1. Explain the difference between subquery, a CTE, and a stored procedure - link .

Subqueries, Common Table Expressions (CTEs), and stored procedures are SQL constructs used to manage and manipulate data in relational databases, but they serve different purposes and have distinct characteristics. Below, I'll explain each, highlight their differences, and provide examples using SQL syntax.

1. Subquery

- **Definition**: A subquery is a query nested within another query (often called the outer query). It's used to return data that the outer query processes. Subqueries are typically enclosed in parentheses and can appear in clauses like SELECT, WHERE, or FROM.
- **Use Case**: When you need a temporary result set for filtering, calculations, or comparisons within a single query.
- **Characteristics**:
    - Executes first, and its result is used by the outer query.
    - Can be correlated (depends on the outer query) or non-correlated (independent).
    - Single-use, not reusable across queries.
    - May impact performance for complex queries due to repeated execution.

Example: Suppose you have a Employees table and want to find employees with a salary above the average salary.

sql

```
SELECT EmployeeID, FirstName, Salary
FROM Employees
```

- `WHERE Salary > (SELECT AVG(Salary) FROM Employees);`

    - The subquery (SELECT AVG(Salary) FROM Employees) calculates the average salary, and the outer query filters employees whose salary exceeds it.

2. Common Table Expression (CTE)

- **Definition**: A CTE is a temporary result set defined within a single query using the WITH clause. It's named and can be referenced multiple times in the main query.
- **Use Case**: When you need a readable, reusable intermediate result within a query, especially for complex or recursive queries.
- **Characteristics**:
    - Improves query readability and maintainability compared to subqueries.
    - Can be referenced multiple times in the same query.
    - Supports recursion for hierarchical data (e.g., organizational charts).
    - Temporary and exists only for the duration of the query.

Example: Using the same Employees table, calculate the average salary and use it to filter employees.

sql

```
WITH AvgSalary AS (
    SELECT AVG(Salary) AS AvgSal
    FROM Employees
)
SELECT EmployeeID, FirstName, Salary
FROM Employees, AvgSalary
```

- WHERE Salary > AvgSalary.AvgSal;
    - The CTE AvgSalary computes the average salary, and the main query references it. This is more readable than a subquery.

3. Stored Procedure

- **Definition**: A stored procedure is a precompiled set of SQL statements stored in the database with a name, which can be executed on demand. It can accept parameters and include logic like loops or conditionals.
- **Use Case**: For reusable, complex operations that need to be executed repeatedly, such as data processing, reporting, or business logic.
- **Characteristics**:
    - Stored in the database and reusable across multiple sessions or applications.
    - Can include procedural logic (e.g., IF, WHILE) and error handling.
    - Can accept input/output parameters for dynamic behavior.
    - Improves performance (precompiled) and security (controlled access).

Example: Create a stored procedure to update an employee's salary based on their EmployeeID.

sql

```
CREATE PROCEDURE UpdateEmployeeSalary
    @EmployeeID INT,
    @NewSalary DECIMAL(10, 2)
AS
BEGIN
    UPDATE Employees
    SET Salary = @NewSalary
    WHERE EmployeeID = @EmployeeID;
END;

-- Execute the stored procedure
```

- EXEC UpdateEmployeeSalary @EmployeeID = 101, @NewSalary = 75000.00;

- The procedure UpdateEmployeeSalary takes two parameters and updates the Employees table. It can be reused anytime by calling EXEC.

2. Similarities between a stored procedure sql and a python function- link

Stored procedures in SQL and Python functions share several similarities, as both are designed to encapsulate reusable logic, improve modularity, and streamline tasks. Below, I outline their key similarities, with examples to illustrate.

Similarities

1. **Reusability:**

Both are defined once and can be called multiple times in different contexts, reducing code duplication.

SQL Stored Procedure Example:

```
CREATE PROCEDURE GetEmployeeDetails
    @EmployeeID INT
AS
BEGIN
    SELECT EmployeeID, FirstName, Salary
    FROM Employees
    WHERE EmployeeID = @EmployeeID;
END;

-- Call the procedure
EXEC GetEmployeeDetails @EmployeeID = 101;
```

Python Function Example:

```
def get_employee_details(employee_id):
    query = f"SELECT EmployeeID, FirstName, Salary FROM Employees WHERE EmployeeID
= {employee_id}"
    # Assume database connection and execution
    return execute_query(query)

# Call the function
result = get_employee_details(101)
```

Both allow the same logic to be reused with different inputs (e.g., EmployeeID).

2. **Parameter Support:**

Both accept input parameters to make them dynamic and flexible, and some stored procedures can return output parameters, similar to how Python functions return values.

SQL Stored Procedure Example:

```
sql
CREATE PROCEDURE UpdateSalary
    @EmployeeID INT,
    @NewSalary DECIMAL(10, 2),
    @Updated BIT OUTPUT
AS
BEGIN
    UPDATE Employees
    SET Salary = @NewSalary
    WHERE EmployeeID = @EmployeeID;
    SET @Updated = 1;
END;

-- Call with output parameter
DECLARE @Result BIT;
EXEC UpdateSalary @EmployeeID = 101, @NewSalary = 75000.00, @Updated = @Result
OUTPUT;
SELECT @Result AS UpdateStatus;
```

Python Function Example:

```
def update_salary(employee_id, new_salary):
    query = f"UPDATE Employees SET Salary = {new_salary} WHERE EmployeeID =
{employee_id}"
    execute_query(query)
    return True  # Indicates success

# Call the function
success = update_salary(101, 75000.00)
print(success)  # Output: True
```

Both use parameters (@EmployeeID, employee_id) to customize behavior and can return results.

3. **Encapsulation of Logic:**

Both encapsulate a block of code to perform a specific task, making code modular and easier to maintain.

SQL Stored Procedure Example:

```
CREATE PROCEDURE CalculateBonus
   @DepartmentID INT
AS
BEGIN
   UPDATE Employees
   SET Salary = Salary + (Salary * 0.1)
   WHERE DepartmentID = @DepartmentID;
END;
```

Python Function Example:

```
def calculate_bonus(department_id):
   query = f"UPDATE Employees SET Salary = Salary + (Salary * 0.1) WHERE DepartmentID =
{department_id}"
   execute_query(query)
```

Both encapsulate the logic for applying a 10% salary bonus to a department.

4. **Modular Structure:**

Both allow breaking down complex tasks into smaller, manageable units that can be tested and debugged independently.

SQL Stored Procedure Example:

```
CREATE PROCEDURE GenerateReport
   @Year INT
AS
BEGIN
   SELECT DepartmentID, COUNT(*) AS EmployeeCount, AVG(Salary) AS AvgSalary
   FROM Employees
   WHERE YEAR(HireDate) = @Year
   GROUP BY DepartmentID;
END;
```

Python Function Example:

```
def generate_report(year):
    query = f"SELECT DepartmentID, COUNT(*) AS EmployeeCount, AVG(Salary) AS AvgSalary
FROM Employees WHERE YEAR(HireDate) = {year} GROUP BY DepartmentID"
    return execute_query(query)
```

Both modularize the task of generating a department-wise report for a given year.

5. **Error Handling:**

Both support mechanisms to handle errors, ensuring robust execution.

SQL Stored Procedure Example:

```
CREATE PROCEDURE SafeUpdateSalary
    @EmployeeID INT,
    @NewSalary DECIMAL(10, 2)
AS
BEGIN
    BEGIN TRY
        UPDATE Employees
        SET Salary = @NewSalary
        WHERE EmployeeID = @EmployeeID;
        SELECT 'Update successful' AS Message;
    END TRY
    BEGIN CATCH
        SELECT ERROR_MESSAGE() AS ErrorMessage;
    END CATCH
END;
```

Python Function Example:

```
def safe_update_salary(employee_id, new_salary):
    try:
        query = f"UPDATE Employees SET Salary = {new_salary} WHERE EmployeeID =
{employee_id}"
        execute_query(query)
        return "Update successful"
    except Exception as e:
        return f"Error: {str(e)}"
```

Both handle errors gracefully, returning success or error messages.
Named Constructs:

Both are named entities (PROCEDURE in SQL, def in Python) that can be invoked by their names, improving code organization.
SQL: EXEC GetEmployeeDetails 101
Python: get_employee_details(101)

Key Notes

- **Execution Context:** Stored procedures run on the database server, optimizing performance for data-intensive tasks. Python functions run in the application layer, offering flexibility for non-database tasks but requiring database connectivity (e.g., via libraries like pyodbc or sqlalchemy).
- **Language Scope:** Stored procedures are limited to SQL and database-specific procedural extensions (e.g., T-SQL for SQL Server). Python functions support broader programming constructs (loops, data structures, external API calls).

Use Case:

- **Stored procedures** are ideal for database-centric operations (e.g., batch updates, complex joins).
- **Python functions** are better for general-purpose logic, data processing, or integrating with other systems.

When to Use Each

- **Stored Procedure:** Use for database-heavy tasks, security (controlled access), or when performance is critical due to server-side execution.
- **Python Function:** Use for application logic, cross-system integration, or when you need Python's extensive libraries (e.g., for data analysis or machine learning).