Advanced SQL
Assignment Questions

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 named result set defined using the WITH clause that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement.

How it improves readability:

- Breaks complex queries into logical, easy-to-understand parts
- Makes queries cleaner and more maintainable
- Avoids repeating subqueries
- Improves clarity, especially for hierarchical or aggregation logic

Q2. Why are some views updatable while others are read-only? Explain with an example.

Answer: A view is updatable only when it directly maps to a single base table without complexity.

Updatable views (conditions):

- Based on a single table
- No JOIN, GROUP BY, DISTINCT, HAVING, or aggregate functions

Example (Updatable View):

```
CREATE VIEW ElectronicsProducts AS
SELECT ProductID, ProductName, Price
FROM Products
WHERE Category = 'Electronics';
```

Read-only views:
Use joins, aggregations, or grouping
The database cannot determine how to update base tables

Example (Read-Only View):

```
CREATE VIEW ProductSales AS
SELECT ProductID, SUM(Quantity)
FROM Sales
GROUP BY ProductID;
```

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

Answer: Stored procedures provide:

- Better performance (precompiled and cached execution plans)
- Code reusability (write once, call many times)
- Improved security (restrict direct table access)
- Reduced network traffic (single procedure call)
- Centralized business logic for easier maintenance

Q4. What is the purpose of triggers in a database? Mention one use case where a trigger is essential.

Answer: A trigger is a database object that automatically executes in response to specific events like INSERT, UPDATE, or DELETE.

Purpose:

- Enforce business rules
- Maintain data integrity
- Automatically log or validate data
- Essential Use Case:
- Automatically recording changes in an audit table whenever a row is updated.

Example:

When an employee’s salary is updated, a trigger logs the old and new salary values.

Q5. Explain the need for data modelling and normalization when designing a database.

Answer: Data Modelling-

- Defines how data is structured and related
- Ensures clarity and scalability
- Helps represent real-world entities accurately

Normalization-

- Reduces data redundancy
- Prevents update, insert, and delete anomalies
- Improves data consistency and integrity

Together, they ensure an efficient, reliable, and maintainable database design.

Dataset (Use for Q6–Q9)

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

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

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

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');
```

Q6. Write a CTE to calculate the total revenue for each product (Revenues = Price × Quantity), and return only products where revenue > 3000.

Answer: Revenue = Price × Quantity

```sql
WITH ProductRevenue AS (
    SELECT
        p.ProductID,
        p.ProductName,
        SUM(p.Price * s.Quantity) AS Revenue
    FROM Products p
    JOIN Sales s
        ON p.ProductID = s.ProductID
    GROUP BY p.ProductID, p.ProductName
)
SELECT *
```

```
FROM ProductRevenue
WHERE Revenue > 3000;
```

Q7. Create a view named that shows: Category, TotalProducts, AveragePrice.

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

- This view summarizes the number of products and the average price per category.

Q8. Create an updatable view containing ProductID, ProductName, and Price. Then update the price of ProductID = 1 using the view.

Answer: Then update the price of ProductID = 1 using the view.**

Step 1: Create an updatable view

```
CREATE VIEW vw_ProductPrice AS
SELECT
```

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

✔ This view is updatable because it:

- Uses a single table
- Has no joins, aggregates, or groupings

Step 2: Update price using the view
Copy code
Sql

```
UPDATE vw_ProductPrice
SET Price = 1300
WHERE ProductID = 1;
```

Q9. Create a stored procedure that accepts a category name and returns all products belonging to that category.

Answer:

```
CREATE PROCEDURE GetProductsByCategory
    @CategoryName VARCHAR(50)
AS
BEGIN
    SELECT *
    FROM Products
```

```
    WHERE Category = @CategoryName;
END;
```

Example execution:

```
EXEC GetProductsByCategory 'Electronics';
```

Q10. Create an AFTER DELETE trigger on the Products table that archives deleted product rows into a new table ProductArchive. The archive should store ProductID, ProductName, Category, Price, and DeletedAt timestamp.

Answer: Step 1: Create archive table

```
CREATE TABLE ProductArchive (
    ProductID INT,
    ProductName VARCHAR(100),
    Category VARCHAR(50),
    Price DECIMAL(10,2),
    DeletedAt DATETIME
);
```

Step 2: Create AFTER DELETE trigger
Copy code
Sql

```
CREATE TRIGGER trg_AfterDelete_Products
ON Products
```

```
AFTER DELETE
AS
BEGIN
    INSERT INTO ProductArchive
    (ProductID, ProductName, Category, Price,
DeletedAt)
    SELECT
        ProductID,
        ProductName,
        Category,
        Price,
        GETDATE()
    FROM deleted;
END;
```

- This trigger automatically stores deleted product data with a timestamp.