POSTGRESQL ASSIGNMENT

Created the database university_db

```
CREATE DATABASE university_db;


-- Connect to the university_db database
\c university_db;

-- Created the students table
CREATE TABLE students (
   student_id SERIAL PRIMARY KEY,
   student_name VARCHAR(100),
   age INTEGER,
   email VARCHAR(100),
   frontend_mark INTEGER,
   backend_mark INTEGER,
   status VARCHAR(50)
);

-- Created the courses table
CREATE TABLE courses (
   course_id SERIAL PRIMARY KEY,
   course_name VARCHAR(100),
   credits INTEGER
);

-- Created the enrollment table
CREATE TABLE enrollment (
   enrollment_id SERIAL PRIMARY KEY,
   student_id INTEGER REFERENCES students(student_id),
   course_id INTEGER REFERENCES courses(course_id)
);



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

INSERT INTO students (student_id, student_name, age,
 email, frontend_mark, backend_mark, status)
VALUES
   (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);
```

```
==>university_db=# select*from students;
 student_id | student_name | age |        email         | frontend_mark | backend_mark | status
------------+--------------+-----+----------------------+---------------+--------------+--------
          1 | Alice        |  22 | alice@example.com    |            55 |           57 |
          2 | Bob          |  21 | bob@example.com      |            34 |           45 |
          3 | Charlie      |  23 | charlie@example.com  |            60 |           59 |
          4 | David        |  20 | david@example.com    |            40 |           49 |
          5 | Eve          |  24 | newemail@example.com |            45 |           34 |
          6 | Rahim        |  23 | rahim@gmail.com      |            46 |           42 |
(6 rows)
```

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

```
INSERT INTO courses (course_id, course_name, credits)
VALUES
   (1, 'Next.js', 3),
   (2, 'React.js', 4),
   (3, 'Databases', 3),
   (4, 'Prisma', 3);
```

```
==>university_db=# select * from courses;
 course_id | course_name | credits
-----------+-------------+---------
         1 | Next.js     |       3
         2 | React.js    |       4
         3 | Databases   |       3
         4 | Prisma      |       3
```

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

```
INSERT INTO enrollment (enrollment_id, student_id, course_id)
VALUES
   (1, 1, 1),
   (2, 1, 2),
   (3, 2, 1),
   (4, 3, 2);
```

```
==>university_db=# select * from enrollment;
 enrollment_id | student_id | course_id
```

```
---------------+------------+-----------
            1 |          1 |         1
            2 |          1 |         2
            3 |          2 |         1
            4 |          3 |         2
```

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

```
==>INSERT INTO students (student_id,student_name, age, email,
 frontend_mark, backend_mark, status)
VALUES (7,'Eswaran Arumugam', 21,
 'eswaran.codepro@gmail.com', 90, 95, NULL);

 university_db=# INSERT INTO students (student_id,student_name, age, email,
university_db(#  frontend_mark, backend_mark, status)
university_db-# VALUES (7,'Eswaran Arumugam', 21,
university_db(#  'eswaran.codepro@gmail.com', 90, 95, NULL);
INSERT 0 1
university_db=# select * from students;
 student_id |   student_name   | age |           email           | frontend_mark | backend_mark
| status
------------+------------------+-----+---------------------------+---------------+--------------+--------
          1 | Alice            |  22 | alice@example.com         |            55 |           57 |
          2 | Bob              |  21 | bob@example.com           |            34 |           45 |
          3 | Charlie          |  23 | charlie@example.com       |            60 |           59 |
          4 | David            |  20 | david@example.com         |            40 |           49 |
          5 | Eve              |  24 | newemail@example.com      |            45 |           34 |
          6 | Rahim            |  23 | rahim@gmail.com           |            46 |           42 |
          7 | Eswaran Arumugam |  21 | eswaran.codepro@gmail.com |            90 |           95
|
(7 rows)
```

Query 2:

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

```
==>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';
```

```
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
--------------
 Alice
 Bob
(2 rows)
```

Query 3:

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

```
==> UPDATE students
SET status = 'Awarded'
WHERE student_id = (
   SELECT student_id
   FROM (SELECT student_id, frontend_mark + backend_mark AS total_marks
      FROM students)
                  AS subquery
   ORDER BY total_marks DESC
   LIMIT 1
);
```

```
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_marks
university_db(#         FROM students
university_db(#     ) AS subquery
university_db(#     ORDER BY total_marks DESC
university_db(#     LIMIT 1
university_db(# );
UPDATE 1
university_db=# select * from students;
 student_id |   student_name   | age |              email              | frontend_mark | backend_mark
| status
------------+------------------+-----+--------------------------------+---------------+--------------+---------
          1 | Alice            |  22 | alice@example.com              |            55 |           57 |
          2 | Bob              |  21 | bob@example.com                |            34 |           45 |
          3 | Charlie          |  23 | charlie@example.com            |            60 |           59 |
          4 | David            |  20 | david@example.com              |            40 |           49 |
          5 | Eve              |  24 | newemail@example.com           |            45 |           34 |
          6 | Rahim            |  23 | rahim@gmail.com                |            46 |           42 |
          7 | Eswaran Arumugam |  21 | eswaran.codepro@gmail.com      |            90 |           95
| Awarded
```

Query 4:

Delete all courses that have no students enrolled.

```
==>DELETE FROM courses
    WHERE NOT EXISTS (
    SELECT 1
    FROM enrollment e
    WHERE e.course_id = courses.course_id
);
```

```
university_db=# DELETE FROM courses
university_db-#     WHERE NOT EXISTS (
university_db(#     SELECT 1
university_db(#     FROM enrollment e
university_db(#     WHERE e.course_id = courses.course_id
university_db(# );
DELETE 2
university_db=#
university_db=# select * from courses;
 course_id | course_name | credits
-----------+-------------+---------
         1 | Next.js     |       3
         2 | React.js    |       4
(2 rows)
```

Query 5:

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

```
==>SELECT student_name
FROM students
ORDER BY student_id
OFFSET 2
LIMIT 2;
```

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

Query 6:

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

```
==>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
ORDER BY c.course_name;
```

```
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
university_db-# ORDER BY c.course_name;
 course_name | students_enrolled
-------------+-------------------
 Next.js     |                 2
 React.js    |                 2
(2 rows)
```

Query 7:

Calculate and display the average age of all students.

```
==>SELECT AVG(age) AS average_age
FROM students;
```

```
university_db=# SELECT AVG(age) AS average_age
university_db-# FROM students;
     average_age
---------------------
 22.0000000000000000
(1 row)
```

Query 8:

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

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

```
university_db=# SELECT student_name
university_db-# FROM students
university_db-# WHERE email LIKE '%example.com';
   student_name
------------------
 Alice
 Bob
 Charlie
 David
 Eve
 (5 rows)
```

QUESTION AND ANSWER

1.Explain the primary key and foreign key concepts in PostgreSQL.

Primary Key:

- **Purpose**: A primary key uniquely identifies each record in a table.
- **Characteristics**: It must be unique and not null. Typically, it's indexed for fast access.
- **Example**: `student_id` in a `students` table.
- Example:university_db=# select student_id from students;

student_id

- -----------
-           1
-           3
-           4
-           5
-           6
-           7
-           2
- (7 rows)

Foreign Key:

- **Purpose**: Establishes a relationship between tables by referencing the primary key of another table.
- **Usage**: Ensures referential integrity and defines relationships between tables.
- **Example**: `student_id` in an `enrollment` table referencing `student_id` in `students`

—---

2.What is the difference between the VARCHAR and CHAR data types?

VARCHAR:

- **Variable-length**: Stores strings of varying lengths up to a specified maximum.
- **Example**: `VARCHAR(100)` can store up to 100 characters.

CHAR:

- **Fixed-length**: Stores strings of a fixed length.
- **Padding**: If the string is shorter than the specified length, it pads spaces.
- **Example**: `CHAR(10)` stores exactly 10 characters.

—---

3.Explain the purpose of the WHERE clause in a SELECT statement.

Purpose: Filters rows based on a condition specified after the WHERE keyword.
Usage: Allows retrieval of specific rows that meet certain criteria.
Example:

```
university_db=# SELECT * FROM students WHERE age > 25;
 student_id | student_name | age |      email      | frontend_mark | backend_mark | status
------------+--------------+-----+-----------------+---------------+--------------+--------
          2 | Bob          |  30 | bob@example.com |            34 |           45 |
(1 row)
```

——---

4.What are the LIMIT and OFFSET clauses used for?

LIMIT: Specifies the maximum number of rows to return in the result set.
OFFSET: Specifies the number of rows to skip before starting to return rows from the result set.
Usage: Enables pagination or fetching a subset of rows.
Example:

```
university_db=#   SELECT * FROM students LIMIT 3 OFFSET 2;
 student_id | student_name | age |        email         | frontend_mark | backend_mark | status
------------+--------------+-----+----------------------+---------------+--------------+--------
          4 | David        |  20 | david@example.com    |            40 |           49 |
          5 | Eve          |  24 | newemail@example.com |            45 |           34 |
          6 | Rahim        |  23 | rahim@gmail.com      |            46 |           42 |
(3 rows)
```

——---

5.How can you perform data modification using UPDATE statements?
Purpose: Updates existing records in a table based on specified conditions.
Syntax: `UPDATE table_name SET column1 = value1, column2 = value2 WHERE condition;`

```
Example:
university_db=# UPDATE students SET age = 30 WHERE student_id = 2;
UPDATE 1
university_db=# select*from students;
 student_id |   student_name    | age |              email
| frontend_mark | backend_mark | status
------------+------------------+-----+----------------------------------
---+---------------+--------------+---------
```

```
         1 | Alice            |  22 | alice@example.com
|           55 |           57 |
         3 | Charlie          |  23 | charlie@example.com
|           60 |           59 |
         4 | David            |  20 | david@example.com
|           40 |           49 |
         5 | Eve              |  24 | newemail@example.com
|           45 |           34 |
         6 | Rahim            |  23 | rahim@gmail.com
|           46 |           42 |
         7 | Eswaran Arumugam |  21 |
eswaran.codepro.doe@example.com |           90 |            95 |
Awarded
         2 | Bob              |  30 | bob@example.com
|           34 |           45 |
(7 rows)
```

—--

6.What is the significance of the JOIN operation, and how does it work in PostgreSQL?

Purpose: Combines rows from two or more tables based on a related column between them.
Types: `INNER JOIN`, `LEFT JOIN` (or `LEFT OUTER JOIN`), `RIGHT JOIN` (or `RIGHT OUTER JOIN`), `FULL JOIN` (or `FULL OUTER JOIN`).
Usage: Helps retrieve related data from multiple tables efficiently

Example:

```
university_db=# SELECT students.student_name, courses.course_name
university_db-# FROM students
university_db-# JOIN enrollment ON students.student_id = enrollment.student_id
university_db-# JOIN courses ON enrollment.course_id = courses.course_id;
 student_name | course_name
--------------+-------------
 Alice        | Next.js
 Alice        | React.js
 Bob          | Next.js
 Charlie      | React.js
```

—--

7.Explain the GROUP BY clause and its role in aggregation operations.

Purpose: Groups rows that have the same values into summary rows, typically used with aggregate functions.
Usage: Performs calculations across groups of rows rather than on individual rows.

```
EXAMPLE:
university_db=# SELECT status, COUNT(*) AS count_students
university_db-# FROM students
university_db-# GROUP BY status;
 status  | count_students
---------+----------------
         |              6
 Awarded |              1
(2 rows)
```

—---

8.How can you calculate aggregate functions like COUNT, SUM, and AVG in PostgreSQL?

Aggregate Functions: Perform calculations on a set of values and return a single value.
Examples:

- `COUNT(column)`: Counts the number of rows.
- `SUM(column)`: Computes the sum of values in a column.
- `AVG(column)`: Calculates the average of values in a column.

Typically used with `GROUP BY` to perform calculations on groups of rows.

Example:

```
university_db=#     SELECT COUNT(*) AS total_students, SUM(age) AS total_age,
AVG(age) AS avg_age
university_db-# FROM students;
 total_students | total_age |       avg_age
----------------+-----------+---------------------
              7 |       163 | 23.2857142857142857
(1 row)
```

—---

9.What is the purpose of an index in PostgreSQL, and how does it optimise query performance?

Purpose: Improves the speed of data retrieval operations on a table at the cost of additional storage space and decreased write performance.
Types: B-tree, Hash, GiST, GIN, etc.

Usage: Speeds up query execution by enabling faster data lookup based on indexed columns.
Example:

```
university_db=# CREATE INDEX idx_student_name ON students(student_name);
CREATE INDEX
```

—--

10.Explain the concept of a PostgreSQL view and how it differs from a table.

View:

- **Definition**: Virtual table derived from one or more tables.
- **Usage**: Simplifies complex queries, restricts access to specific columns, or provides summary information.
- **Example**: `CREATE VIEW view_name AS SELECT..;`

Table:

- **Definition**: Physical storage of data in rows and columns.
- **Usage**: Stores persistent data.
- **Example**:

  ```
  university_db=# CREATE VIEW student_courses AS

  university_db-# SELECT students.student_name, courses.course_name

  university_db-# FROM students

  university_db-# JOIN enrollment ON students.student_id = enrollment.student_id

  university_db-# JOIN courses ON enrollment.course_id = courses.course_id;

  CREATE VIEW

  —-------------------------------------------------------------
  ```