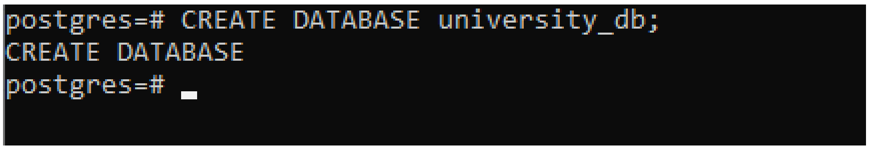
POSTGRESQL ASSIGNMENT

Database Setup:

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

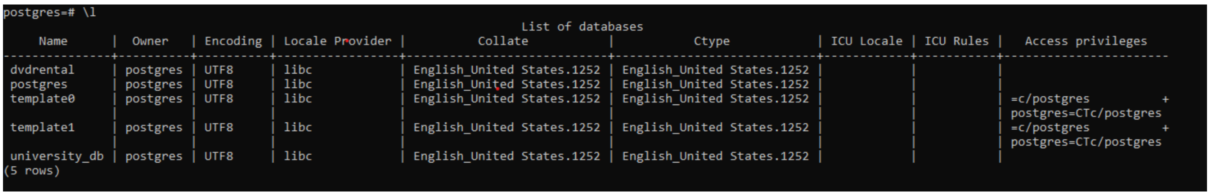
Query: CREATE DATABASE university_db;

Sample output:



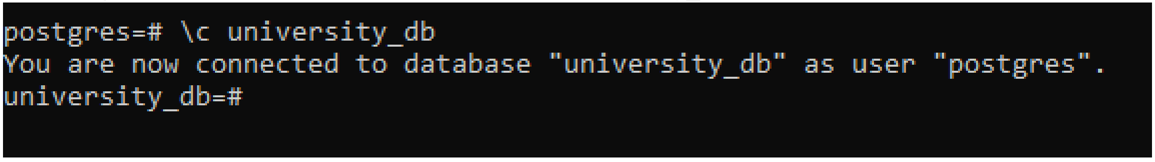
```
postgres=# CREATE DATABASE university_db;
CREATE DATABASE
postgres=# _
```

- **To List The Databases :**



```
postgres=# \l
                                                                   List of databases
     Name      |  Owner   | Encoding | Locale Provider |          Collate           |           Ctype            | ICU Locale | ICU Rules |   Access privileges
---------------+----------+----------+-----------------+----------------------------+----------------------------+------------+-----------+-----------------------
 dvdrental     | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |            |           |
 postgres      | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |            |           |
 template0     | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |            |           | =c/postgres          +
               |          |          |                 |                            |                            |            |           | postgres=CTc/postgres
 template1     | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |            |           | =c/postgres          +
               |          |          |                 |                            |                            |            |           | postgres=CTc/postgres
 university_db | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |            |           |
(5 rows)
```

- **To change and connect the database into the university_db.**



```
postgres=# \c university_db
You are now connected to database "university_db" as user "postgres".
university_db=#
```

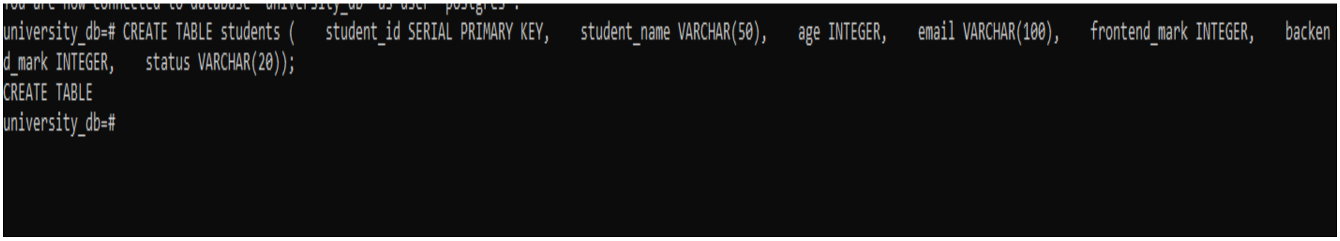
Table Creation:

2. 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.

Query: CREATE TABLE students (student_id SERIAL PRIMARY KEY, student_name VARCHAR(50),   age INTEGER,   email VARCHAR(100), frontend_mark INTEGER,   backend_mark INTEGER,   status VARCHAR(20));

Terminal Output:



```
You are now connected to database "university_db" as user "postgres".
university_db=# CREATE TABLE students (    student_id SERIAL PRIMARY KEY,    student_name VARCHAR(50),    age INTEGER,    email VARCHAR(100),    frontend_mark INTEGER,    backen
d_mark INTEGER,    status VARCHAR(20));
CREATE TABLE
university_db=#
```

3.Create a "courses" table with the following fields:

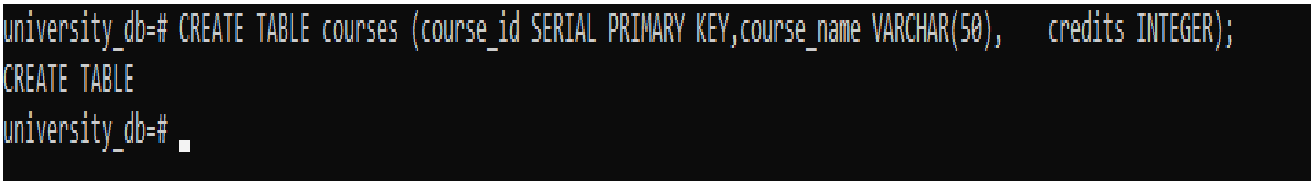
● course_id (Primary Key): Integer, unique identifier for courses.

● course_name: String, indicating the course's name.

● credits: Integer, signifying the number of credits for the course.

Query: CREATE TABLE courses (course_id SERIAL PRIMARY KEY, course_name VARCHAR(50), credits INTEGER);

TerminalOutput:



```
university_db=# CREATE TABLE courses (course_id SERIAL PRIMARY KEY,course_name VARCHAR(50),    credits INTEGER);
CREATE TABLE
university_db=#
```

4.Create an "enrollment" table with the following fields:

● enrollment_id (Primary Key): Integer, unique identifier for enrollments.

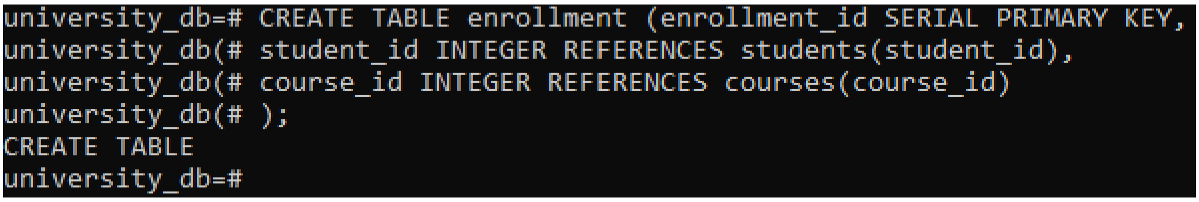
● student_id (Foreign Key): Integer, referencing student_id in "Students" table.

● course_id (Foreign Key): Integer, referencing course_id in "Courses" table.

Query: CREATE TABLE enrollment (enrollment_id SERIAL PRIMARY KEY,

student_id INTEGER REFERENCES students(student_id), course_id INTEGER REFERENCES courses(course_id));

Terminal_Output:



```
university_db=# CREATE TABLE enrollment (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
university_db=#
```

5. Insert the following sample data into the "students" table

student_id	student_name	age	email	frontend_mark	backend_mark	status
1	Alice	22	alice@example.com	55	57	NULL
2	Bob	21	bob@example.com	34	45	NULL
3	Charlie	23	charlie@example.com	60	59	NULL
4	David	20	david@example.com	40	49	NULL
5	Eve	24	newemail@example.com	45	34	NULL
6	Rahim	23	rahim@gmail.com	46	42	NULL

QUERY: *INSERT INTO students (student_name, age, email, frontend_mark, backend_mark, status) VALUES ('K.VIJAY KUMAR', 22, 'Vijaykumar@gmail.com', 100, 100, NULL),*

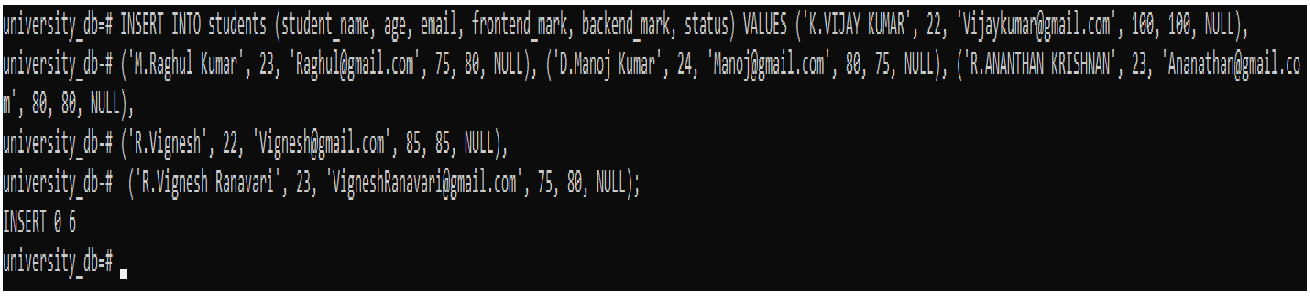
('M.Raghul Kumar', 23, 'Raghul@gmail.com', 75, 80, NULL), ('D.Manoj Kumar', 24, 'Manoj@gmail.com', 80, 75, NULL),

('R.ANANTHAN KRISHNAN', 23, 'Ananathan@gmail.com', 80, 80, NULL),

('R.Vignesh', 22, 'Vignesh@gmail.com', 85, 85, NULL),

('R.Vignesh Ranavari', 23, 'VigneshRanavari@gmail.com', 75, 80, NULL);

<u>TerminalOutput:</u>



```
university_db=# INSERT INTO students (student_name, age, email, frontend_mark, backend_mark, status) VALUES ('K.VIJAY KUMAR', 22, 'Vijaykumar@gmail.com', 100, 100, NULL),
university_db-# ('M.Raghul Kumar', 23, 'Raghul@gmail.com', 75, 80, NULL), ('D.Manoj Kumar', 24, 'Manoj@gmail.com', 80, 75, NULL), ('R.ANANTHAN KRISHNAN', 23, 'Ananathan@gmail.co
m', 80, 80, NULL),
university_db-# ('R.Vignesh', 22, 'Vignesh@gmail.com', 85, 85, NULL),
university_db-#  ('R.Vignesh Ranavari', 23, 'VigneshRanavari@gmail.com', 75, 80, NULL);
INSERT 0 6
university_db=#
```

6.Insert the following sample data into the "courses" table:

course_id	course_name	credits
1	Next.js	3
2	React.js	4
3	Databases	3
4	Prisma	3

<u>Query:</u>

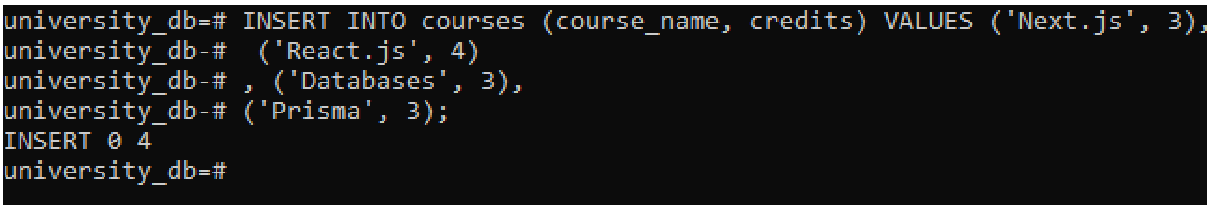
INSERT INTO courses (course_name, credits) VALUES ('Next.js', 3),

('React.js', 4)

, ('Databases', 3),

('Prisma', 3);

TerminalOutput:



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

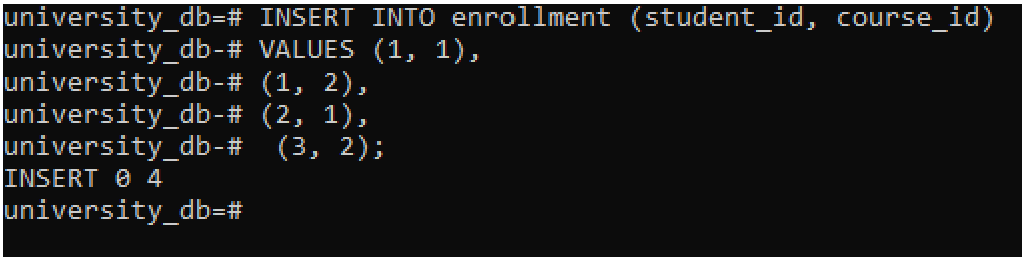
7.Insert the following sample data into the "enrollment" table:

enrollment_id	student_id	course_id
1	1	1
2	1	2
3	2	1
4	3	2

Query:

```
INSERT INTO enrollment (student_id, course_id)

VALUES (1, 1),

(1, 2),

(2, 1),

 (3, 2);
```

TerminalOutput:



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

Execute SQL queries to fulfill the ensuing tasks:

Query 1:

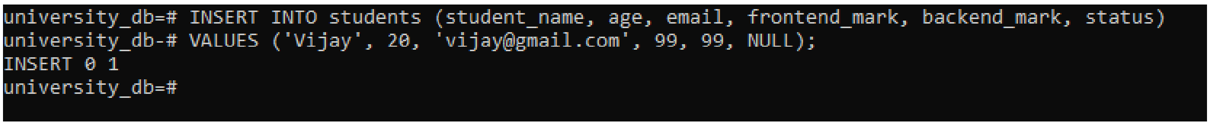
Insert a new student record with the following details:

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

Query: INSERT INTO students (student_name, age, email, frontend_mark, backend_mark, status)

VALUES ('Vijay', 20, 'vijay@gmail.com', 99, 99, NULL);

Terminal_output:



```
university_db=# INSERT INTO students (student_name, age, email, frontend_mark, backend_mark, status)
university_db-# VALUES ('Vijay', 20, 'vijay@gmail.com', 99, 99, NULL);
INSERT 0 1
university_db=#
```

Query 2:

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

Sample Output:

student_name
Alice
Bob

Query: SELECT s.student_name

FROM students s

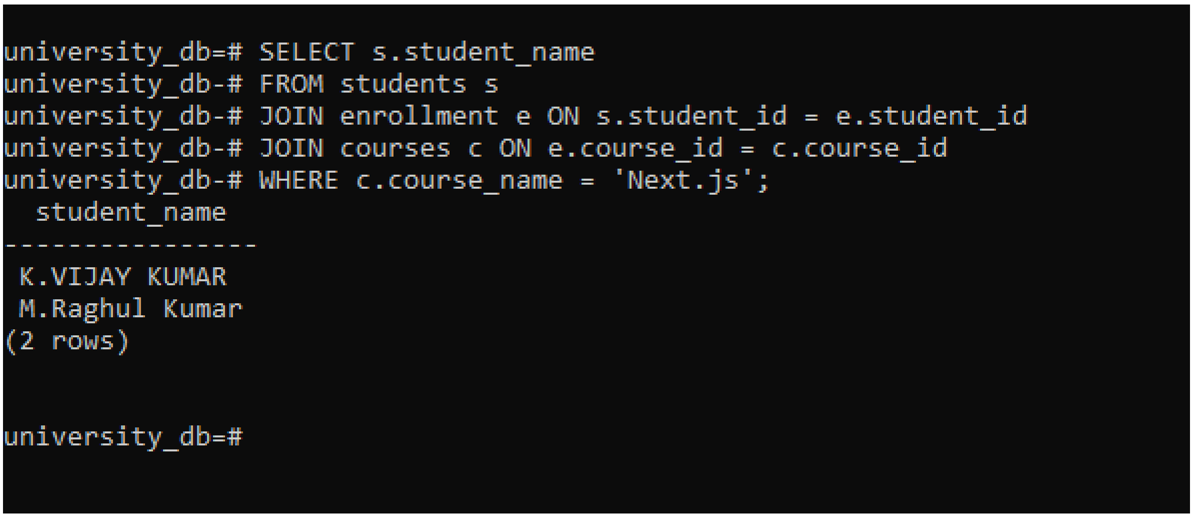
JOIN enrollment e ON s.student_id = e.student_id

JOIN courses c ON e.course_id = c.course_id

WHERE c.course_name = 'Next.js';

student_name

Terminal_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 = 'Next.js';
  student_name
----------------
 K.VIJAY KUMAR
 M.Raghul Kumar
(2 rows)


university_db=#
```

Query 3:

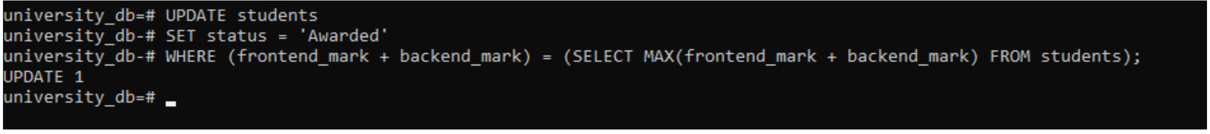
Update the status of the student with the highest total (frontend_mark + backend_mark) mark to 'Awarded'

Query: UPDATE students

SET status = 'Awarded'

WHERE (frontend_mark + backend_mark) = (SELECT MAX(frontend_mark + backend_mark) FROM students);

Terminal_output:



```
university_db=# UPDATE students
university_db-# SET status = 'Awarded'
university_db-# WHERE (frontend_mark + backend_mark) = (SELECT MAX(frontend_mark + backend_mark) FROM students);
UPDATE 1
university_db=# _
```

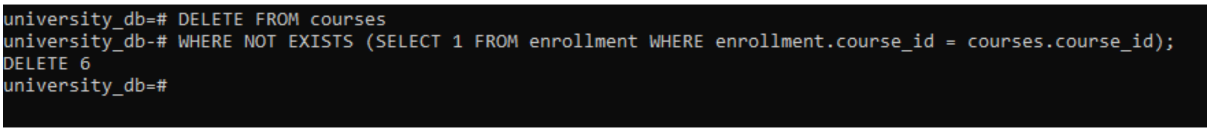
Query 4:

Delete all courses that have no students enrolled.

Query: DELETE FROM courses

WHERE NOT EXISTS (SELECT 1 FROM enrollment WHERE enrollment.course_id = courses.course_id);

Terminal_output:



```
university_db=# DELETE FROM courses
university_db-# WHERE NOT EXISTS (SELECT 1 FROM enrollment WHERE enrollment.course_id = courses.course_id);
DELETE 6
university_db=#
```

Query 5:

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

Sample Output:

Student name:

Charile

David

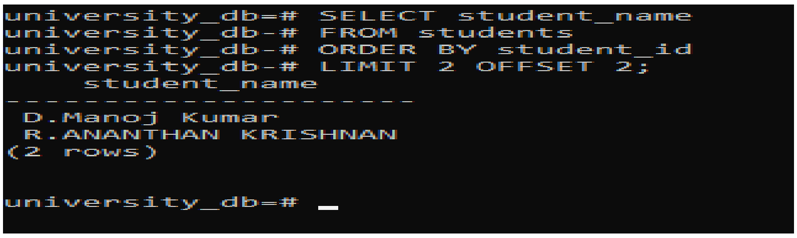
Query: SELECT student_name

FROM students

ORDER BY student_id

LIMIT 2 OFFSET 2;

Terminal_output:



```
university_db=# SELECT student_name
university_db-# FROM students
university_db-# ORDER BY student_id
university_db-# LIMIT 2 OFFSET 2;
    student_name
---------------------
 D.Manoj Kumar
 R.ANANTHAN KRISHNAN
(2 rows)

university_db=# _
```

Query 6:

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

Sample Output:

course_name	**students_enrolled**
Next.js	2
React.js	2

Query: SELECT c.course_name, COUNT(e.student_id) AS students_enrolled

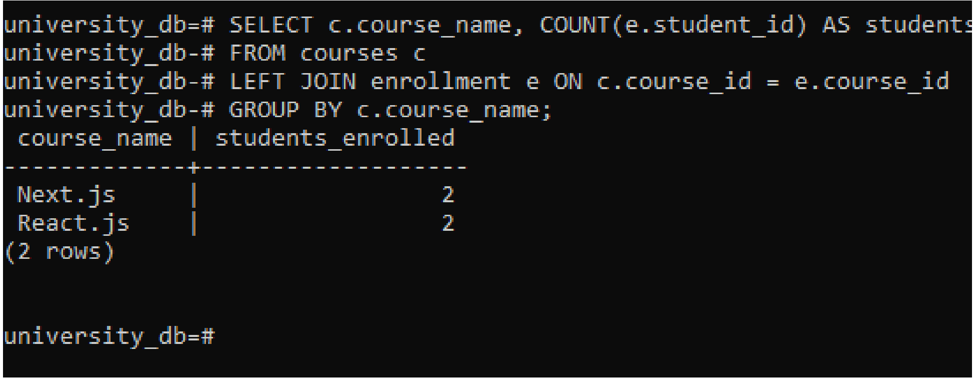
FROM courses c

LEFT JOIN enrollment e ON c.course_id = e.course_id

GROUP BY c.course_name;

course_name	students_enrolled

Terminal_output:



```
university_db=# SELECT c.course_name, COUNT(e.student_id) AS students
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
-------------+-------------------
 Next.js     |                 2
 React.js    |                 2
(2 rows)


university_db=#
```

Query 7:

Calculate and display the average age of all students.

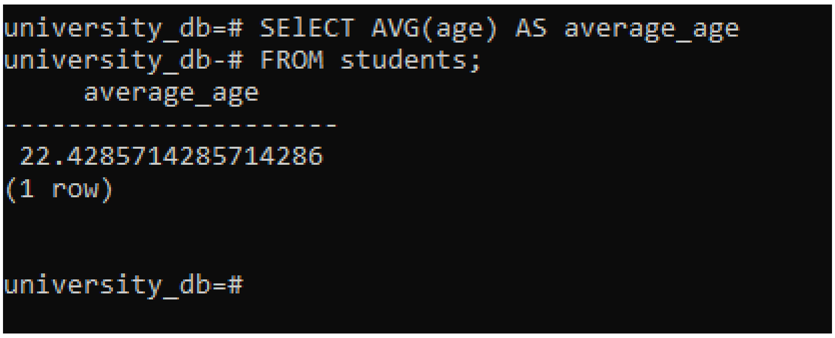
Sample Output:

average_age
22.2857142857142857

Query: SELECT AVG(age) AS average_age

FROM students;

Terminal_output:



```
university_db=# SElECT AVG(age) AS average_age
university_db-# FROM students;
     average_age
---------------------
 22.4285714285714286
(1 row)


university_db=#
```

Query 8:

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

Sample Output:

student_name

Alice

Bob

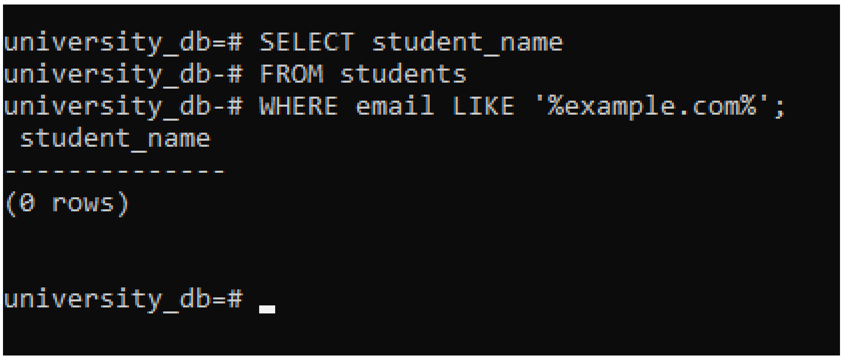
Charlie

`David

Query:

```
SELECT student_name
FROM students
WHERE email LIKE '%example.com%';
```

Terminal_output:



```
university_db=# SELECT student_name
university_db-# FROM students
university_db-# WHERE email LIKE '%example.com%';
 student_name
--------------
(0 rows)


university_db=# 
```

Explanation of Concepts

1. **Primary Key and Foreign Key in PostgreSQL**:
   - **Primary Key**: A primary key uniquely identifies each record in a table. It ensures data integrity and serves as a reference point for relationships.
   - **Foreign Key**: A foreign key establishes a relationship between tables. It references the primary key of another table to enforce referential integrity and maintain data consistency.
2. **VARCHAR vs CHAR Data Types**:
   - **VARCHAR**: Variable-length character string. It can hold varying lengths of characters up to a specified maximum.
   - **CHAR**: Fixed-length character string. It stores exactly the number of characters specified, padding with spaces if necessary.
3. **Purpose of WHERE Clause in SELECT Statement**:
   - The `WHERE` clause filters records based on specified conditions in a `SELECT` statement. It allows retrieval of specific rows that meet the given criteria.
4. **LIMIT and OFFSET Clauses**:
   - **LIMIT**: Specifies the maximum number of rows returned by a query.
   - **OFFSET**: Specifies the number of rows to skip before starting to return rows.
5. **Data Modification Using UPDATE Statements**:
   - `UPDATE` statements modify existing records in a table. They allow changes to specific columns' values based on specified conditions.
6. **Significance of JOIN Operation in PostgreSQL**:
   - `JOIN` combines rows from two or more tables based on a related column between them. It enables retrieval of related data across tables in a single query.
7. **GROUP BY Clause and Aggregation Operations**:
   - The `GROUP BY` clause groups rows that have the same values into summary rows. It is used in conjunction with aggregate functions like `COUNT`, `SUM`, `AVG` to perform calculations on grouped data.
8. **Aggregate Functions (COUNT, SUM, AVG) in PostgreSQL**:
   - Aggregate functions operate on a set of values and return a single value. Examples include `COUNT` (counts rows), `SUM` (sums values), `AVG` (calculates average).
9. **Purpose of Index in PostgreSQL**:
   - An index in PostgreSQL improves query performance by allowing faster retrieval of rows from a table. It is created on columns to speed up data retrieval operations.
10. **PostgreSQL View vs Table**:
    - A view is a virtual table based on the result set of a SELECT query. It does not store data physically but provides a convenient way to access and manipulate complex queries. A table, on the other hand, stores data physically in the database.