ASSIGNMENT TITLE

ADVANCED SQL ASSIGNMENT

Submitted By : Bhaskar Singh Rajput

Course : Data Analytics With AI – September batch Live

Institude : PW Skills

Date : 23 December 2025

Email: bhaskarsinghrajput07@gmail.com

Q1. What is a Common Table Expression (CTE) and how does it improve SQL query readability?

Answer:

A Common Table Expression (CTE) is a temporary result set created using the `WITH` clause. It is mainly used to store the output of a query temporarily so that it can be reused in the main query.

CTE improves readability because it divides a complex SQL query into smaller logical parts. When queries become very long or contain multiple calculations, they become difficult to read.
CTE helps by giving a meaningful name to the intermediate result.

Example:
Suppose we want to find employees whose salary is greater than 60,000.

```
WITH HighSalaryEmployees AS (
    SELECT * FROM Employees WHERE Salary > 60000
)
SELECT * FROM HighSalaryEmployees;
```

Explanation:
First, the CTE stores employees having salary above 60,000.
Then the main query simply selects data from the CTE.
This is easier to understand compared to writing the same logic inside a nested subquery.

Q2. Why are some views updatable while others are read-only?

Answer:

Views are virtual tables created using SELECT statements.
Some views can be updated, while others cannot, depending on how they are created.

A view is **updatable** when:

- It is created using only one base table
- It does not use JOIN
- It does not contain GROUP BY or aggregate functions

If a view is created using joins or aggregation, SQL cannot identify which table row should be updated.
Therefore, such views become **read-only**.

Example (Updatable View):

```
CREATE VIEW vw_Products AS
```

```
SELECT ProductID, ProductName, Price
FROM Products;
```

This view is updatable because it directly represents one table.

Non-Updatable View Example:

```
CREATE VIEW vw_ProductSummary AS
SELECT Category, AVG(Price)
FROM Products
GROUP BY Category;
```

This view is read-only because it uses aggregation.

Q3. What advantages do stored procedures offer compared to writing raw SQL repeatedly?

Answer:

Stored procedures are SQL programs that are stored inside the database and executed whenever required.

Advantages of stored procedures:

- They improve performance because they are precompiled
- The same logic can be reused multiple times
- They reduce chances of writing wrong queries repeatedly
- Security is improved because users don’t access tables directly

Example:
Instead of writing:

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

again and again, we can create a stored procedure and reuse it.

Stored procedures make database operations efficient and organized.

Q4. What is the purpose of triggers in a database? Give one real-life use case.

Answer:

A trigger is a database object that automatically executes when an event such as INSERT, UPDATE, or DELETE occurs on a table.

Triggers are mainly used to:

- Maintain data consistency
- Automatically log changes
- Enforce business rules

Real-life use case:
If a product is deleted from a table, its details should not be lost.
A trigger can automatically insert the deleted data into an archive table.

This is useful for auditing and maintaining history.

Q5. Explain the need for data modelling and normalization.

Answer:

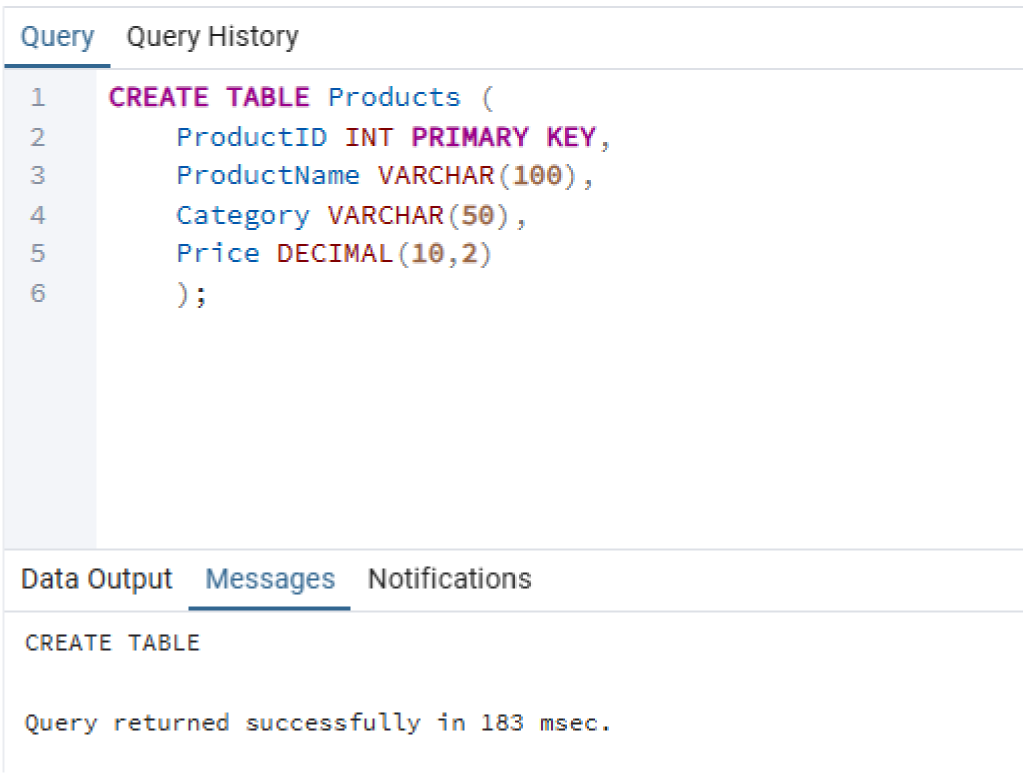
Data modelling defines how data is structured in tables and how tables are related to each other.
Normalization is the process of organizing data to avoid duplication.

Why they are important:

- Reduce data redundancy
- Prevent update anomalies
- Maintain data consistency
- Make database easy to maintain

Example:
Instead of storing department name repeatedly for each employee, we store department details in a separate table.
This avoids unnecessary repetition.

CREATE TABLE FOR PRODUCT:


Query Query History

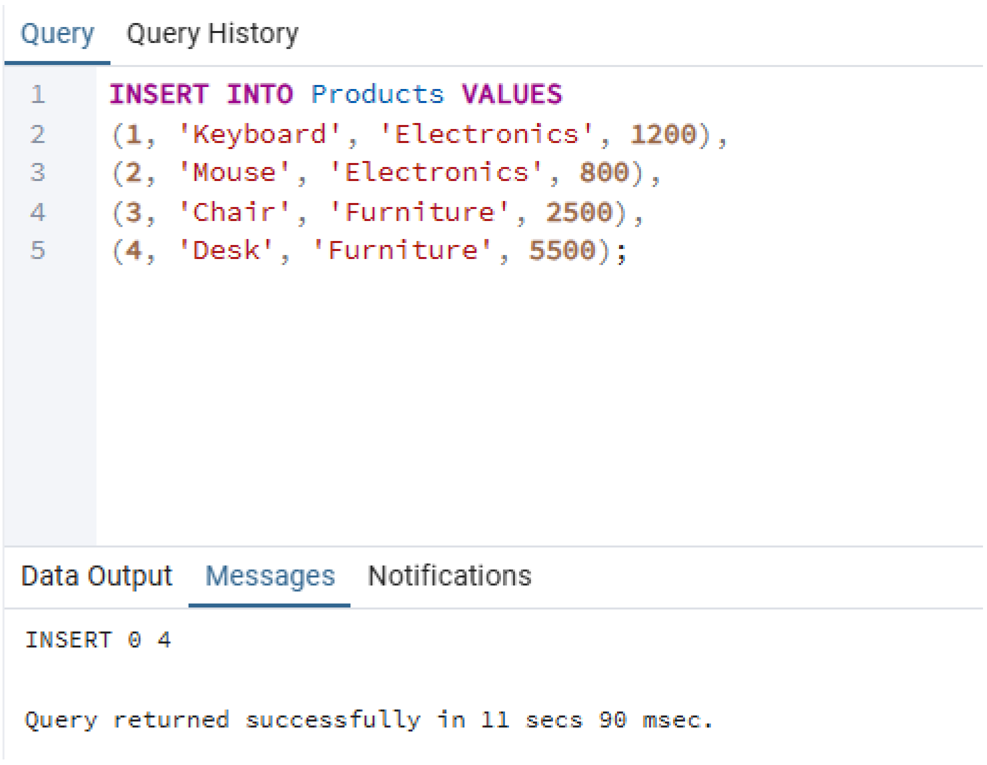
```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100),
    Category VARCHAR(50),
    Price DECIMAL(10,2)
    );
```

Data Output Messages Notifications

```
CREATE TABLE

Query returned successfully in 183 msec.
```

INSERT DATA:


Query Query History

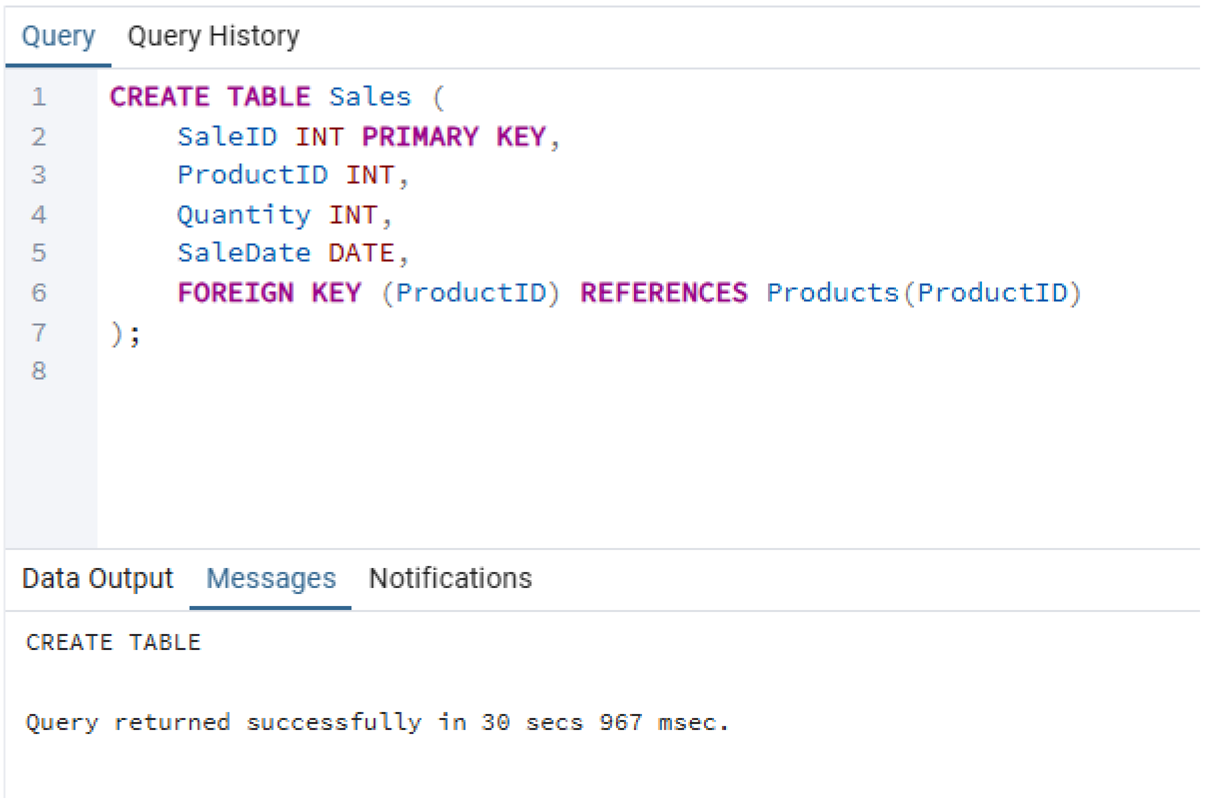
```
INSERT INTO Products VALUES
(1, 'Keyboard', 'Electronics', 1200),
(2, 'Mouse', 'Electronics', 800),
(3, 'Chair', 'Furniture', 2500),
(4, 'Desk', 'Furniture', 5500);
```

Data Output Messages Notifications

```
INSERT 0 4

Query returned successfully in 11 secs 90 msec.
```

CREATE TABLE FOR SALES:


Query Query History

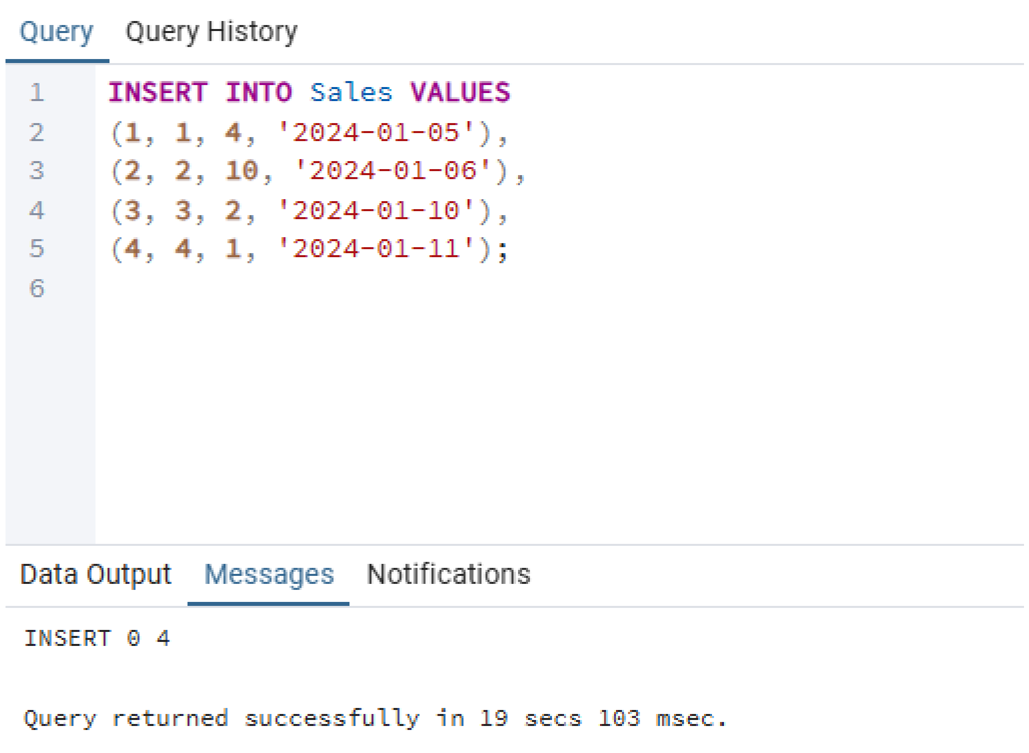
```sql
CREATE TABLE Sales (
    SaleID INT PRIMARY KEY,
    ProductID INT,
    Quantity INT,
    SaleDate DATE,
    FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
);
```

Data Output Messages Notifications

```
CREATE TABLE

Query returned successfully in 30 secs 967 msec.
```

INSERT DATA:


Query Query History

```sql
INSERT INTO Sales VALUES
(1, 1, 4, '2024-01-05'),
(2, 2, 10, '2024-01-06'),
(3, 3, 2, '2024-01-10'),
(4, 4, 1, '2024-01-11');
```

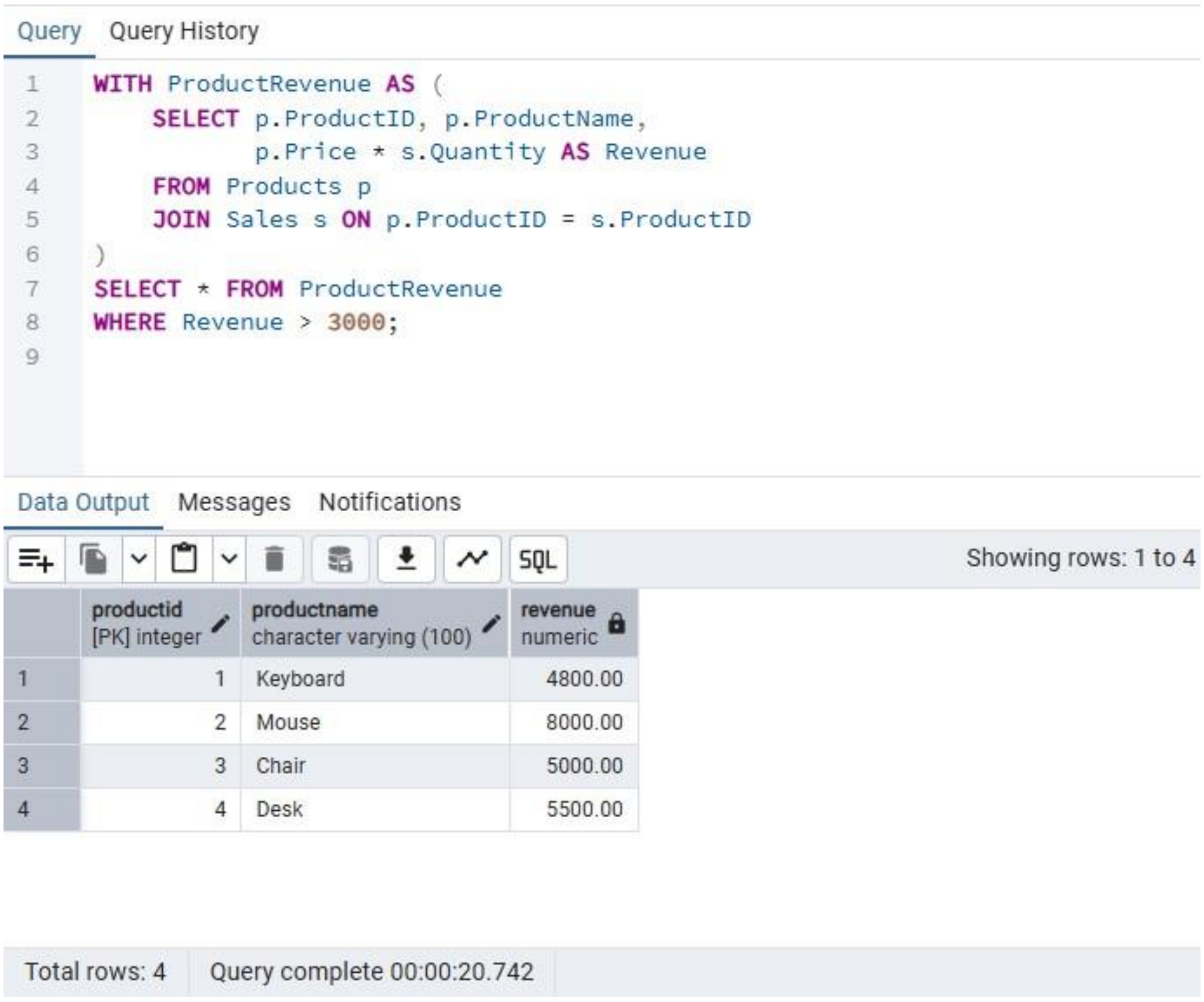
Data Output Messages Notifications

```
INSERT 0 4

Query returned successfully in 19 secs 103 msec.
```

Q6. CTE to calculate total revenue for each product where revenue > 3000

Query:



```
WITH ProductRevenue AS (
    SELECT p.ProductID, p.ProductName,
           p.Price * s.Quantity AS Revenue
    FROM Products p
    JOIN Sales s ON p.ProductID = s.ProductID
)
SELECT * FROM ProductRevenue
WHERE Revenue > 3000;
```

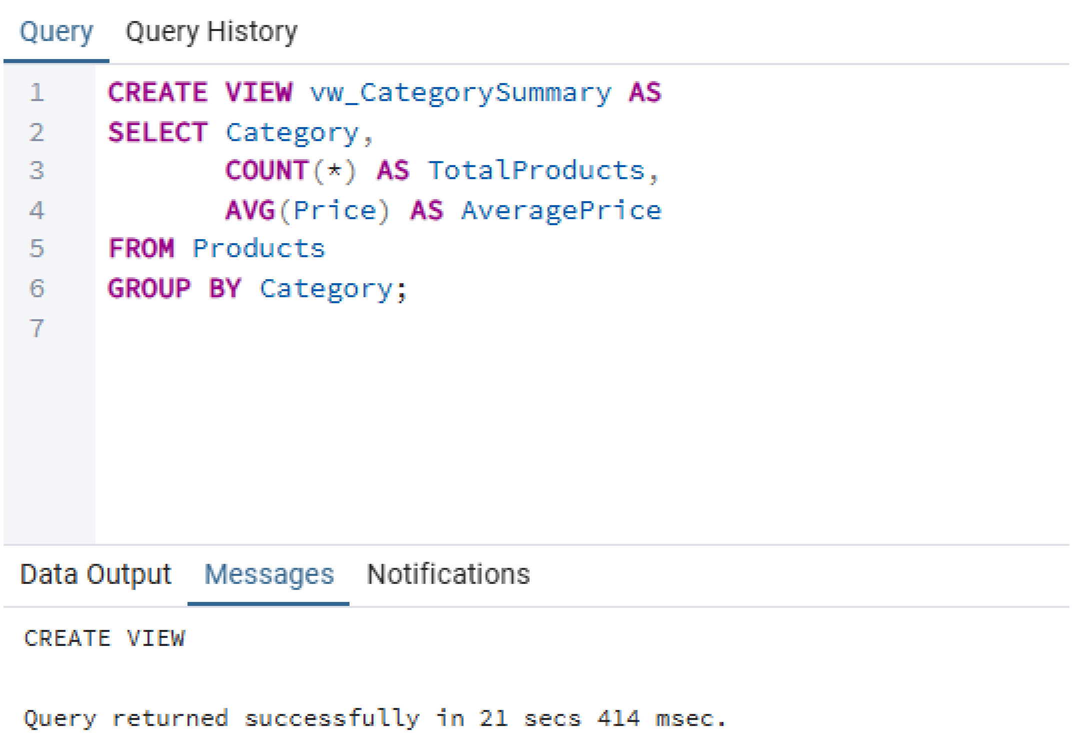
	productid [PK] integer	productname character varying (100)	revenue numeric
1	1	Keyboard	4800.00
2	2	Mouse	8000.00
3	3	Chair	5000.00
4	4	Desk	5500.00

Total rows: 4   Query complete 00:00:20.742

Explanation:
First, the CTE calculates revenue by multiplying price and quantity.
Then, the main query filters products whose revenue is greater than 3000.
Using CTE avoids repeating the calculation logic.

Q7. Create a view showing category-wise summary

Query:


Query Query History

```sql
CREATE VIEW vw_CategorySummary AS
SELECT Category,
       COUNT(*) AS TotalProducts,
       AVG(Price) AS AveragePrice
FROM Products
GROUP BY Category;
```

Data Output Messages Notifications

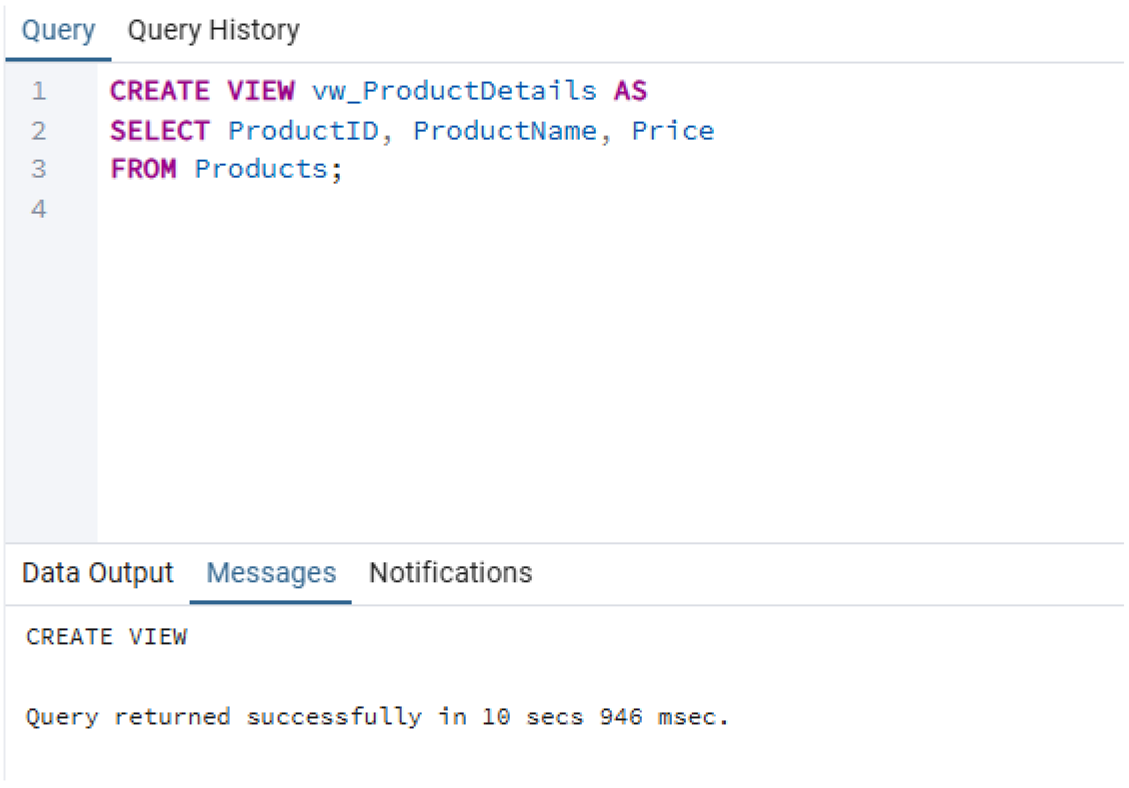
```
CREATE VIEW

Query returned successfully in 21 secs 414 msec.
```

Explanation:
This view shows how many products are available in each category and their average price. Such views are helpful for management reports.

Q8. Create an updatable view and update price

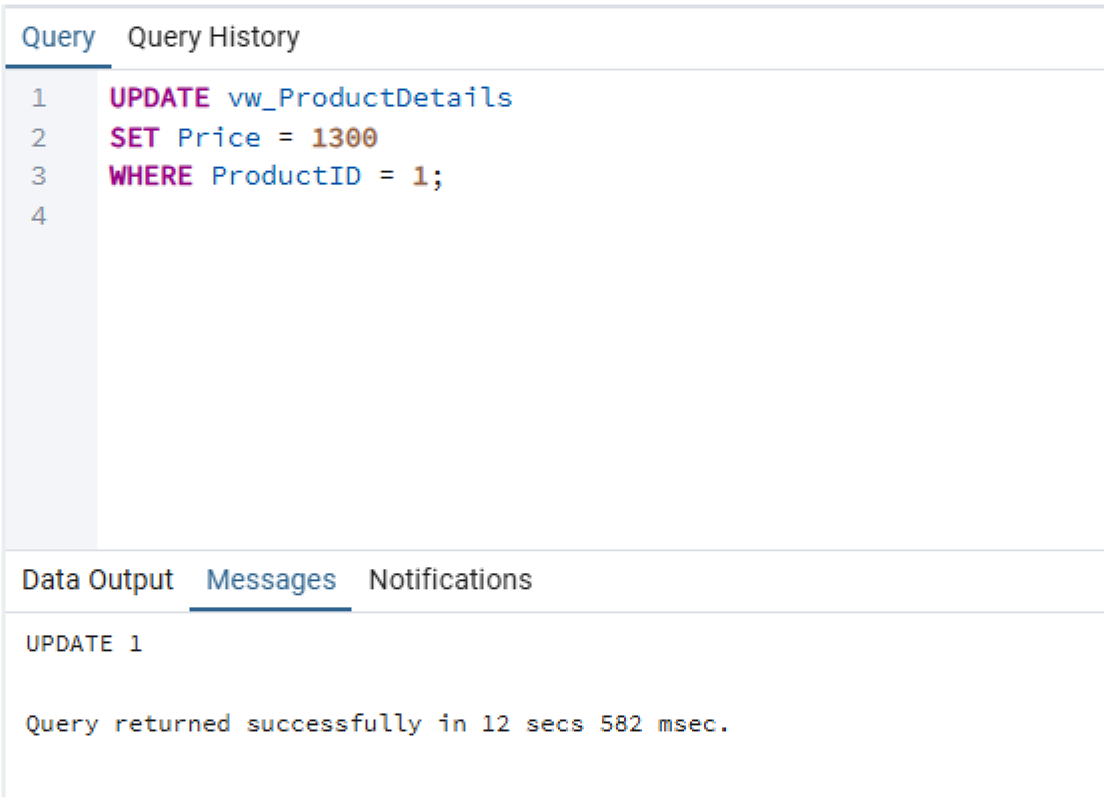
Query:



```sql
CREATE VIEW vw_ProductDetails AS
SELECT ProductID, ProductName, Price
FROM Products;
```

```
CREATE VIEW

Query returned successfully in 10 secs 946 msec.
```



```sql
UPDATE vw_ProductDetails
SET Price = 1300
WHERE ProductID = 1;
```

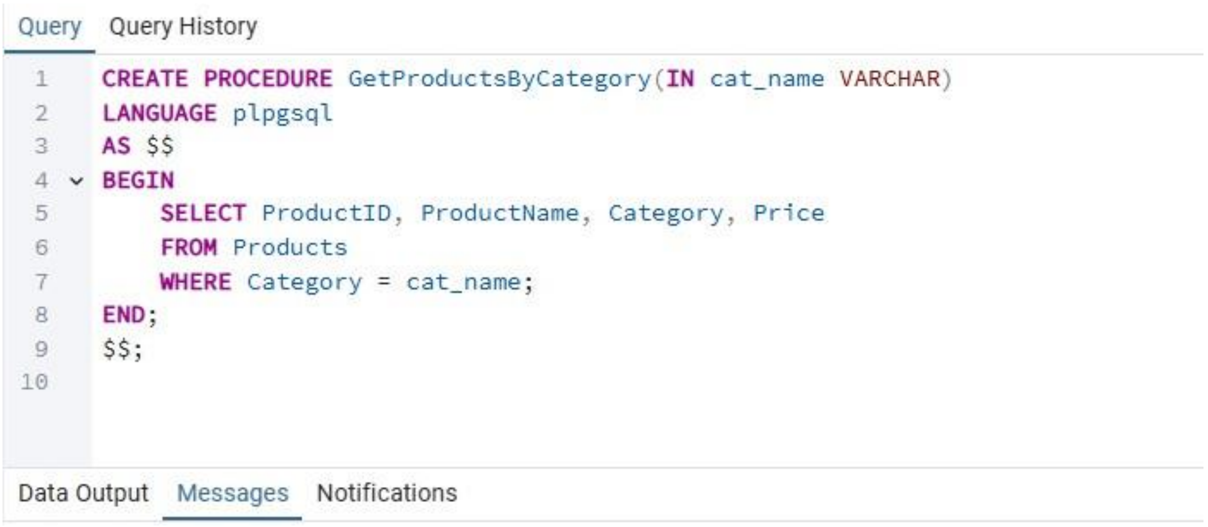
```
UPDATE 1

Query returned successfully in 12 secs 582 msec.
```

Explanation:
Since the view is based on one table and contains no aggregation, it is updatable.
Updating the view updates the base table automatically.

Q9. Create a stored procedure to return products by category.

Query:

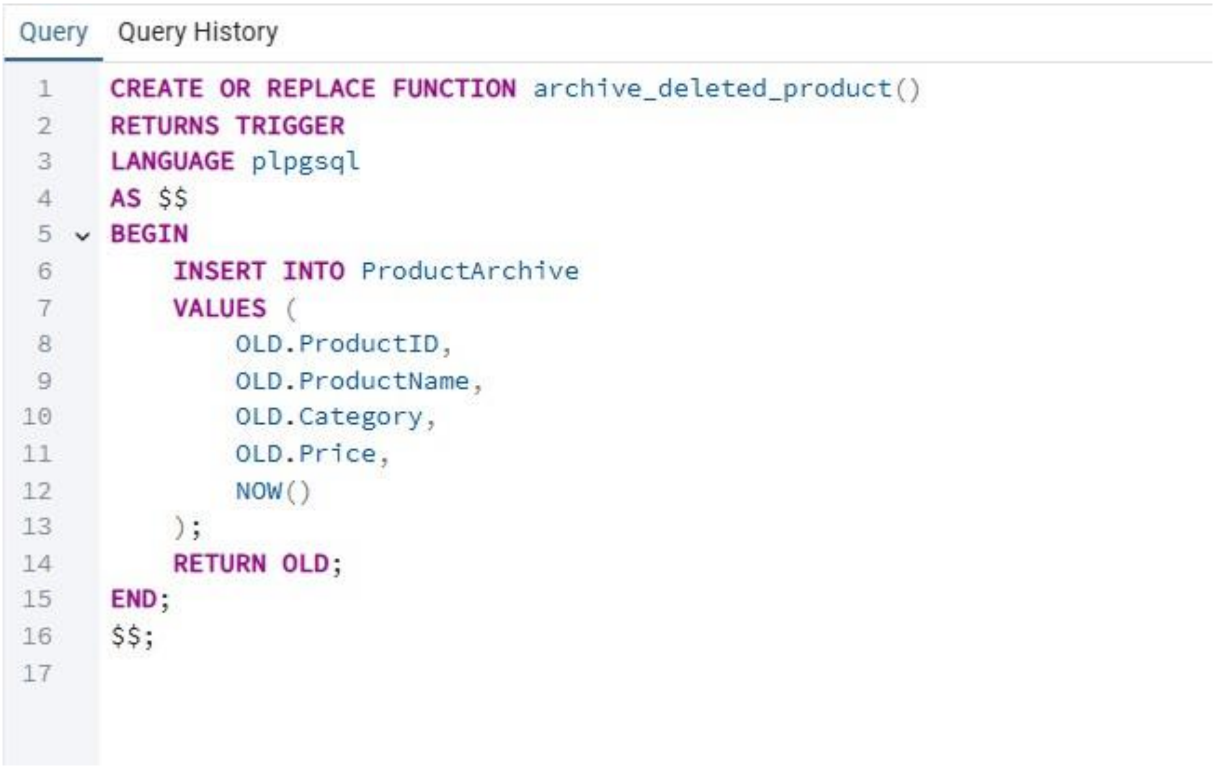


```sql
CREATE PROCEDURE GetProductsByCategory(IN cat_name VARCHAR)
LANGUAGE plpgsql
AS $$
BEGIN
    SELECT ProductID, ProductName, Category, Price
    FROM Products
    WHERE Category = cat_name;
END;
$$;
```

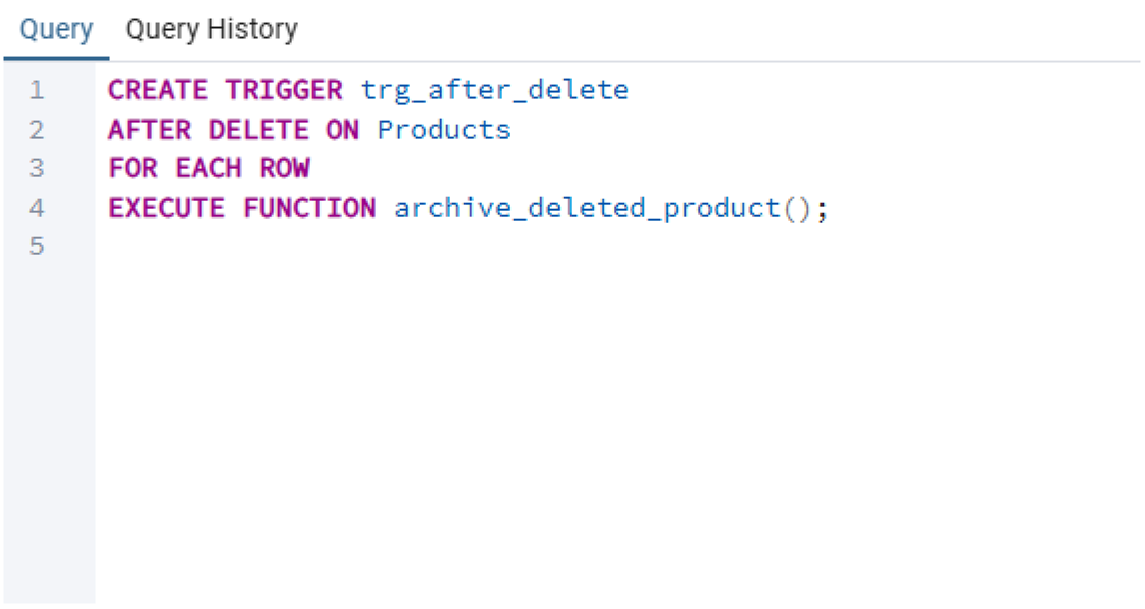
Explanation:
This procedure takes category name as input and returns products of that category.
It avoids writing the same query multiple times.

Q10. Create a trigger to archive deleted products.

Query:



```sql
CREATE OR REPLACE FUNCTION archive_deleted_product()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO ProductArchive
    VALUES (
        OLD.ProductID,
        OLD.ProductName,
        OLD.Category,
        OLD.Price,
        NOW()
    );
    RETURN OLD;
END;
$$;
```



```sql
CREATE TRIGGER trg_after_delete
AFTER DELETE ON Products
FOR EACH ROW
EXECUTE FUNCTION archive_deleted_product();
```

Explanation:
Whenever a product is deleted, the trigger stores its details in the archive table.
This ensures no important data is lost.