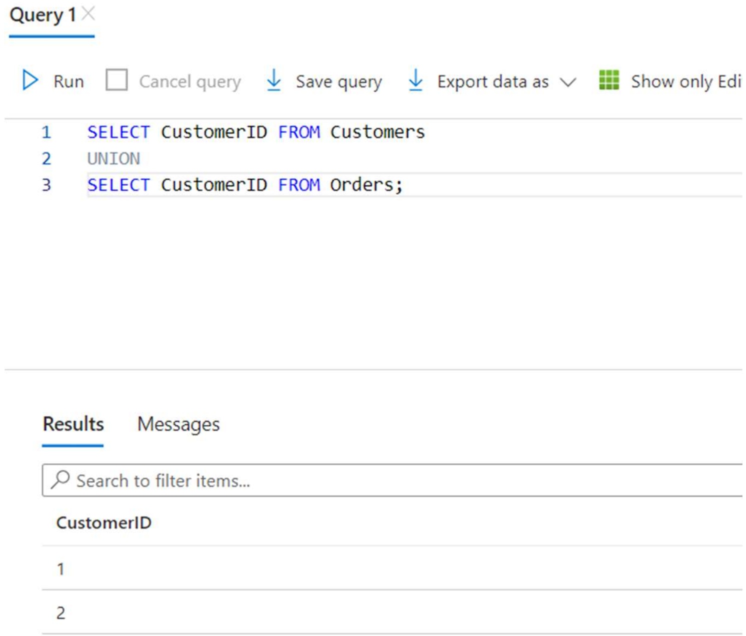
```
SELECT Customers.*, Orders.*
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

CustomerID	FirstName	LastName	Email	OrderID	CustomerID	Order
1	John	Doe	john.doe@example.c...	1	1	2023-
2	Jane	Smith	jane.smith@example...	2	2	2023-

UNION

1. The UNION operator merges the results of two or more SELECT queries, removing duplicate records.

Query: SELECT CustomerID FROM Customers
UNION
SELECT CustomerID FROM Orders;



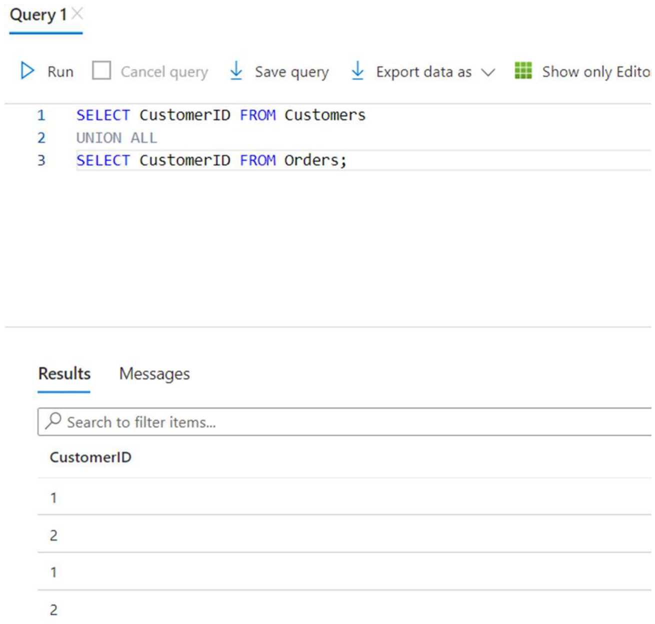
```
SELECT CustomerID FROM Customers
UNION
SELECT CustomerID FROM Orders;
```

CustomerID
1
2

UNION ALL

1. The UNION ALL operator merges the results of two or more SELECT queries, including duplicate records.

Query: SELECT CustomerID FROM Customers
UNION ALL
SELECT CustomerID FROM Orders;



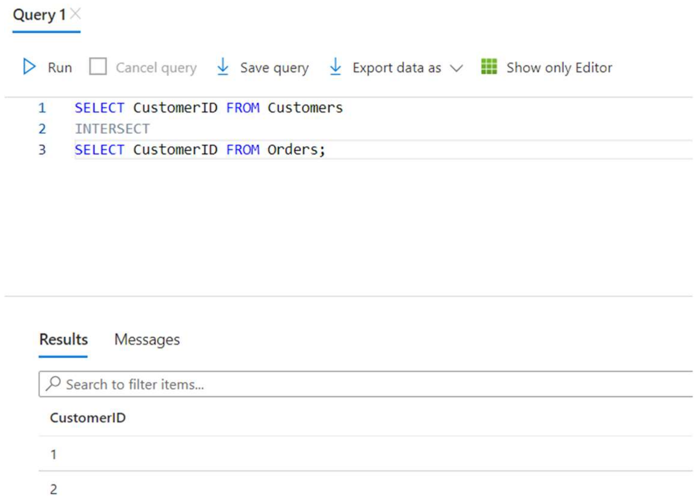
```sql
SELECT CustomerID FROM Customers
UNION ALL
SELECT CustomerID FROM Orders;
```

CustomerID
1
2
1
2

INTERSECT

1. The INTERSECT operator retrieves the common records between the results of two SELECT queries.

Query: SELECT CustomerID FROM Customers
INTERSECT
SELECT CustomerID FROM Orders;



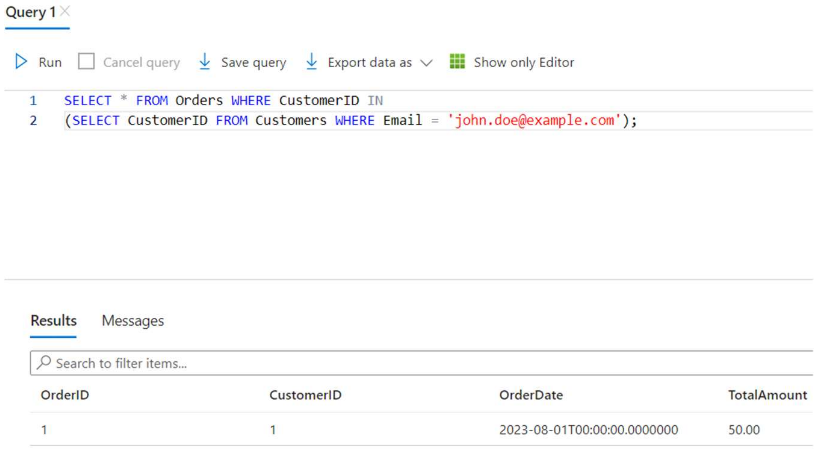
```sql
SELECT CustomerID FROM Customers
INTERSECT
SELECT CustomerID FROM Orders;
```

CustomerID
1
2

Subqueries

1. Subqueries are queries within queries. Let's use subqueries to retrieve specific subsets of data.
2. Retrieve orders for customers with a specific email.

Query: SELECT * FROM Orders WHERE CustomerID IN
(SELECT CustomerID FROM Customers WHERE Email = 'john.doe@example.com');

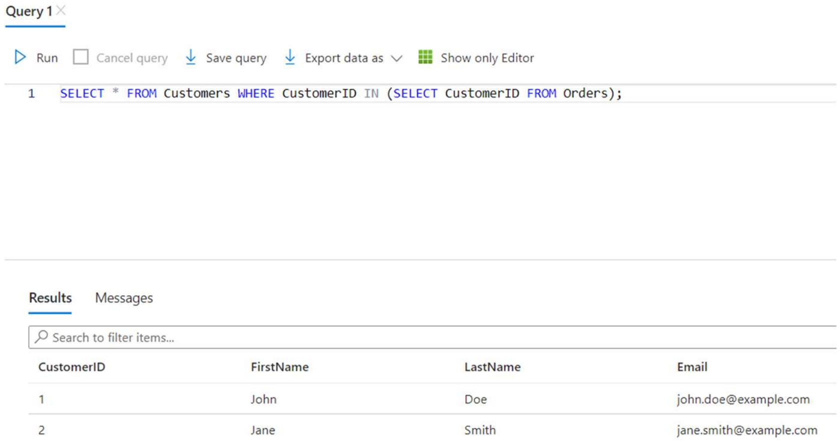


```
SELECT * FROM Orders WHERE CustomerID IN
(SELECT CustomerID FROM Customers WHERE Email = 'john.doe@example.com');
```

OrderID	CustomerID	OrderDate	TotalAmount
1	1	2023-08-01T00:00:00.0000000	50.00

3. Retrieve customers who placed orders.

Query: SELECT * FROM Customers WHERE CustomerID IN (SELECT CustomerID FROM Orders);



```
SELECT * FROM Customers WHERE CustomerID IN (SELECT CustomerID FROM Orders);
```

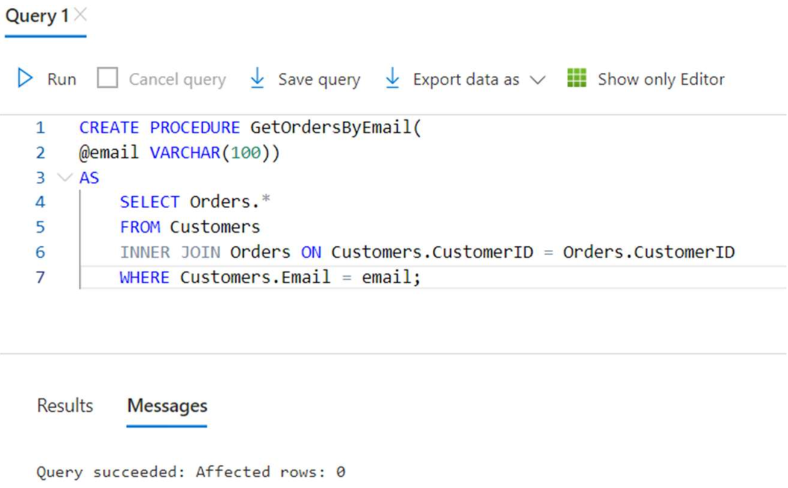
CustomerID	FirstName	LastName	Email
1	John	Doe	john.doe@example.com
2	Jane	Smith	jane.smith@example.com

Stored Procedure

1. A stored procedure is a set of SQL statements that can be executed as a single unit.
2. Let's create a stored procedure that retrieves orders for a specific customer based on their email.

Query: CREATE PROCEDURE GetOrdersByEmail(
@email VARCHAR(100))
AS
  SELECT Orders.*

```
  FROM Customers
  INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID
  WHERE Customers.Email = email;
```

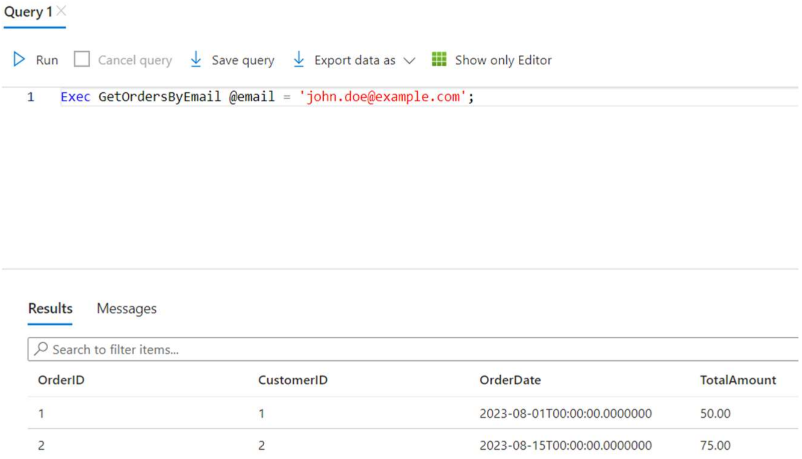


```
CREATE PROCEDURE GetOrdersByEmail(
@email VARCHAR(100))
AS
    SELECT Orders.*
    FROM Customers
    INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID
    WHERE Customers.Email = email;
```

```
Query succeeded: Affected rows: 0
```

3. Execute the stored procedure to get orders for a customer with a specific email

Query: Exec GetOrdersByEmail @email = 'john.doe@example.com';



```
Exec GetOrdersByEmail @email = 'john.doe@example.com';
```

OrderID	CustomerID	OrderDate	TotalAmount
1	1	2023-08-01T00:00:00.0000000	50.00
2	2	2023-08-15T00:00:00.0000000	75.00

Functions

1. A scalar-valued function is a function that returns a single value.

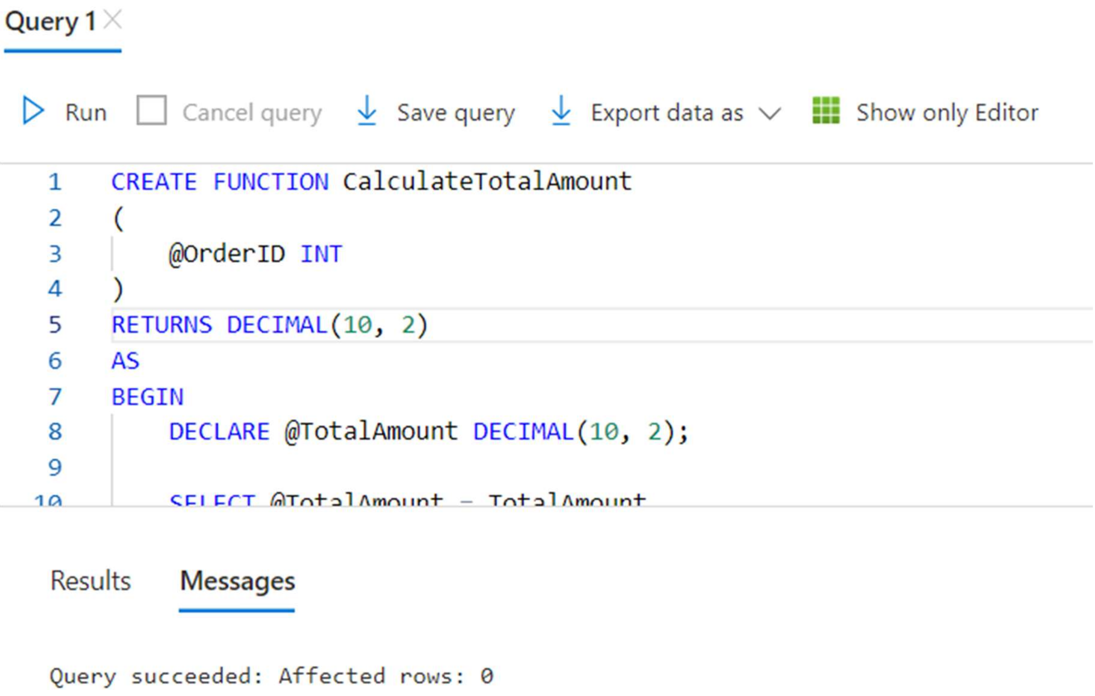
Query: CREATE FUNCTION CalculateTotalAmount

```
(
  @OrderID INT
)
RETURNS DECIMAL(10, 2)
AS
BEGIN
  DECLARE @TotalAmount DECIMAL(10, 2);

  SELECT @TotalAmount = TotalAmount
  FROM Orders
```

```
    WHERE OrderID = @OrderID;

    RETURN @TotalAmount;
END;
```



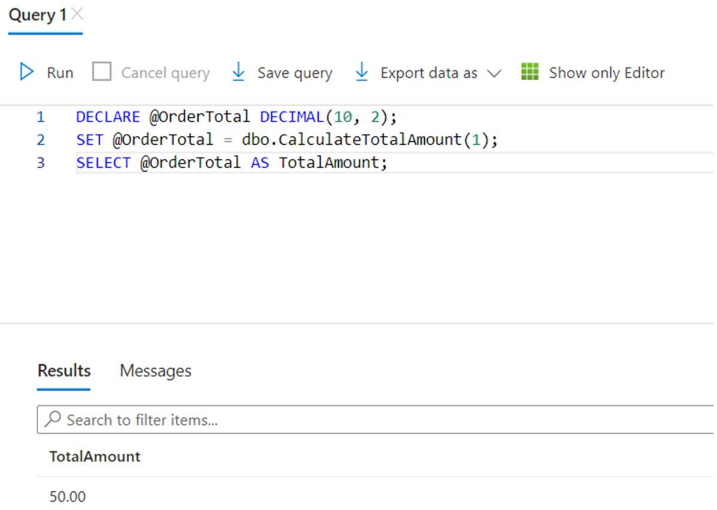
```
CREATE FUNCTION CalculateTotalAmount
(
    @OrderID INT
)
RETURNS DECIMAL(10, 2)
AS
BEGIN
    DECLARE @TotalAmount DECIMAL(10, 2);

    SELECT @TotalAmount = TotalAmount
```

```
Query succeeded: Affected rows: 0
```

2. Call the scalar-valued function.

Query: DECLARE @OrderTotal DECIMAL(10, 2);
SET @OrderTotal = dbo.CalculateTotalAmount(1);
SELECT @OrderTotal AS TotalAmount;



```
DECLARE @OrderTotal DECIMAL(10, 2);
SET @OrderTotal = dbo.CalculateTotalAmount(1);
SELECT @OrderTotal AS TotalAmount;
```

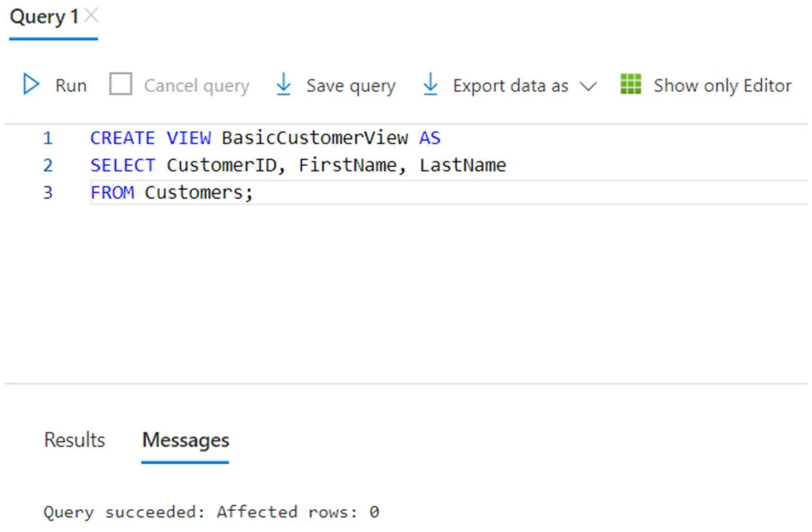
TotalAmount
50.00

Views

Creating a Simple View

1. Creating a basic view that selects columns from the Customers table.

Query: CREATE VIEW BasicCustomerView AS
SELECT CustomerID, FirstName, LastName
FROM Customers;



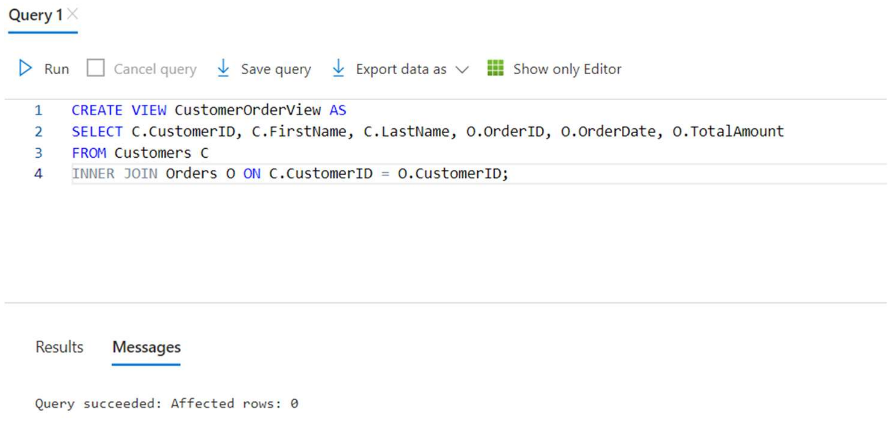
```
CREATE VIEW BasicCustomerView AS
SELECT CustomerID, FirstName, LastName
FROM Customers;
```

```
Query succeeded: Affected rows: 0
```

View with Join

2. Creating a view that combines customer information with their order details.

Query: CREATE VIEW CustomerOrderView AS
SELECT C.CustomerID, C.FirstName, C.LastName, O.OrderID, O.OrderDate, O.TotalAmount
FROM Customers C
INNER JOIN Orders O ON C.CustomerID = O.CustomerID;



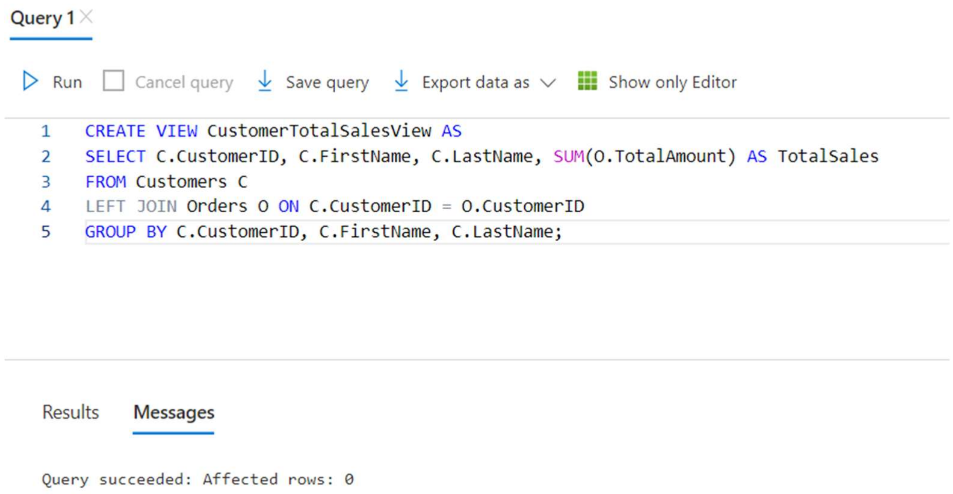
```
CREATE VIEW CustomerOrderView AS
SELECT C.CustomerID, C.FirstName, C.LastName, O.OrderID, O.OrderDate, O.TotalAmount
FROM Customers C
INNER JOIN Orders O ON C.CustomerID = O.CustomerID;
```

```
Query succeeded: Affected rows: 0
```

View with Aggregation

3. Creating a view that shows the total sales amount for each customer.

Query: CREATE VIEW CustomerTotalSalesView AS
SELECT C.CustomerID, C.FirstName, C.LastName, SUM(O.TotalAmount) AS TotalSales
FROM Customers C
LEFT JOIN Orders O ON C.CustomerID = O.CustomerID
GROUP BY C.CustomerID, C.FirstName, C.LastName;



```
CREATE VIEW CustomerTotalSalesView AS
SELECT C.CustomerID, C.FirstName, C.LastName, SUM(O.TotalAmount) AS TotalSales
FROM Customers C
LEFT JOIN Orders O ON C.CustomerID = O.CustomerID
GROUP BY C.CustomerID, C.FirstName, C.LastName;
```

```
Query succeeded: Affected rows: 0
```

Indexes

Create a "Products" table to explore the impact of indexes on query performance. You will start by creating the table without any specific index (heap). Then, you will add a clustered index, a non-clustered index, and a columnstore index to the table. Through a series of queries, you will compare the query execution times for different search conditions and analyze the performance improvements or differences brought by each index type.

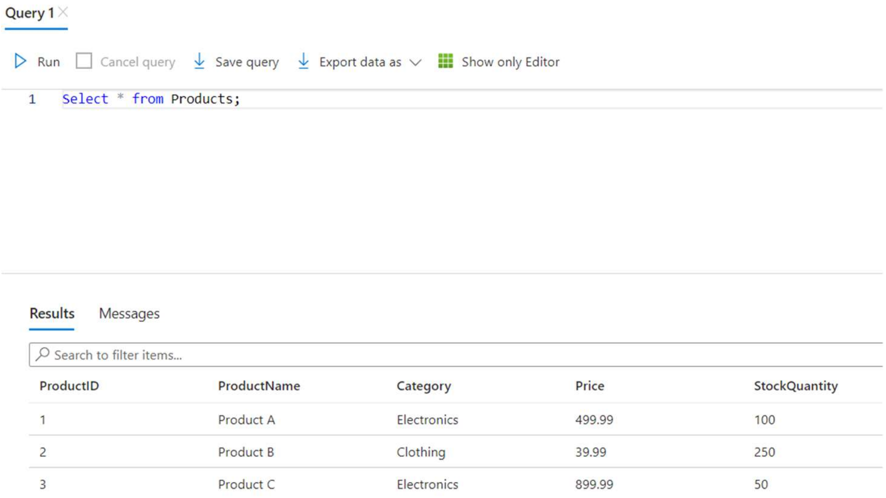
1. Use the below to Create a Product table and Insert some sample data.

```
Query: CREATE TABLE Products (
   ProductID INT,
   ProductName VARCHAR(100),
   Category VARCHAR(50),
   Price DECIMAL(10, 2),
   StockQuantity INT
);

INSERT INTO Products (ProductID, ProductName, Category, Price, StockQuantity)
VALUES
   (1, 'Product A', 'Electronics', 499.99, 100),
   (2, 'Product B', 'Clothing', 39.99, 250),
   (3, 'Product C', 'Electronics', 899.99, 50)
```

2. To check the Data use the below query.

Query: Select * from Products;



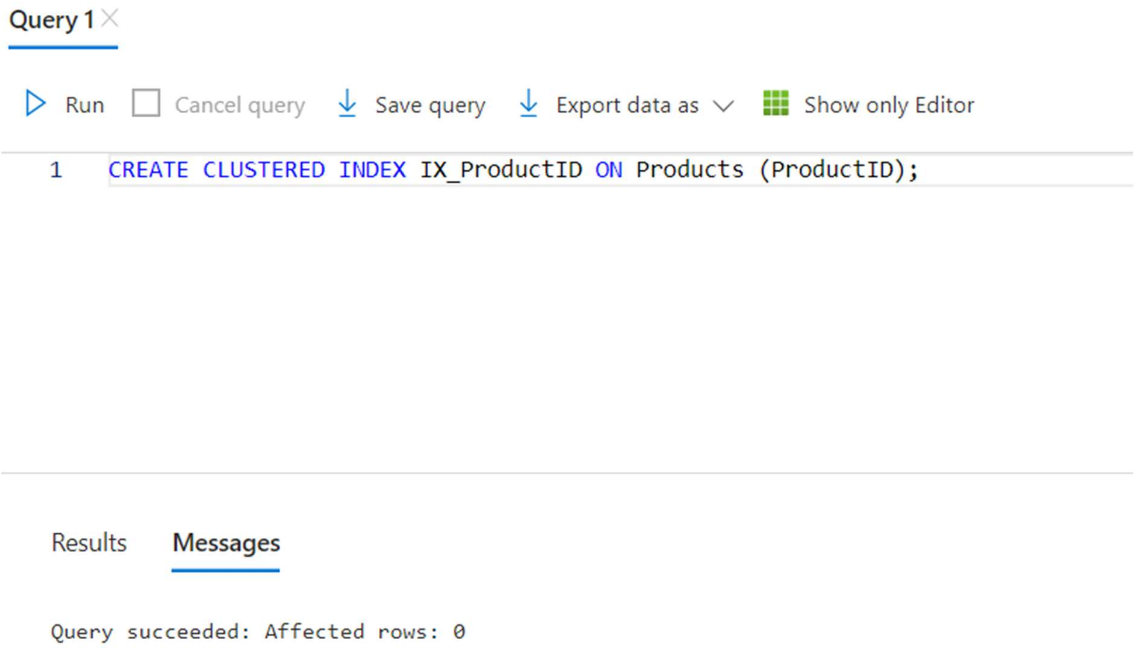
```
Select * from Products;
```

ProductID	ProductName	Category	Price	StockQuantity
1	Product A	Electronics	499.99	100
2	Product B	Clothing	39.99	250
3	Product C	Electronics	899.99	50

Clustered Index

3. A clustered index determines the physical order of rows in the table. Let's create a clustered index on the ProductID column:

Query: CREATE CLUSTERED INDEX IX_ProductID ON Products (ProductID);



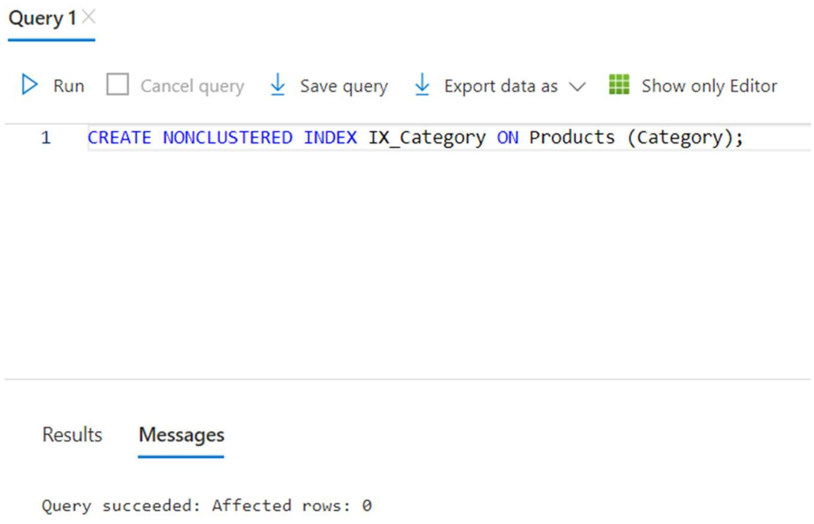
```
CREATE CLUSTERED INDEX IX_ProductID ON Products (ProductID);
```

```
Query succeeded: Affected rows: 0
```

Non-Clustered Index

4. A non-clustered index creates a separate structure for index data. Let's create a non-clustered index on the Category column:

Query: CREATE NONCLUSTERED INDEX IX_Category ON Products (Category);



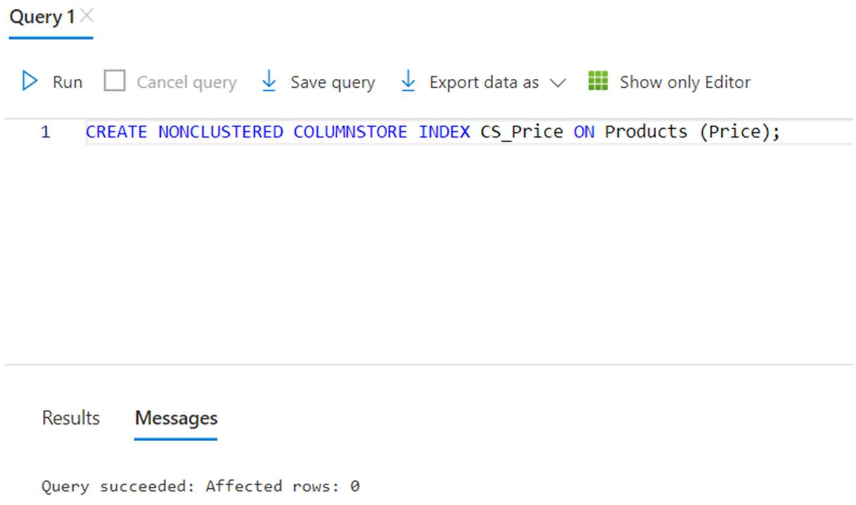
```
CREATE NONCLUSTERED INDEX IX_Category ON Products (Category);
```

```
Query succeeded: Affected rows: 0
```

Columnstore Index

5. A columnstore index stores data in a columnar format optimized for analytical queries. Let's create a columnstore index on the Price column.

Query: CREATE NONCLUSTERED COLUMNSTORE INDEX CS_Price ON Products (Price);



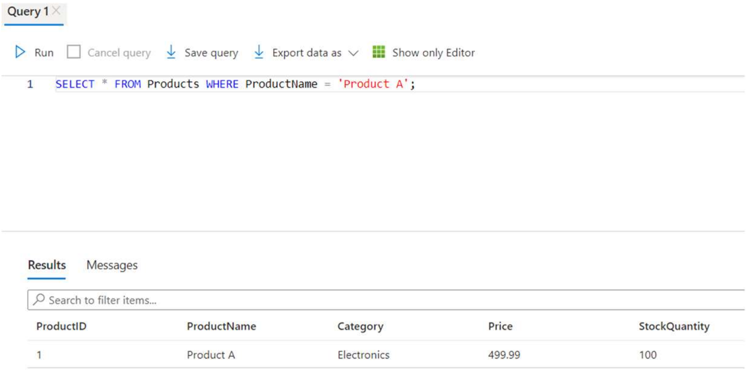
```
CREATE NONCLUSTERED COLUMNSTORE INDEX CS_Price ON Products (Price);
```

```
Query succeeded: Affected rows: 0
```

Compare Query Performance

6. Now, let's run some queries and compare their execution times for different index types:
7. Query using no index (Heap)

Query: SELECT * FROM Products WHERE ProductName = 'Product A';

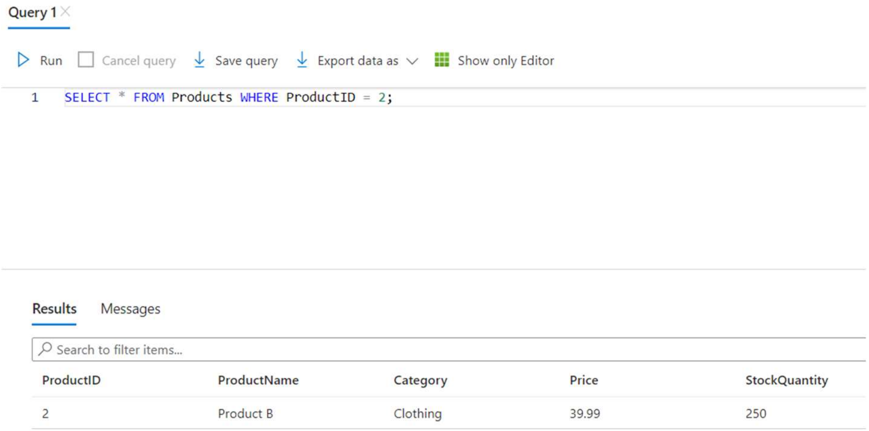


```
SELECT * FROM Products WHERE ProductName = 'Product A';
```

ProductID	ProductName	Category	Price	StockQuantity
1	Product A	Electronics	499.99	100

8. Query using clustered index (ProductID)

Query: SELECT * FROM Products WHERE ProductID = 2;

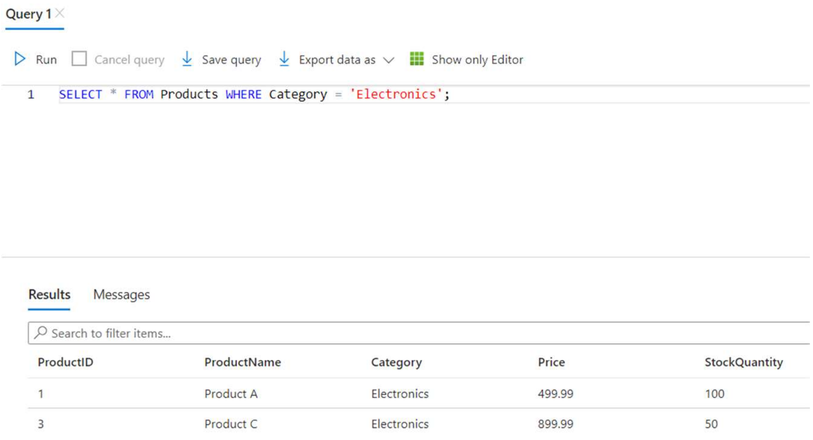


```
SELECT * FROM Products WHERE ProductID = 2;
```

ProductID	ProductName	Category	Price	StockQuantity
2	Product B	Clothing	39.99	250

9. Query using a non-clustered index (Category)

Query: SELECT * FROM Products WHERE Category = 'Electronics';

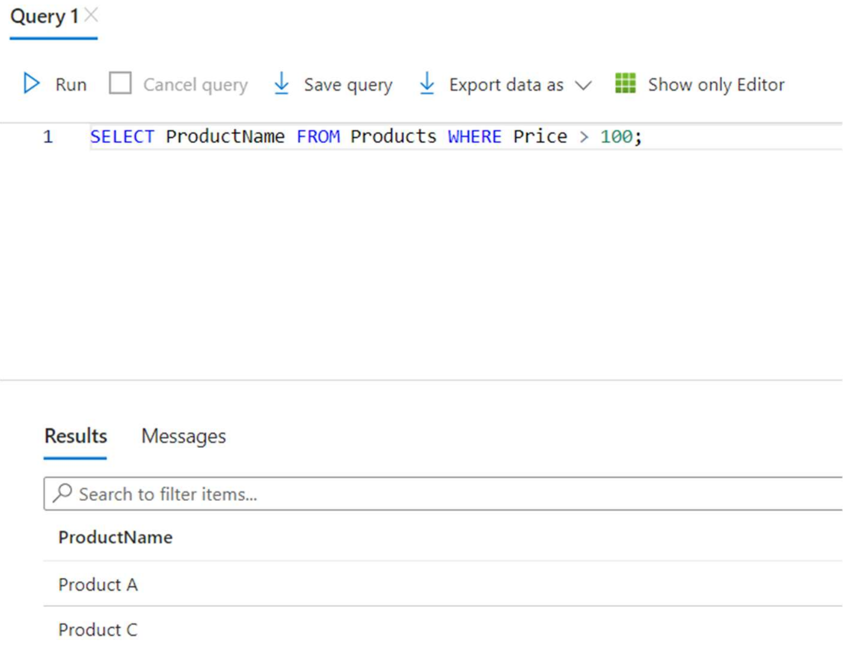


```
SELECT * FROM Products WHERE Category = 'Electronics';
```

ProductID	ProductName	Category	Price	StockQuantity
1	Product A	Electronics	499.99	100
3	Product C	Electronics	899.99	50

10. Query using columnstore index (Price).

Query: SELECT ProductName FROM Products WHERE Price > 100;



```sql
SELECT ProductName FROM Products WHERE Price > 100;
```

ProductName
Product A
Product C