

## **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  
1  CREATE TABLE Products (  
2      ProductID INT PRIMARY KEY,  
3      ProductName VARCHAR(100),  
4      Category VARCHAR(50),  
5      Price DECIMAL(10,2)  
6  );
```

Data Output Messages Notifications

CREATE TABLE

Query returned successfully in 183 msec.

## INSERT DATA:

```
Query  Query History  
1  INSERT INTO Products VALUES  
2      (1, 'Keyboard', 'Electronics', 1200),  
3      (2, 'Mouse', 'Electronics', 800),  
4      (3, 'Chair', 'Furniture', 2500),  
5      (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

```
1 CREATE TABLE Sales (
2     SaleID INT PRIMARY KEY,
3     ProductID INT,
4     Quantity INT,
5     SaleDate DATE,
6     FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
7 );
8
```

Data Output Messages Notifications

CREATE TABLE

Query returned successfully in 30 secs 967 msec.

## INSERT DATA:

Query Query History

```
1 INSERT INTO Sales VALUES
2 (1, 1, 4, '2024-01-05'),
3 (2, 2, 10, '2024-01-06'),
4 (3, 3, 2, '2024-01-10'),
5 (4, 4, 1, '2024-01-11');
6
```

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:

Query Query History

```
1 WITH ProductRevenue AS (
2     SELECT p.ProductID, p.ProductName,
3             p.Price * s.Quantity AS Revenue
4     FROM Products p
5     JOIN Sales s ON p.ProductID = s.ProductID
6 )
7 SELECT * FROM ProductRevenue
8 WHERE Revenue > 3000;
9
```

Data Output Messages Notifications

The screenshot shows a database interface with a toolbar at the top containing various icons for file operations and SQL navigation. Below the toolbar is a table titled "Showing rows: 1 to 4". The table has four columns: "productid [PK] integer", "productname character varying (100)", and "revenue numeric". The data consists of four rows with the following values:

	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

```
1 CREATE VIEW vw_CategorySummary AS
2 SELECT Category,
3        COUNT(*) AS TotalProducts,
4        AVG(Price) AS AveragePrice
5 FROM Products
6 GROUP BY Category;
7
```

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:

Query Query History

```
1 CREATE VIEW vw_ProductDetails AS
2 SELECT ProductID, ProductName, Price
3 FROM Products;
4
```

Data Output Messages Notifications

CREATE VIEW

Query returned successfully in 10 secs 946 msec.

Query Query History

```
1 UPDATE vw_ProductDetails
2 SET Price = 1300
3 WHERE ProductID = 1;
4
```

Data Output Messages Notifications

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:**

```
Query  Query History
1 CREATE PROCEDURE GetProductsByCategory(IN cat_name VARCHAR)
2 LANGUAGE plpgsql
3 AS $$ 
4 BEGIN
5     SELECT ProductID, ProductName, Category, Price
6     FROM Products
7     WHERE Category = cat_name;
8 END;
9 $$;
10
```

Data Output Messages Notifications

**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:**

```
Query  Query History
1 CREATE OR REPLACE FUNCTION archive_deleted_product()
2 RETURNS TRIGGER
3 LANGUAGE plpgsql
4 AS $$ 
5 BEGIN
6     INSERT INTO ProductArchive
7     VALUES (
8         OLD.ProductID,
9         OLD.ProductName,
10        OLD.Category,
11        OLD.Price,
12        NOW()
13    );
14    RETURN OLD;
15 END;
16 $$;
17
```

Query Query History

```
1 CREATE TRIGGER trg_after_delete
2 AFTER DELETE ON Products
3 FOR EACH ROW
4 EXECUTE FUNCTION archive_deleted_product();
5
```

**Explanation:**

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