SQL Assignment 4

Question 1:- Explain different types of views. Demonstrate with suitable examples

Answer :- Besides simple views, there are other types of views, including:

Complex Views: These views involve more complex logic, such as using aggregate functions, GROUP BY, JOIN operations on multiple tables, or subqueries.

Materialized Views: Unlike standard views (which are essentially stored queries), materialized views store the actual result set of the query. This can improve performance for complex queries that are frequently accessed, but they require storage space and need to be refreshed periodically to reflect changes in the underlying tables.

Dynamic Views: This term can sometimes be used to refer to standard views that are based on underlying data that changes frequently, or in some database systems, it might refer to views that are created temporarily or whose definition can change based on parameters.

Let's demonstrate a complex view using our employees table and a hypothetical departments table.

```
cursor.execute('''
CREATE TABLE departments (
    department_id INTEGER PRIMARY KEY,
    department_name TEXT
)
''')

# Insert some sample data into departments
cursor.execute("INSERT INTO departments VALUES (1, 'Sales')")
cursor.execute("INSERT INTO departments VALUES (2, 'IT')")

conn.commit()
```

```
print("Table 'departments' created and populated.")
```

Now, let's create a complex view that joins the `employees` and `departments` tables to show employee names along with their department names.

```
# Create a complex view
cursor.execute('''
CREATE VIEW employee_department_view AS
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
JOIN departments d ON e.department = d.department_name
''')

print("View 'employee_department_view' created."
```

Querying the complex view:

```
# Query the complex view
cursor.execute("SELECT * FROM employee_department_view")
rows = cursor.fetchall()

# Display the results
df = pd.DataFrame(rows, columns=[description[0] for description in
cursor.description])
display(df)
```

Question 2:- What is the difference between function and stored procedure? Write syntax for creating functions and stored procedures.

Answer :- Let's go over the differences between functions and stored procedures and their syntax.

Here's a summary of the key distinctions:

*   **Return Value:** Functions are designed to return a single value and **must** have a `RETURN` statement. Stored procedures **can**

return multiple values through `OUT` or `INOUT` parameters, or they might not return any value at all.
*   **Usage in SQL:** Functions can be called directly within SQL statements (e.g., in `SELECT`, `WHERE`, `HAVING` clauses). Stored procedures **cannot** be used in this way; they are executed using `EXEC` or `CALL` statements.
*   **Transaction Management:** Stored procedures have the ability to manage transactions (e.g., using `COMMIT` and `ROLLBACK`). Functions generally **cannot** manage transactions independently; they are typically part of a larger transaction initiated by the calling statement or procedure.
*   **Side Effects:** Functions are generally expected to be deterministic and have no side effects (they should not modify data). Stored procedures **can** perform data modification operations (insert, update, delete).
*   **Parameters:** Both can accept input parameters. Stored procedures can also have output (`OUT`) and input/output (`INOUT`) parameters.

Now, let's look at the general syntax for creating them. Keep in mind that the exact syntax can vary depending on the specific database system (e.g., SQL Server, PostgreSQL, MySQL, Oracle).
General Syntax

Function:
```
CREATE [OR REPLACE] FUNCTION function_name (parameter1 datatype,
parameter2 datatype, ...)
RETURNS return_datatype
[LANGUAGE language_name]
AS $$
-- Function body (SQL statements)
-- MUST include a RETURN statement
$$;
```
Stored Procedure:

```
CREATE [OR REPLACE] PROCEDURE procedure_name (parameter1
[IN/OUT/INOUT] datatype, parameter2 [IN/OUT/INOUT] datatype, ...)
[LANGUAGE language_name]
AS $$
-- Procedure body (SQL statements)
-- May include COMMIT or ROLLBACK
$$;
```

```
-- Example Function (PostgreSQL)
CREATE OR REPLACE FUNCTION get_employee_full_name (emp_id INT)
RETURNS TEXT
LANGUAGE plpgsql
AS $$
DECLARE
    full_name TEXT;
BEGIN
    SELECT first_name || ' ' || last_name INTO full_name FROM
employees WHERE employee_id = emp_id;
    RETURN full_name;
END;
$$;
```

```
-- Example Stored Procedure (PostgreSQL)
CREATE OR REPLACE PROCEDURE add_new_employee (
    IN p_first_name TEXT,
    IN p_last_name TEXT,
    IN p_department TEXT,
    IN p_salary REAL,
    OUT p_employee_id INT
)
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO employees (first_name, last_name, department,
salary)
    VALUES (p_first_name, p_last_name, p_department, p_salary)
    RETURNING employee_id INTO p_employee_id; -- Get the newly
generated ID
    COMMIT; -- Commit the transaction
END;
```

```
$$;
```

Question 3:- What is an index in SQL? What are the different types of indexes in SQL?

Answer :- An index in SQL is a database structure that helps speed up data retrieval operations by providing quick access to rows in a table without scanning the entire table. It works similarly to an index in a book, allowing the database search engine to quickly locate specific data. While indexes improve the performance of `SELECT` queries, they can slow down `UPDATE` and `INSERT` operations.

Here are different types of indexes in SQL:

- **Clustered Index**: This type of index sorts and stores the data rows of a table or view based on their key values. A table can only have one clustered index, as it dictates the physical storage order of the data. It is ideal for range queries and ordered data retrieval.
- **Non-Clustered Index**: This index is a separate structure from the data rows and contains non-clustered key values with pointers to the actual data rows. A table can have multiple non-clustered indexes, and they do not affect the physical order of the data. They are useful for quick lookups and searches on non-primary key columns.
- **Unique Index**: This index guarantees that the indexed column(s) do not contain duplicate values, ensuring that each row in the table is unique. It is often automatically created by `PRIMARY KEY` and `UNIQUE` constraints.
- **Filtered Index**: This is an optimized non-clustered index that indexes a subset of rows in a table based on a filter predicate. It is useful when a column has a small number of relevant values for specific queries.
- **Columnstore Index**: Designed for data warehousing workloads, this index stores and manages data using column-based storage, which can significantly improve query performance for large datasets.
- **Hash Index**: This index uses a hash function to map keys to buckets or slots, each containing a pointer to a row.
- **Single-Column Index**: An index created on only one column of a table.
- **Composite Index**: An index created on two or more columns of a table.
- **Implicit Index**: These are indexes automatically created by the database server, for example, when primary key and unique constraints are applied to a table.

Question 4:- Showcase an example of exception handling in SQL stored procedure.

Answer :- Here's an example of exception handling in an SQL stored procedure, using PostgreSQL syntax:

```
CREATE OR REPLACE PROCEDURE transfer_funds (
   IN p_sender_account_id INT,
   IN p_receiver_account_id INT,
   IN p_amount NUMERIC
)
LANGUAGE plpgsql
AS $$
BEGIN
   -- Start a transaction
   BEGIN
      -- Deduct amount from sender's account
      UPDATE accounts
      SET balance = balance - p_amount
      WHERE account_id = p_sender_account_id;

      -- Check if sender has sufficient funds (or if the account exists)
      IF NOT FOUND THEN
          RAISE EXCEPTION 'Sender account % not found or insufficient
funds.', p_sender_account_id;
      END IF;

      -- Add amount to receiver's account
      UPDATE accounts
      SET balance = balance + p_amount
      WHERE account_id = p_receiver_account_id;

      -- Check if receiver account exists
      IF NOT FOUND THEN
```

```
              RAISE EXCEPTION 'Receiver account % not found.',
p_receiver_account_id;
    END IF;

    -- Commit the transaction if all operations are successful
    COMMIT;

  EXCEPTION
    -- Handle any exceptions that occur within the transaction
    WHEN OTHERS THEN
      -- Rollback the transaction to undo any changes
      ROLLBACK;
      -- Re-raise the exception to inform the caller
          RAISE NOTICE 'An error occurred during fund transfer: %',
SQLERRM;
      RAISE;
  END;
END;
$$;
```

Explanation:

- **`CREATE OR REPLACE PROCEDURE transfer_funds (...)`**: Defines a stored procedure named `transfer_funds` that takes sender and receiver account IDs, and the amount to transfer as input parameters.
- **`LANGUAGE plpgsql`**: Specifies that the procedure is written in PL/pgSQL, PostgreSQL's procedural language.
- **`BEGIN ... END;`**: Encloses the main body of the procedure.
- **`BEGIN ... EXCEPTION WHEN OTHERS THEN ... END;`**: This is the core of exception handling.
  - The `BEGIN` block inside the procedure defines a transaction.
  - **`UPDATE accounts SET balance = balance - p_amount WHERE account_id = p_sender_account_id;`**: This statement attempts to deduct the specified amount from the sender's account.
  - **`IF NOT FOUND THEN RAISE EXCEPTION ... END IF;`**: This checks if the `UPDATE` statement affected any rows. If `NOT FOUND` is true, it means either the sender's account ID doesn't exist or the sender has insufficient funds (if a `CHECK`

constraint is in place for negative balances), and it raises a custom exception.

- `UPDATE accounts SET balance = balance + p_amount WHERE account_id = p_receiver_account_id;`: This statement attempts to add the amount to the receiver's account.
- `IF NOT FOUND THEN RAISE EXCEPTION ... END IF;`: Similar to the sender, this checks if the receiver's account exists.
- `COMMIT;`: If all operations within the `BEGIN` block are successful, the transaction is committed, making the changes permanent.
- `EXCEPTION WHEN OTHERS THEN ...`: This block catches any exception that occurs within the `BEGIN` block.
  - `ROLLBACK;`: If an exception is caught, the transaction is rolled back, undoing any changes made during the current transaction, ensuring data consistency.
  - `RAISE NOTICE 'An error occurred during fund transfer: %', SQLERRM;`: This logs a notice message with the error details (`SQLERRM` provides the error message).
  - `RAISE;`: This re-raises the original exception, allowing the calling environment to be aware of the error.

This example demonstrates how to use `BEGIN...EXCEPTION...END` blocks to manage transactions and handle errors gracefully within an SQL stored procedure, ensuring that either all operations succeed or all are rolled back in case of a failure.

Question5 :- Create a SQL function to split strings into rows on a given character?

Input String: Stephen;peter;berry;Olivier;caroline;

Stephen
Peter
Berry
Oliver
Caroline

Answer :- Here's a SQL function (for PostgreSQL, which supports `unnest` with `string_to_array`) to split a string into rows based on a given character:

```
CREATE OR REPLACE FUNCTION split_string_to_rows(
    p_input_string TEXT,
    p_delimiter TEXT
)
```

```
RETURNS TABLE (
    split_value TEXT
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY SELECT unnest(string_to_array(p_input_string, p_delimiter));
END;
$$;

-- Example Usage:
SELECT split_value
FROM split_string_to_rows('Stephen;peter;berry;Olivier;caroline;', ';');
```

Explanation:

- **`CREATE OR REPLACE FUNCTION split_string_to_rows(...)`**: This defines a new function named `split_string_to_rows` that takes two `TEXT` parameters: `p_input_string` (the string to split) and `p_delimiter` (the character to split by).
- **`RETURNS TABLE (split_value TEXT)`**: This specifies that the function will return a table with a single column named `split_value` of type `TEXT`.
- **`LANGUAGE plpgsql`**: This indicates that the function body is written in PL/pgSQL, PostgreSQL's procedural language.
- **`string_to_array(p_input_string, p_delimiter)`**: This built-in PostgreSQL function takes the input string and the delimiter, and converts the string into an array of strings.
- **`unnest(...)`**: This function expands an array into a set of rows. So, each element of the array created by `string_to_array` becomes a separate row in the result set.

Output for the example usage:

split_value
Stephen
peter
berry
Olivier
caroline

Question 6:- What is a temporary and a variable table? Write suitable syntax to create temporary tables and variable tables.
Answer :- Here's a breakdown of temporary and variable tables in SQL, along with their syntax:Temporary Tables

Temporary tables are actual tables that are physically created in the `tempdb` database. They are useful for storing intermediate results when you need full DDL (Data Definition Language) or DML (Data Manipulation Language) access, or when dealing with large datasets.

Key Characteristics:

- **Storage:** Stored in the `tempdb` database.
- **Indexes:** You can create indexes on temporary tables, which can improve performance for large amounts of data.
- **Transactions:** Temporary tables honor transactions, meaning changes can be rolled back.
- **Schema Modification:** Allow schema modifications (e.g., `ALTER`, `CREATE`, `DROP`).
- **Scope:**
    - **Local Temporary Tables (`#temp`):** Visible only to the current session and are automatically dropped when the session ends. If created within a stored procedure, they are visible in child routines but not outside the stored procedure.
    - **Global Temporary Tables (`##temp`):** Visible to all sessions and are dropped automatically when the creating session terminates and no other sessions are using them.

Syntax for Creating Temporary Tables:
Local Temporary Table:
```
CREATE TABLE #TableName (
   Column1 DataType,
   Column2 DataType
);
```
Example:
```
CREATE TABLE #StudentTemp (
   StudentID INT,
   Name VARCHAR(50),
   Address VARCHAR(150)
```
-);

Global Temporary Table:
```
CREATE TABLE ##TableName (
   Column1 DataType,
   Column2 DataType
);
```
Example:
```
CREATE TABLE ##Global_Table_Name (
   ProductID INT,
   ProductName VARCHAR(100)
```

-);

Table Variables

Table variables are created like other variables using the `DECLARE` statement and are primarily designed for smaller datasets. While often thought to exist only in memory, they also reside in the `tempdb` database if there is memory pressure.

Key Characteristics:

- **Storage:** Created in memory, but can be pushed to `tempdb` under memory pressure.
- **Indexes:** Cannot explicitly create indexes, but can have implicit indexes through `PRIMARY KEY` or `UNIQUE` constraints.
- **Transactions:** Do not participate in transactions, logging, or locking, which can make them faster for small operations.
- **Schema Modification:** Do not allow schema modifications after declaration.
- **Scope:** Only visible within the current batch or stored procedure where they are declared. They are automatically dropped when the batch execution completes or the stored procedure finishes.
- **Functions:** Can be passed as parameters to functions and stored procedures, and can be used within user-defined functions.

Syntax for Creating Table Variables:
```
DECLARE @TableName TABLE (
   Column1 DataType [CONSTRAINT ConstraintName PRIMARY KEY/UNIQUE],
   Column2 DataType
);
```
Example:
```
DECLARE @TStudent TABLE (
   RollNo INT IDENTITY(1,1) PRIMARY KEY,
   StudentID INT,
   Name VARCHAR(50)
);
```