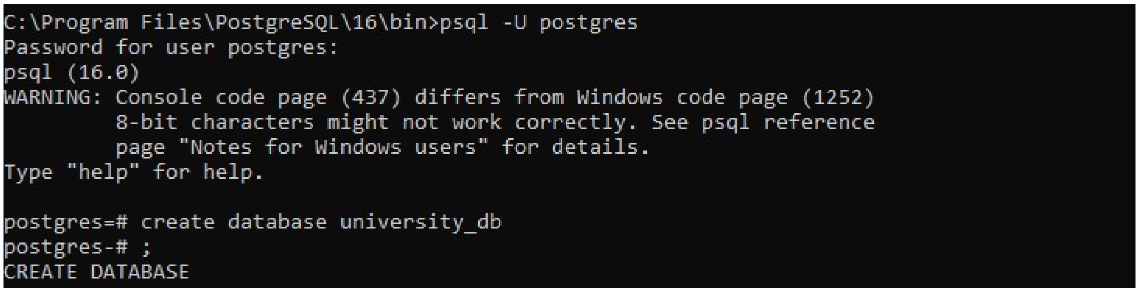
PostgreSQL Assignment

Database Setup:

- Create a fresh database titled "**university_db**" or any other appropriate name.



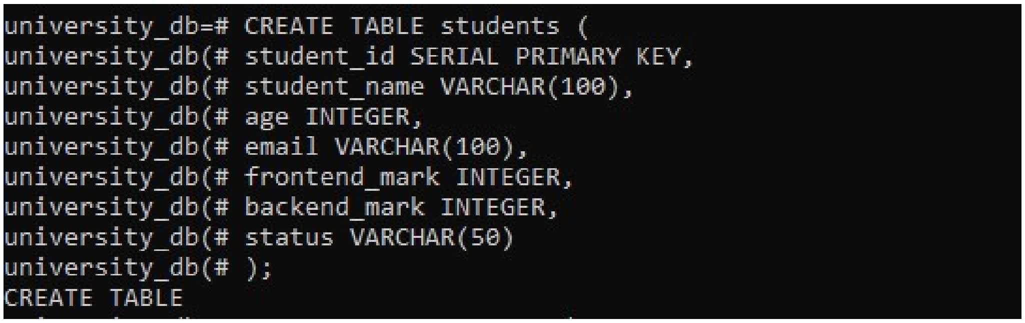
```
C:\Program Files\PostgreSQL\16\bin>psql -U postgres
Password for user postgres:
psql (16.0)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=# create database university_db
postgres-# ;
CREATE DATABASE
```

Table Creation:

Create a "**students**" table with the following fields:

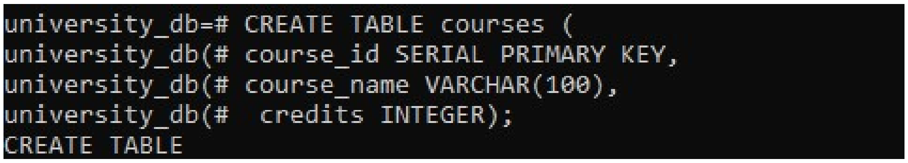
- student_id (Primary Key): Integer, unique identifier for students.
- student_name: String, representing the student's name.
- age: Integer, indicating the student's age.
- email: String, storing the student's email address.
- frontend_mark: Integer, indicating the student's frontend assignment marks.
- backend_mark: Integer, indicating the student's backend assignment marks.
- status: String, storing the student's result status.



```
university_db=# CREATE TABLE students (
university_db(# student_id SERIAL PRIMARY KEY,
university_db(# student_name VARCHAR(100),
university_db(# age INTEGER,
university_db(# email VARCHAR(100),
university_db(# frontend_mark INTEGER,
university_db(# backend_mark INTEGER,
university_db(# status VARCHAR(50)
university_db(# );
CREATE TABLE
```

Create a "**courses**" table with the following fields:

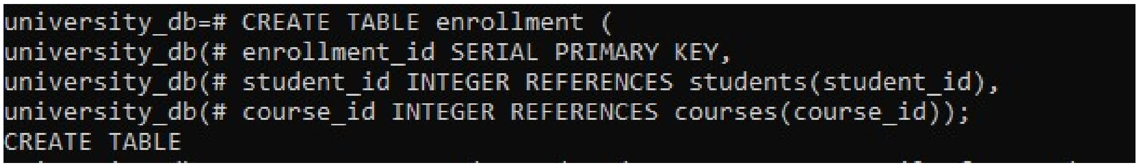
- course_id (Primary Key): Integer, unique identifier for courses.
- course_name: String, indicating the course's name.
- credits: Integer, signifying the number of credits for the course.



```
university_db=# CREATE TABLE courses (
university_db(# course_id SERIAL PRIMARY KEY,
university_db(# course_name VARCHAR(100),
university_db(#  credits INTEGER);
CREATE TABLE
```

Create an "**enrollment**" table with the following fields:

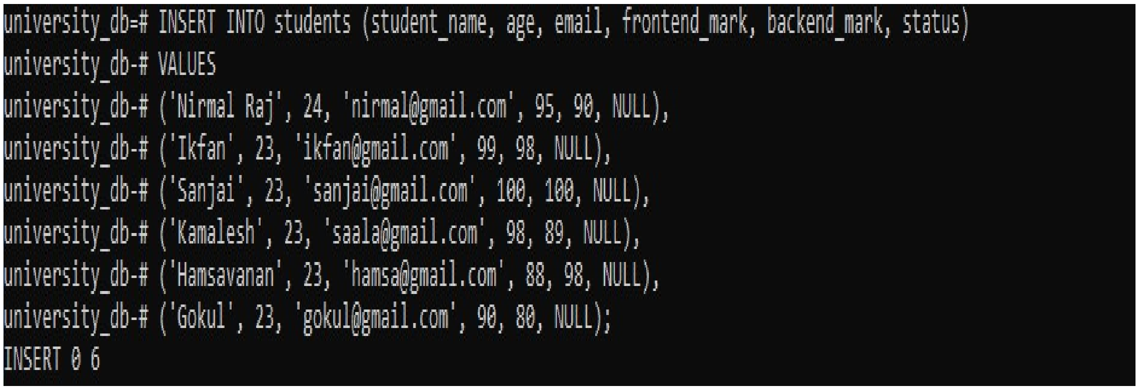
- enrollment_id (Primary Key): Integer, unique identifier for enrollments.
- student_id (Foreign Key): Integer, referencing student_id in "Students" table.
- course_id (Foreign Key): Integer, referencing course_id in "Courses" table.



```
university_db=# CREATE TABLE enrollment (
university_db(# enrollment_id SERIAL PRIMARY KEY,
university_db(# student_id INTEGER REFERENCES students(student_id),
university_db(# course_id INTEGER REFERENCES courses(course_id));
CREATE TABLE
```

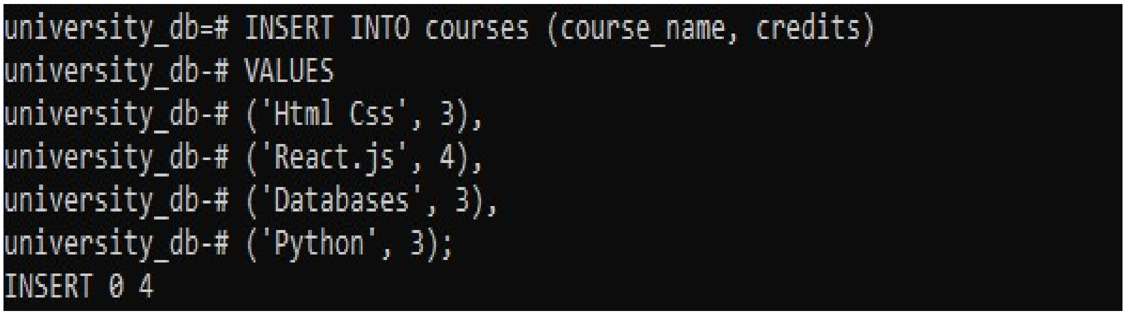
Sample Data

- Insert the following sample data into the "**students**" table:



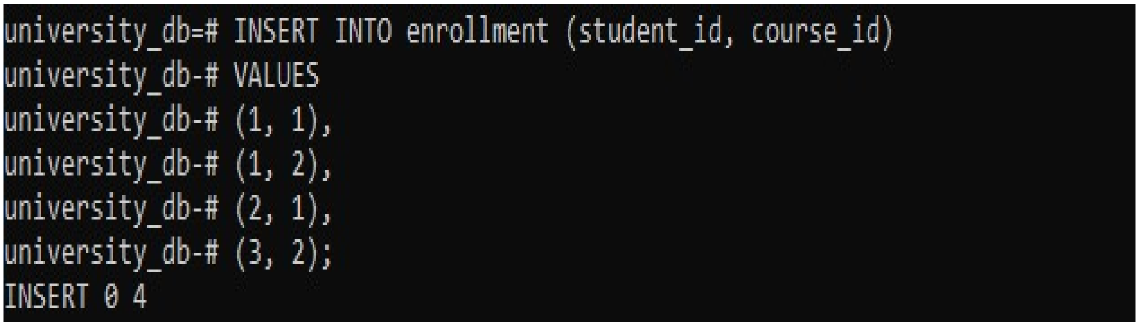
```
university_db=# INSERT INTO students (student_name, age, email, frontend_mark, backend_mark, status)
university_db-# VALUES
university_db-# ('Nirmal Raj', 24, 'nirmal@gmail.com', 95, 90, NULL),
university_db-# ('Ikfan', 23, 'ikfan@gmail.com', 99, 98, NULL),
university_db-# ('Sanjai', 23, 'sanjai@gmail.com', 100, 100, NULL),
university_db-# ('Kamalesh', 23, 'saala@gmail.com', 98, 89, NULL),
university_db-# ('Hamsavanan', 23, 'hamsa@gmail.com', 88, 98, NULL),
university_db-# ('Gokul', 23, 'gokul@gmail.com', 90, 80, NULL);
INSERT 0 6
```

- Insert the following sample data into the "**courses**" table:



```
university_db=# INSERT INTO courses (course_name, credits)
university_db-# VALUES
university_db-# ('Html Css', 3),
university_db-# ('React.js', 4),
university_db-# ('Databases', 3),
university_db-# ('Python', 3);
INSERT 0 4
```

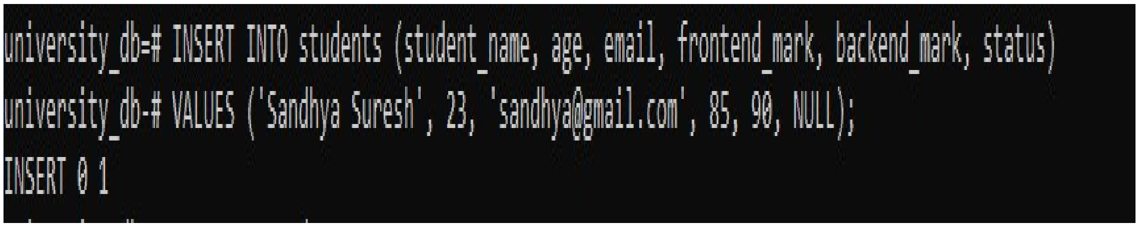
- Insert the following sample data into the "**enrollment**" table:



```
university_db=# INSERT INTO enrollment (student_id, course_id)
university_db-# VALUES
university_db-# (1, 1),
university_db-# (1, 2),
university_db-# (2, 1),
university_db-# (3, 2);
INSERT 0 4
```

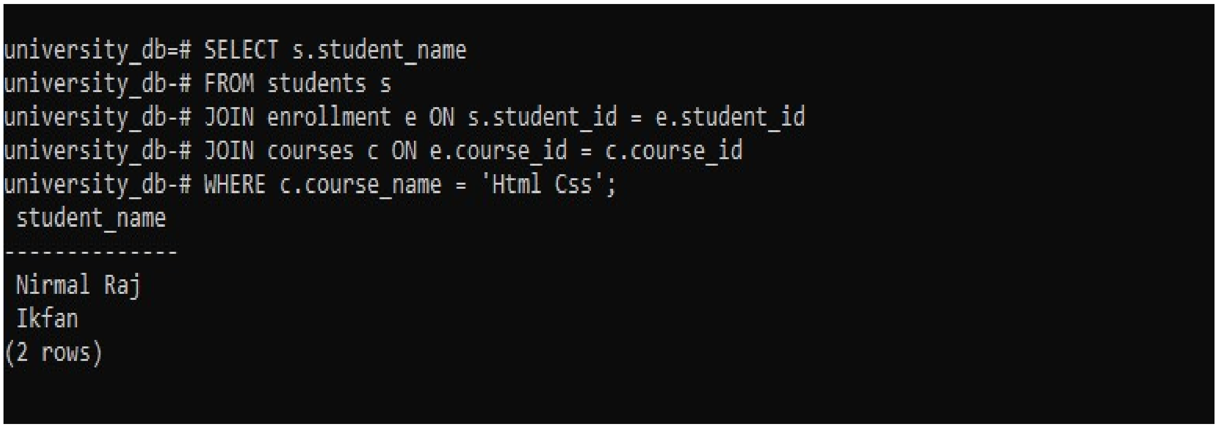
Execute SQL queries to fulfill the ensuing tasks:
Query 1:
Insert a new student record with the following details:

- Name: YourName
- Age: YourAge
- Email: YourEmail
- Frontend-Mark: YourMark
- Backend-Mark: YourMark
- Status: NULL



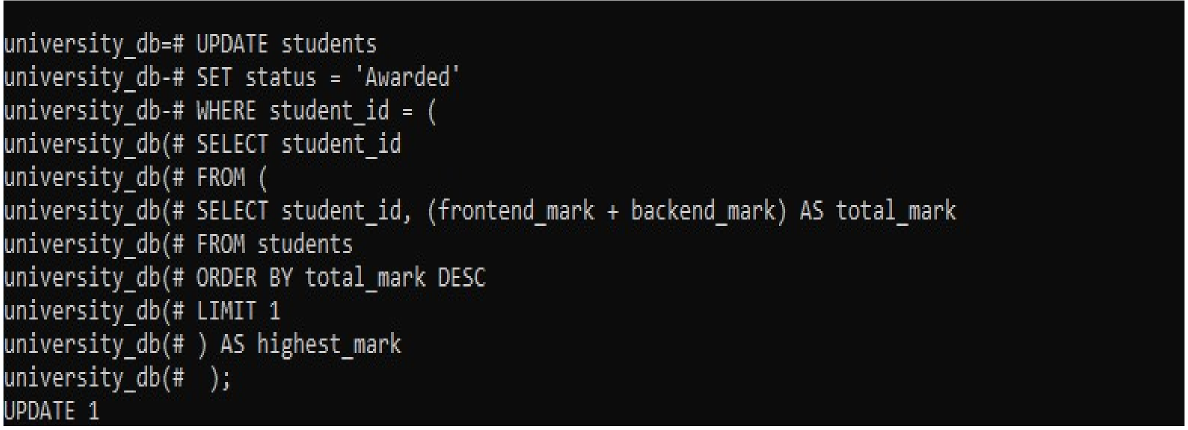
```
university_db=# INSERT INTO students (student_name, age, email, frontend_mark, backend_mark, status)
university_db-# VALUES ('Sandhya Suresh', 23, 'sandhya@gmail.com', 85, 90, NULL);
INSERT 0 1
```

Query 2:
Retrieve the names of all students who are enrolled in the course titled Html Css.js'.
Sample Output:



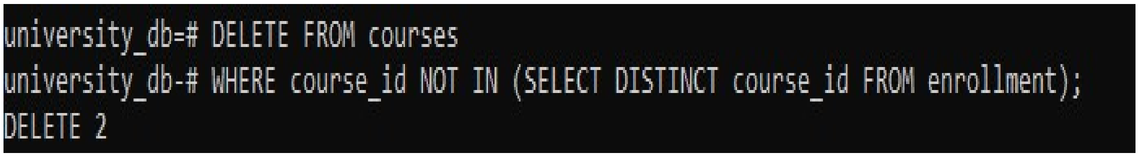
```
university_db=# SELECT s.student_name
university_db-# FROM students s
university_db-# JOIN enrollment e ON s.student_id = e.student_id
university_db-# JOIN courses c ON e.course_id = c.course_id
university_db-# WHERE c.course_name = 'Html Css';
 student_name
--------------
 Nirmal Raj
 Ikfan
(2 rows)
```

Query 3:
Update the status of the student with the highest total (frontend_mark + backend_mark) mark to 'Awarded'



```
university_db=# UPDATE students
university_db-# SET status = 'Awarded'
university_db-# WHERE student_id = (
university_db(# SELECT student_id
university_db(# FROM (
university_db(# SELECT student_id, (frontend_mark + backend_mark) AS total_mark
university_db(# FROM students
university_db(# ORDER BY total_mark DESC
university_db(# LIMIT 1
university_db(# ) AS highest_mark
university_db(#  );
UPDATE 1
```

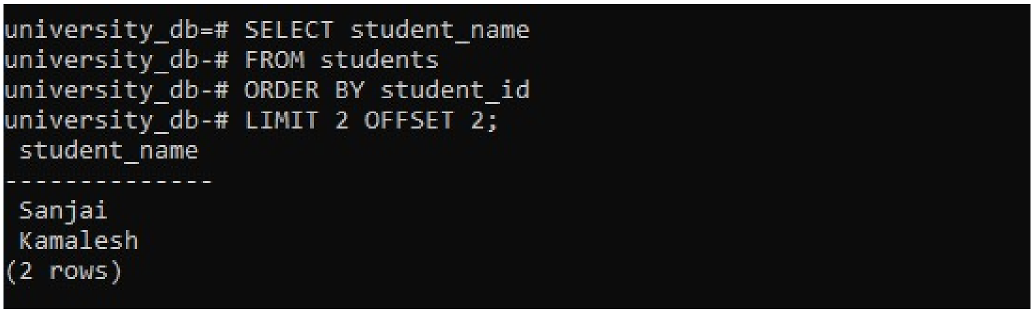
Query 4:
Delete all courses that have no students enrolled.



```
university_db=# DELETE FROM courses
university_db-# WHERE course_id NOT IN (SELECT DISTINCT course_id FROM enrollment);
DELETE 2
```

Query 5:
Retrieve the names of students using a limit of 2, starting from the 3rd student.

Sample Output:

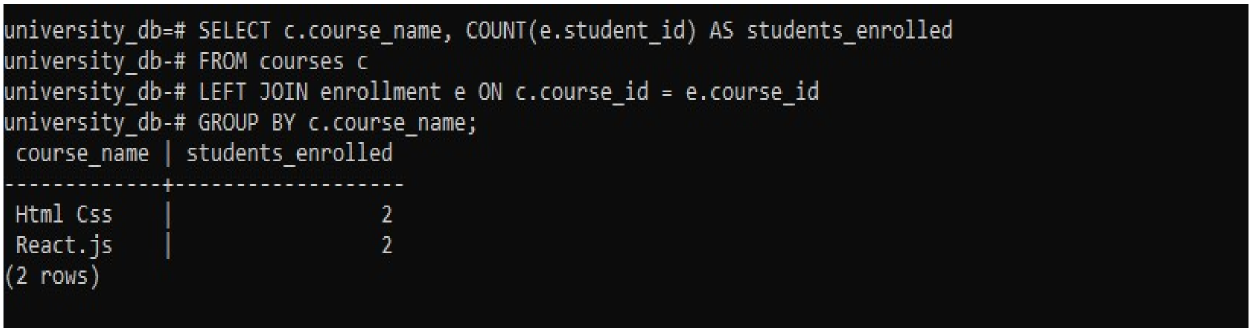


```
university_db=# SELECT student_name
university_db-# FROM students
university_db-# ORDER BY student_id
university_db-# LIMIT 2 OFFSET 2;
 student_name
--------------
 Sanjai
 Kamalesh
(2 rows)
```

Query 6:

Retrieve the course names and the number of students enrolled in each course.

Sample Output:

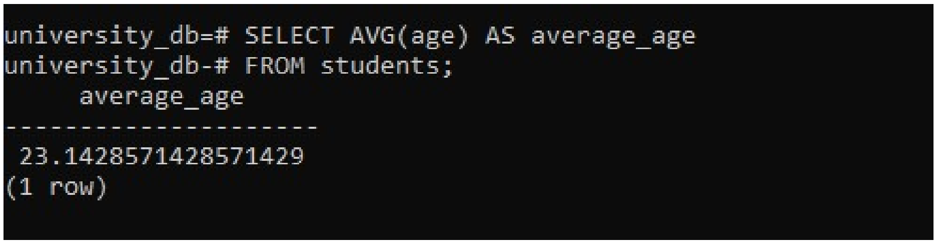


```
university_db=# SELECT c.course_name, COUNT(e.student_id) AS students_enrolled
university_db-# FROM courses c
university_db-# LEFT JOIN enrollment e ON c.course_id = e.course_id
university_db-# GROUP BY c.course_name;
 course_name | students_enrolled
-------------+-------------------
 Html Css    |                 2
 React.js    |                 2
(2 rows)
```

Query 7:

Calculate and display the average age of all students.

Sample Output:

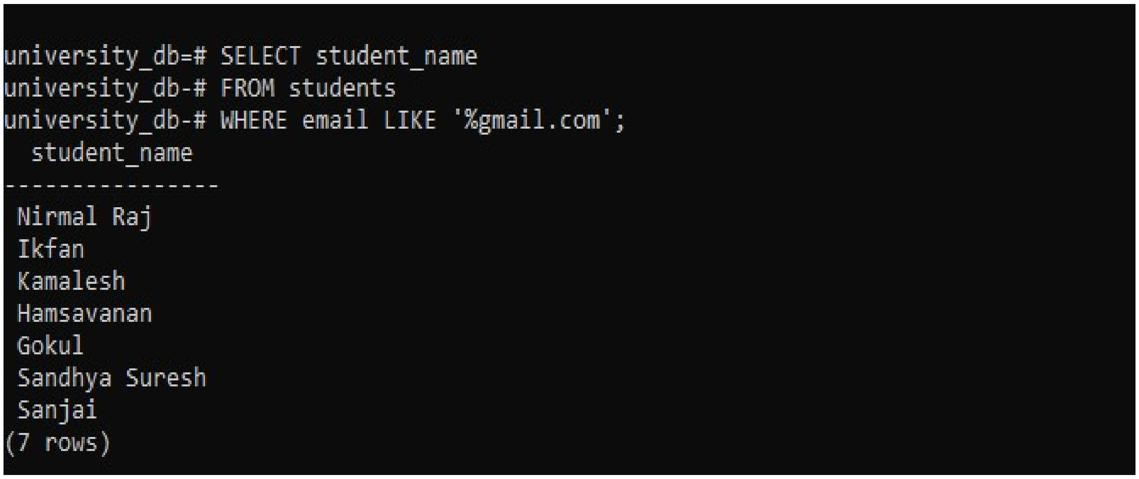


```
university_db=# SELECT AVG(age) AS average_age
university_db-# FROM students;
     average_age
---------------------
 23.1428571428571429
(1 row)
```

Query 8:

Retrieve the names of students whose email addresses contain ‘gmail.com'.

Sample Output:



```
university_db=# SELECT student_name
university_db-# FROM students
university_db-# WHERE email LIKE '%gmail.com';
  student_name
----------------
 Nirmal Raj
 Ikfan
 Kamalesh
 Hamsavanan
 Gokul
 Sandhya Suresh
 Sanjai
(7 rows)
```

Based on the above table data explain the concept along with the example for below items

1. Explain the primary key and foreign key concepts in PostgreSQL.
2. What is the difference between the VARCHAR and CHAR data types?
3. Explain the purpose of the WHERE clause in a SELECT statement.
4. What are the LIMIT and OFFSET clauses used for?
5. How can you perform data modification using UPDATE statements?
6. What is the significance of the JOIN operation, and how does it work in PostgreSQL?
7. Explain the GROUP BY clause and its role in aggregation operations.
8. How can you calculate aggregate functions like COUNT, SUM, and AVG in PostgreSQL?
9. What is the purpose of an index in PostgreSQL, and how does it optimize query performance?
10. Explain the concept of a PostgreSQL view and how it differs from a table.

1. **Primary Key and Foreign Key in PostgreSQL**:

- **Primary Key**: A primary key is a column or a set of columns that uniquely identifies each row in a table. It enforces entity integrity and ensures that each record is uniquely identifiable.

- **Foreign Key**: A foreign key is a column or a set of columns in a table that refers to the primary key of another table. It establishes a link between two tables, enforcing referential integrity and ensuring that data remains consistent between related tables.

2 **Difference between VARCHAR and CHAR data types**:

- **VARCHAR**: Variable-length character string. It stores strings of varying lengths, up to a specified maximum.
- **CHAR**: Fixed-length character string. It stores strings of a fixed length, padding shorter strings with spaces.

3 **Purpose of the WHERE clause in a SELECT statement**:

- The **WHERE** clause filters records based on a condition, allowing you to retrieve only the rows that meet specific criteria from a table.

4 **LIMIT and OFFSET clauses**:

- **LIMIT**: Specifies the maximum number of rows to return in a result set.
- **OFFSET**: Specifies the number of rows to skip before starting to return rows.

5 **Performing data modification using UPDATE statements**:

- The **UPDATE** statement in PostgreSQL modifies existing records in a table based on a specified condition, allowing you to change the values of one or more columns.

6 **Significance of the JOIN operation in PostgreSQL**:

- **JOIN** combines rows from two or more tables based on a related column between them, allowing you to retrieve data that spans across multiple tables.

7 **GROUP BY clause and its role in aggregation operations**:

- The **GROUP BY** clause groups rows that have the same values into summary rows, and it is used with aggregate functions like SUM, COUNT, AVG, etc., to perform calculations on grouped data.

8 **Calculating aggregate functions in PostgreSQL**:

- Aggregate functions like **COUNT**, **SUM**, and **AVG** calculate values across a set of rows. For example:

- **COUNT(column_name)**: Counts the number of rows that match a specified condition.
- **SUM(column_name)**: Calculates the sum of numeric values in a column.
- **AVG(column_name)**: Calculates the average value of numeric values in a column.

9 **Purpose of an index in PostgreSQL**:

- An **index** in PostgreSQL is a database object that improves the speed of data retrieval operations on a table at the cost of additional storage space. It enhances query performance by allowing the database to locate rows quickly using indexed columns.

10 **PostgreSQL view and its difference from a table**:

- A **view** in PostgreSQL is a virtual table based on the result set of a SELECT query. It does not store data physically but provides a way to present data from one or more tables or views in a structured format. Unlike a table, a view does not hold actual data but acts as a stored query that can be queried like a table.