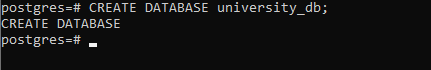
**POSTGRESQL ASSIGNMENT**

**Database Setup:**

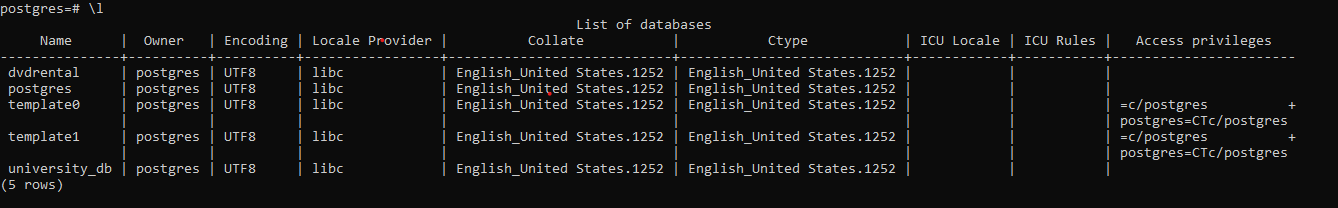
**1.Create a fresh database titled "university\_db" or any other appropriate name.**

**Query:** CREATE DATABASE university\_db;

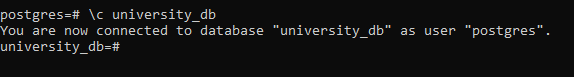
**Sample output:**

****

* **To List The Databases :**

****

* **To change and connect the database into the university\_db.**

****

**Table Creation**:

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

● student\_id (Primary Key): Integer, unique identifier for students.

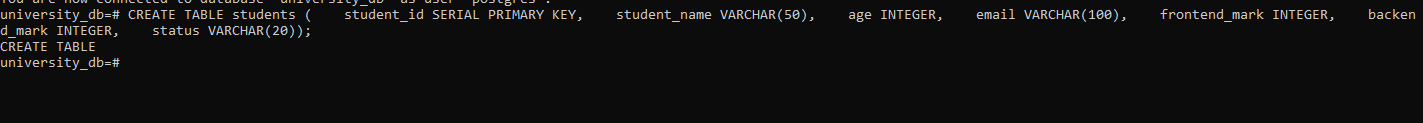
● student\_name: String, representing the student's name.

● 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:**

**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:**

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

● enrollment\_id (Primary Key): Integer, unique identifier for enrollments.

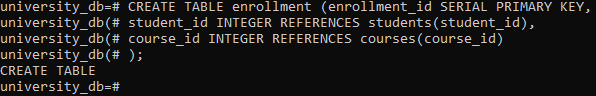
● student\_id (Foreign Key): Integer, referencing student\_id in "Students" table.

● 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:**

****

**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](mailto:alice@example.com) | 55 | 57 | NULL |
| 2 | Bob | 21 | [bob@example.com](mailto:bob@example.com) | 34 | 45 | NULL |
| 3 | Charlie | 23 | [charlie@example.com](mailto:charlie@example.com) | 60 | 59 | NULL |
| 4 | David | 20 | [david@example.com](mailto:david@example.com) | 40 | 49 | NULL |
| 5 | Eve | 24 | [newemail@example.com](mailto:newemail@example.com) | 45 | 34 | NULL |
| 6 | Rahim | 23 | [rahim@gmail.com](mailto:rahim@gmail.com) | 46 | 42 | NULL |

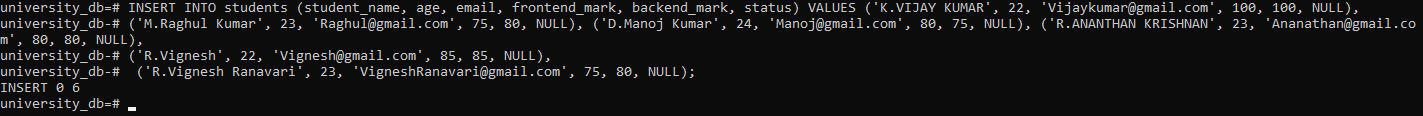
**QUERY:** *INSERT INTO students (student\_name, age, email, frontend\_mark, backend\_mark, status) VALUES ('K.VIJAY KUMAR', 22, 'Vijaykumar@gmail.com', 100, 100, NULL),*

*('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);*

**TerminalOutput:**

****

**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 |
|  |  |  |
| **Query:** |  |  |

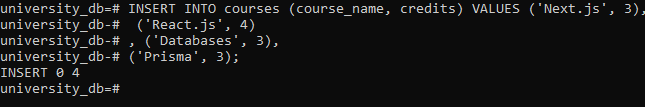
INSERT INTO courses (course\_name, credits) VALUES ('Next.js', 3),

('React.js', 4)

, ('Databases', 3),

('Prisma', 3);

**TerminalOutput:**

****

**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)

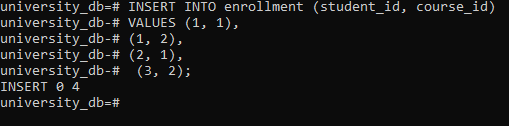
VALUES (1, 1),

(1, 2),

(2, 1),

(3, 2);

**TerminalOutput:**

****

Execute SQL queries to fulfill the ensuing tasks:

Query 1:

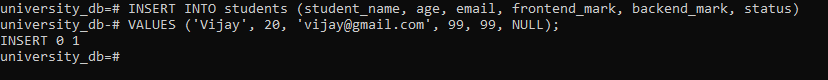
Insert a new student record with the following details:

* 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:**



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';  **Terminal\_output:** |

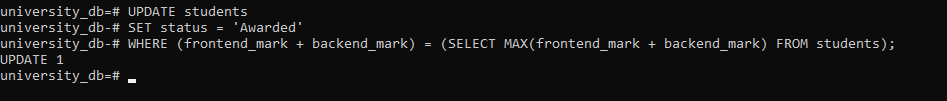
Query 3:

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:**

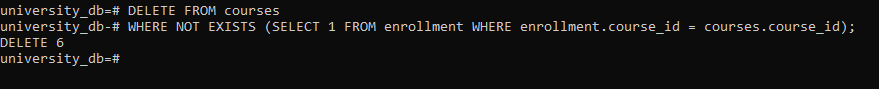
Query 4:

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:**



Query 5:

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

**Sample Output:**

**Student name:**

Charile

David

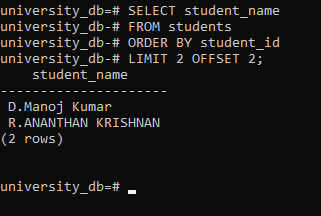
**Query:** SELECT student\_name

FROM students

ORDER BY student\_id

LIMIT 2 OFFSET 2;

**Terminal\_output:**



|  |
| --- |
|  |

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;  **Terminal\_output:** |  |

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:** |

Query 8:

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

**Sample Output:**

| **student\_name** |
| --- |
| Alice |
| Bob |
| Charlie |
| `David |

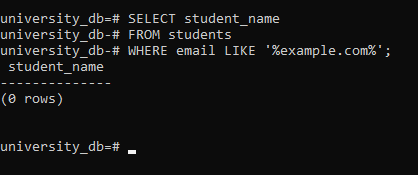
**Query:**

SELECT student\_name

FROM students

WHERE email LIKE '%example.com%';

**Terminal\_output:**

****

### 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.