PostgreSQL Assignment

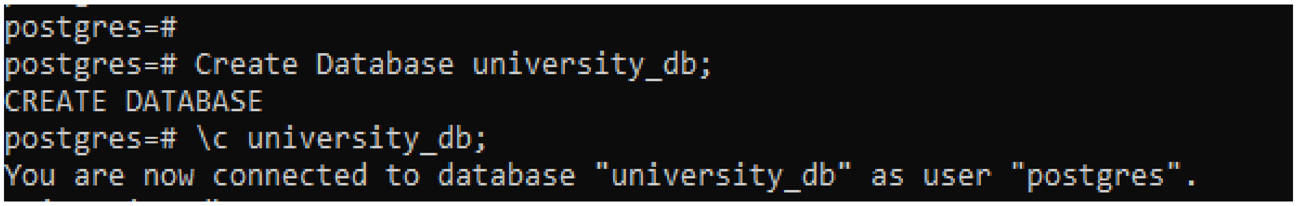
Assignment Description

In this assignment, you will work with PostgreSQL, a powerful open-source relational database management system. Your task involves creating 03 tables based on the provided sample data and then writing and executing queries to perform various database operations such as creating, reading, updating, and deleting data. Additionally, you will explore concepts like LIMIT and OFFSET, JOIN operations, GROUP BY, aggregation and LIKE.

Instructions:

Database Setup:

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

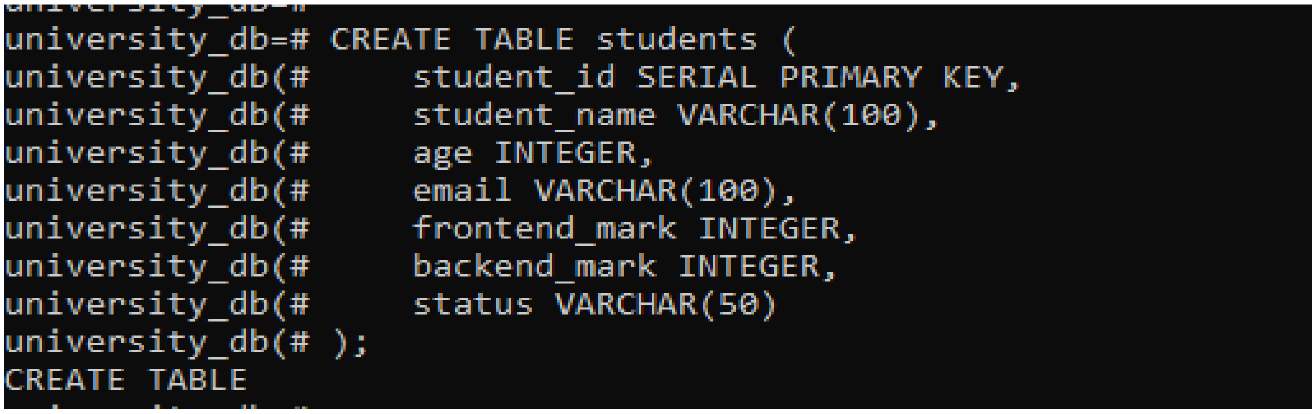


```
postgres=#
postgres=# Create Database university_db;
CREATE DATABASE
postgres=# \c university_db;
You are now connected to database "university_db" as user "postgres".
```

Table Creation:

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

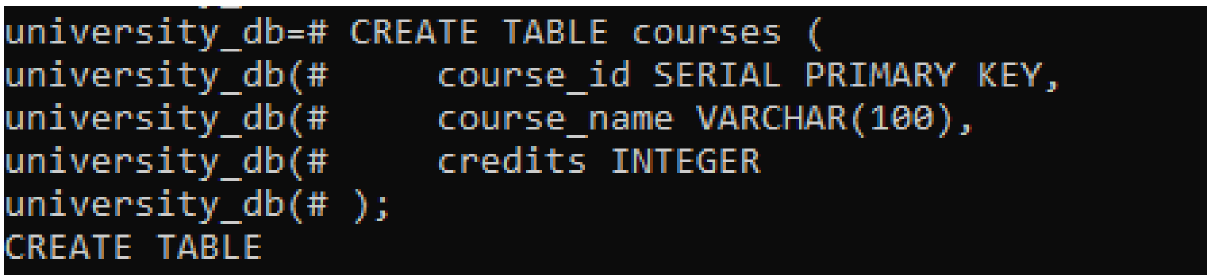
- student_id (Primary Key): Integer, unique identifier for students.
- student_name: String, representing the student's name.
- age: Integer, indicating the student's age.
- email: String, storing the student's email address.
- frontend_mark: Integer, indicating the student's frontend assignment marks.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

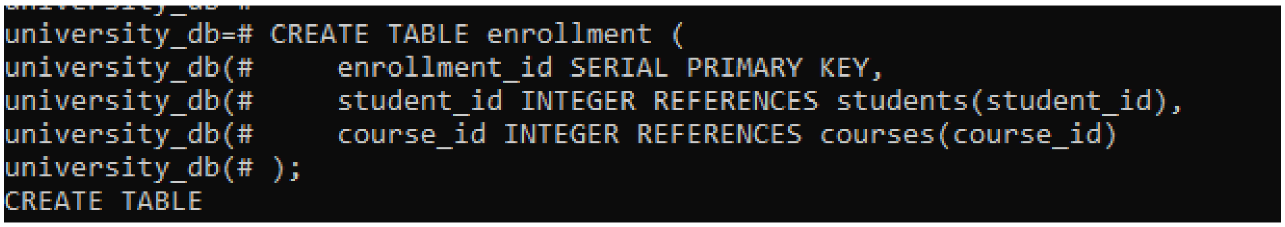
- course_id (Primary Key): Integer, unique identifier for courses.
- course_name: String, indicating the course's name.
- credits: Integer, signifying the number of credits for the course.



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

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

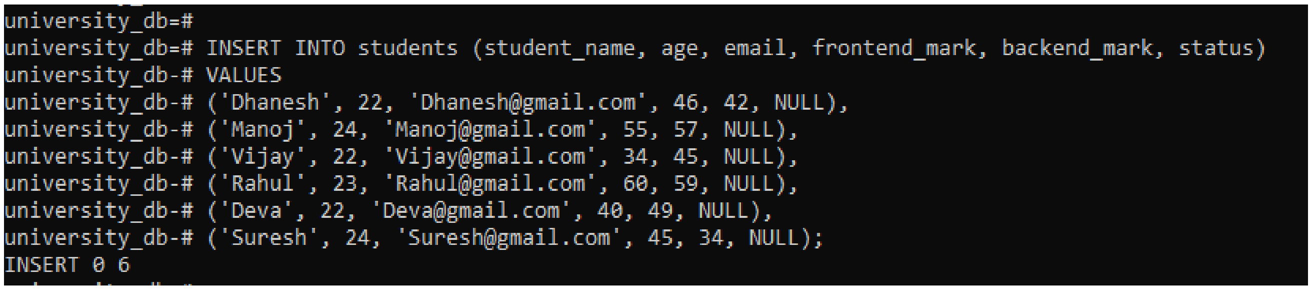
- enrollment_id (Primary Key): Integer, unique identifier for enrollments.
- student_id (Foreign Key): Integer, referencing student_id in "Students" table.
- course_id (Foreign Key): Integer, referencing course_id in "Courses" table.



```
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)
university_db(# );
CREATE TABLE
```

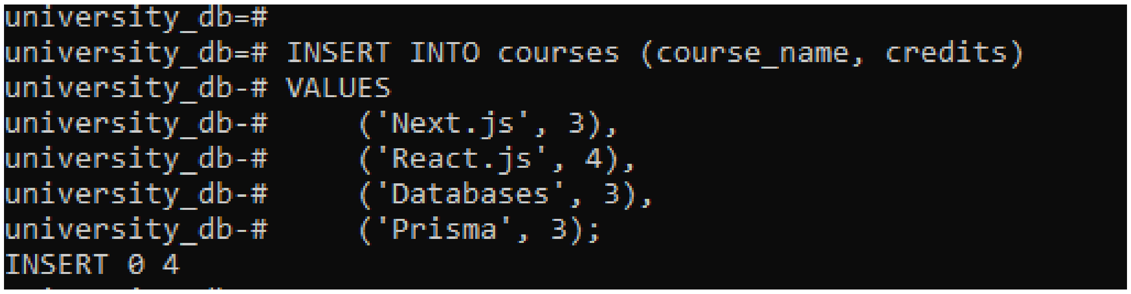
Sample Data

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



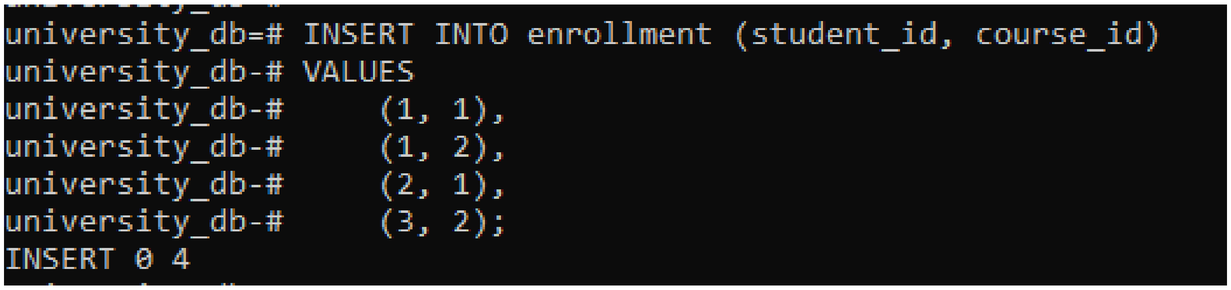
```
university_db=#
university_db=# INSERT INTO students (student_name, age, email, frontend_mark, backend_mark, status)
university_db-# VALUES
university_db-# ('Dhanesh', 22, 'Dhanesh@gmail.com', 46, 42, NULL),
university_db-# ('Manoj', 24, 'Manoj@gmail.com', 55, 57, NULL),
university_db-# ('Vijay', 22, 'Vijay@gmail.com', 34, 45, NULL),
university_db-# ('Rahul', 23, 'Rahul@gmail.com', 60, 59, NULL),
university_db-# ('Deva', 22, 'Deva@gmail.com', 40, 49, NULL),
university_db-# ('Suresh', 24, 'Suresh@gmail.com', 45, 34, NULL);
INSERT 0 6
```

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



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

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



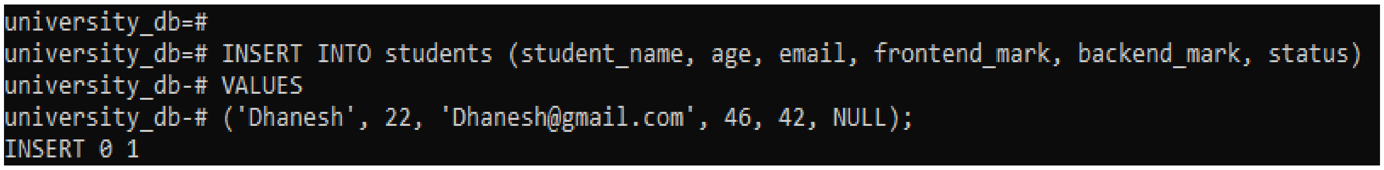
```
university_db=# INSERT INTO enrollment (student_id, course_id)
university_db-# VALUES
university_db-#     (1, 1),
university_db-#     (1, 2),
university_db-#     (2, 1),
university_db-#     (3, 2);
INSERT 0 4
```

Execute SQL queries to fulfill the ensuing tasks:

Query 1:

Insert a new student record with the following details:

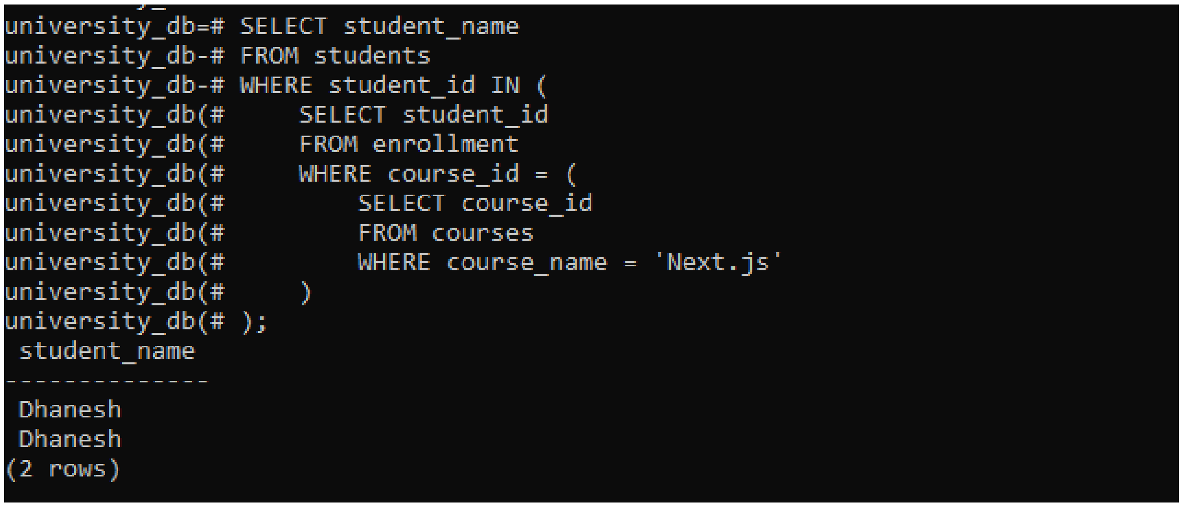
- Name: YourName
- Age: YourAge
- Email: YourEmail
- Frontend-Mark: YourMark
- Backend-Mark: YourMark
- Status: NULL



```
university_db=#
university_db=# INSERT INTO students (student_name, age, email, frontend_mark, backend_mark, status)
university_db-# VALUES
university_db-# ('Dhanesh', 22, 'Dhanesh@gmail.com', 46, 42, NULL);
INSERT 0 1
```

Query 2:

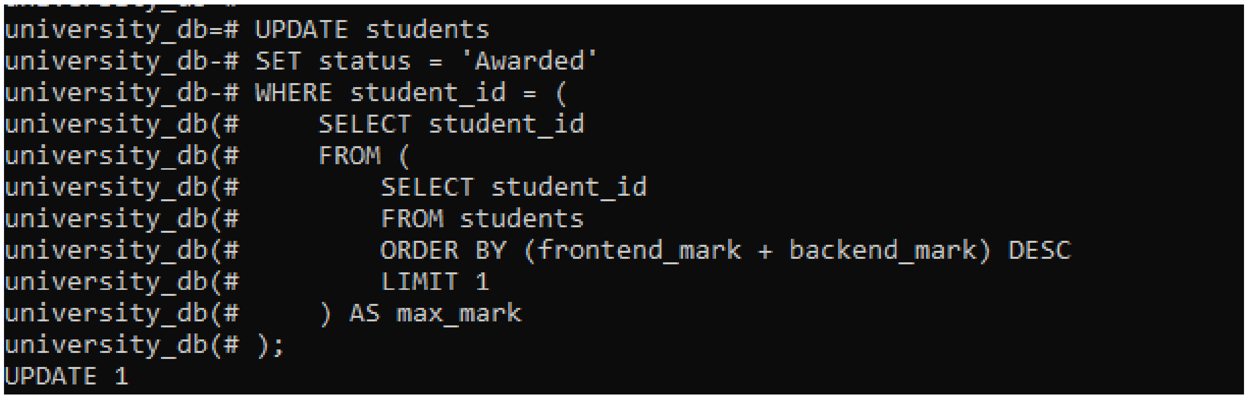
Retrieve the names of all students who are enrolled in the course titled 'Next.js'.



```
university_db=# SELECT student_name
university_db-# FROM students
university_db-# WHERE student_id IN (
university_db(#     SELECT student_id
university_db(#     FROM enrollment
university_db(#     WHERE course_id = (
university_db(#         SELECT course_id
university_db(#         FROM courses
university_db(#         WHERE course_name = 'Next.js'
university_db(#     )
university_db(# );
 student_name
--------------
 Dhanesh
 Dhanesh
(2 rows)
```

Query 3:

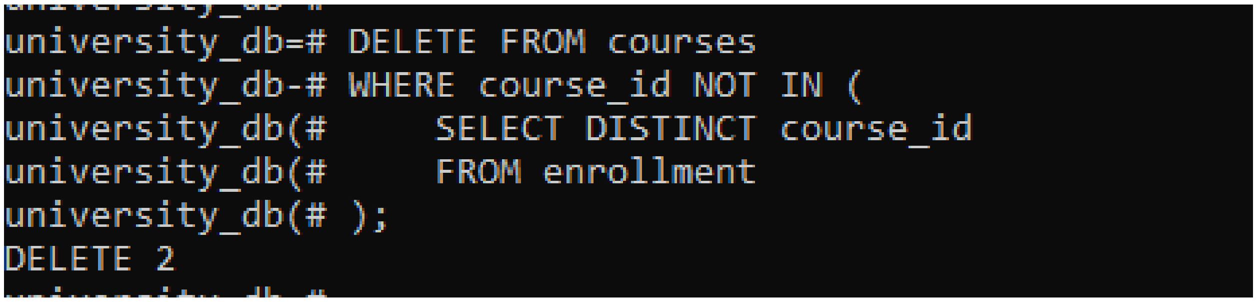
Update the status of the student with the highest total (frontend_mark + backend_mark) mark to 'Awarded



```
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
university_db(#         FROM students
university_db(#         ORDER BY (frontend_mark + backend_mark) DESC
university_db(#         LIMIT 1
university_db(#     ) AS max_mark
university_db(# );
UPDATE 1
```

Query 4:

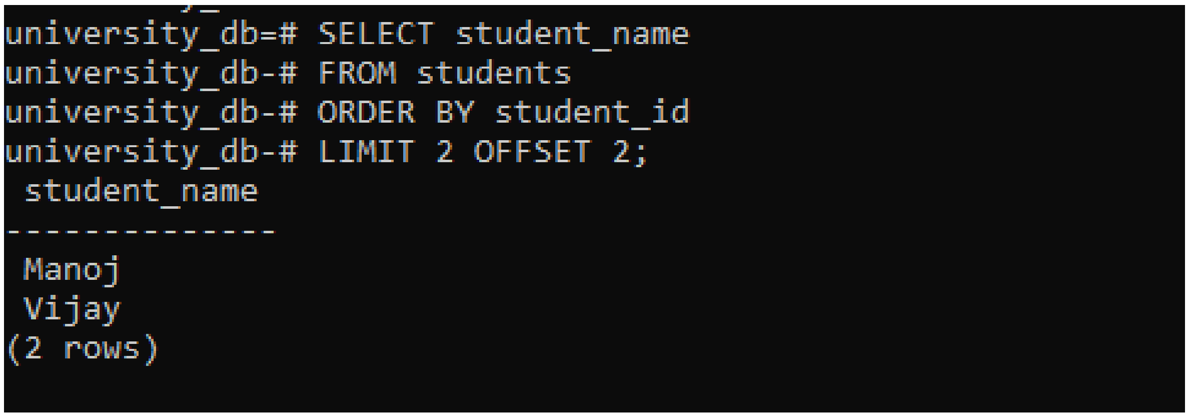
Delete all courses that have no students enrolled.



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

Query 5:

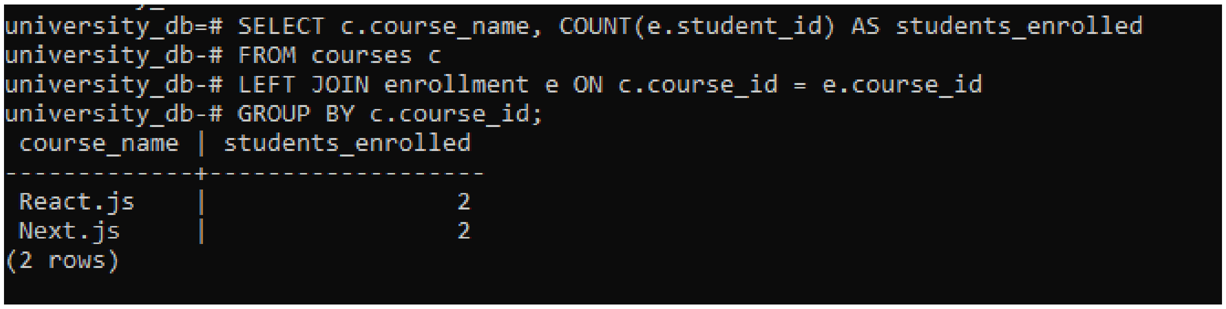
Retrieve the names of students using a limit of 2, starting from the 3rd student.



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

Query 6:

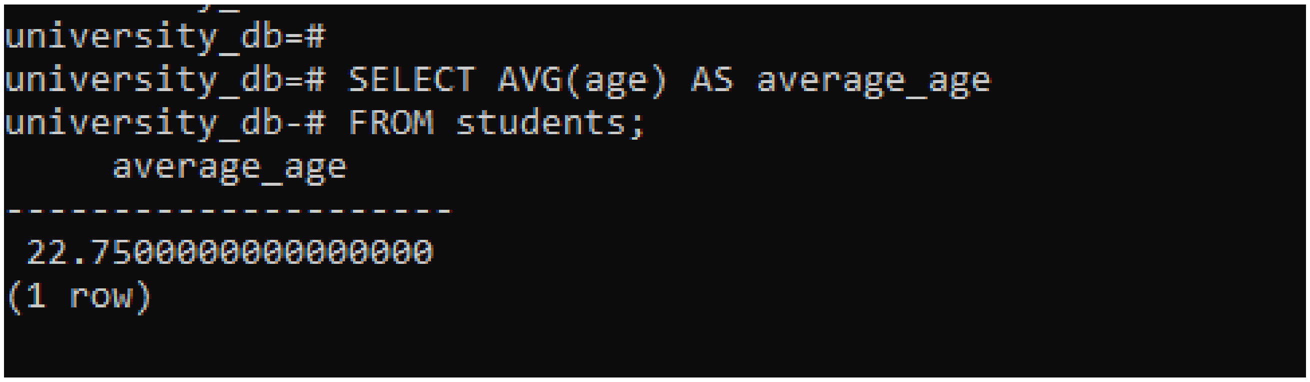
Retrieve the course names and the number of students enrolled in each course.



```
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_id;
 course_name | students_enrolled
-------------+-------------------
 React.js    |                 2
 Next.js     |                 2
(2 rows)
```

Query 7:

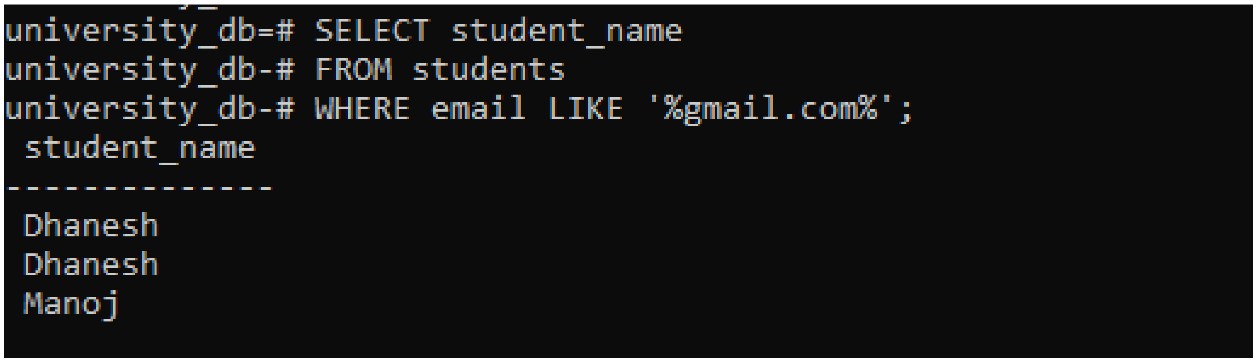
Calculate and display the average age of all students.



```
university_db=#
university_db=# SELECT AVG(age) AS average_age
university_db-# FROM students;
     average_age
---------------------
 22.7500000000000000
(1 row)
```

Query 8:

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



```
university_db=# SELECT student_name
university_db-# FROM students
university_db-# WHERE email LIKE '%gmail.com%';
 student_name
--------------
 Dhanesh
 Dhanesh
 Manoj
```

Prepare the SQL code for table creation, sample data insertion, and the seven queries in a text document or your preferred format. Include comments explaining each query's purpose and functionality. **Save your document as "PostgreSQL_Assignment.sql" or any other appropriate name.**

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.
    - **Primary Key:** A primary key uniquely identifies each record in a table. It ensures each row is unique and cannot contain NULL values. Example: ***student_id*** in ***students*** table.
    - **Foreign Key:** A foreign key establishes a link between two tables, referencing the primary key of another table. It enforces referential integrity. Example: ***student_id*** in ***enrollment*** table references ***student_id*** in ***students*** table.

2. What is the difference between the VARCHAR and CHAR data types?
    - **VARCHAR:** Variable-length character string. Stores characters of varying lengths up to a maximum specified length.

- **CHAR:** Fixed-length character string. Pads shorter strings with spaces to match the defined length. Example: ***student_name*** is VARCHAR because names vary in length.

3. Explain the purpose of the WHERE clause in a SELECT statement.
    - The ***WHERE*** clause filters rows based on a specified condition. It allows retrieval of rows that satisfy the condition. Example: ***WHERE email LIKE '%example.com'*** filters students whose email addresses contain 'example.com'.

4. What are the LIMIT and OFFSET clauses used for?
    - **LIMIT:** Specifies the maximum number of rows to return.
    - **OFFSET:** Specifies how many rows to skip before starting to return rows. Used for pagination. Example: ***LIMIT 2 OFFSET 2*** retrieves 2 rows starting from the 3rd row.

5. How can you perform data modification using UPDATE statements?
    - ***UPDATE*** statement modifies existing records in a table based on specified conditions. Example: ***UPDATE students SET status = 'Awarded' WHERE*** . . . updates the status of a student based on certain criteria.

6. What is the significance of the JOIN operation, and how does it work in PostgreSQL?
    - **JOIN:** Combines rows from two or more tables based on related columns. Allows retrieval of related data from multiple tables in a single query. Example: ***LEFT JOIN enrollment ON c.course_id = e.course_id*** retrieves courses with the count of students enrolled.

7. Explain the GROUP BY clause and its role in aggregation operations.
    - **GROUP BY:** Groups rows that share a common value into summary rows. Used with aggregate functions like `COUNT`, `SUM`, `AVG`, etc., to perform operations on groups of data. Example: ***GROUP BY c.course_name*** groups courses for counting enrolled students.

8. How can you calculate aggregate functions like COUNT, SUM, and AVG in PostgreSQL?
    - **COUNT:** Counts the number of rows returned by a query.
    - **SUM:** Calculates the sum of values in a column.
    - **AVG:** Calculates the average of values in a column. Example: ***SELECT AVG(age) AS average_age FROM students*** calculates the average age of all students.

9. What is the purpose of an index in PostgreSQL, and how does it optimize query performance?
    - **Index:** Improves query performance by reducing the number of data pages PostgreSQL needs to read. Speeds up data retrieval operations. Example: ***CREATE INDEX idx_student_email ON students(email);*** creates an index on the ***email*** column for faster email-based searches.

10. Explain the concept of a PostgreSQL view and how it differs from a table.
    - **View:** Virtual table based on the result set of a SELECT query. Stores a query definition but not data itself. Simplifies complex queries, provides data security by limiting access to columns. Example: ***CREATE VIEW view_students AS SELECT student_name, age FROM students WHERE age > 21 ;*** creates a view of students older than 21.