**Analytics Lab: Relational Databases**

Scenario

As an analytics consultant for a university, your task is to build a PostgreSQL database to organize student, instructor, and course data. In this lab, you’ll create a database, insert a few records using the provided SQL code files, and run queries to perform the given tasks. This is your opportunity to lay the foundation for data-driven decision-making at the university.

*Upon completion, you will be able to:*

* Create a database in PostgreSQL.
* Populate the database with data.
* Execute SQL queries to accomplish specific tasks.

*Provided code files:*

* [DDL.](https://drive.google.com/file/d/1XIFTiKUQVZIWLvep8EaiZMb6ZRKBxkQq/view?usp=sharing) This is the Data Definition Language SQL code that creates a schema named university and the tables that are part of the schema. Use this the first time.
  + If you like to recreate the schema, to avoid errors, please remember first to drop the existing schema using the code block below:

DROP SCHEMA university CASCADE;

* [SQL code for creating small relations](https://db-book.com/university-lab-dir/sample_tables-dir/smallRelations/smallRelationsInsertFile.sql). This file contains SQL insert statements to load the sample data into all the tables, after first deleting any data the tables currently contain.

*A useful resource:*

* [*Database System Concepts* textbook website](https://www.db-book.com/)

Technical Challenges

1. Create a PostgreSQL database using PgAdmin. Then, run the code in the “DDL” file above in order to set up the database. Next, run the code in the “SQL code for creating small relations” file above in order to populate it with the sample data. Before running this code, set the university schema as the default schema, and set the search path using the code block below:

SET search\_path TO university;

1. Try out some queries, and briefly explain what they do and how many records you see in the results as comments for each query below:

SELECT \* FROM instructor;

SELECT name FROM instructor   
WHERE dept\_name = 'Comp. Sci.' AND salary > 70000;

SELECT \* FROM instructor, department   
WHERE instructor.dept\_name = department.dept\_name;

1. Write a SQL query to find the names of all the instructors from the Biology department.
2. Write a SQL query to find the names of courses in the Computer Science department which have 3 credits.
3. For the student with ID 12345, write a SQL query to show the course\_id and title of all courses registered for by the student.
4. Write a SQL query to display the IDs of all instructors who have never taught a course.
5. Write a SQL query to find the names of all students who have ever taken any Comp. Sci. course. (There should be no duplicate names.)
6. Write a SQL query to find the maximum and minimum enrollment across all sections. Consider only sections that had some enrollment; don't worry about sections with no enrolled students.
7. Write a SQL query to return the section(s) with the highest enrollment for all courses, and list the enrollment for those sections, using a subquery.
8. Grades are mapped to a grade point as follows: A:10, B:8, C:6, D:4 and F:0. Create a table to store these mappings, and use it to write a query to find the cumulative grade point average (GPA). (This is calculated by averaging the grade points earned by a student in all courses. For example, if a student earns an A (4.0), a B (3.0), and a C (2.0) in three different courses, the average would be (4.0 + 3.0 + 2.0) / 3 = 3.0, resulting in a cumulative GPA of 3.0.) To ensure that no student receives a GPA before they have received all their final grades, make sure that, for students who have a null grade in any course they have taken, the cumulative GPA is shown as null.

Discussion Questions

The provided DDL code file includes various constraints (such as primary keys, foreign keys, and check constraints) to ensure data integrity in the university database design.

1. Why are these constraints crucial for maintaining accurate and reliable data?
2. What potential issues might arise if these constraints were not implemented correctly in the database design?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Solution

1. Create a PostgreSQL database using PgAdmin. Then, run the code in the “DDL” file above in order to set up the database. Next, run the code in the “SQL code for creating small relations” file above in order to populate it with the sample data. Before running this code, set the university schema as the default schema and set the search path using the code block below:

SET search\_path TO university;

1. Try out some queries, and briefly explain what they do and how many records you see in the results as comments for each query below:

SELECT \* FROM instructor;

SELECT name FROM instructor   
WHERE dept\_name = 'Comp. Sci.' AND salary > 70000;

SELECT \* FROM instructor, department   
WHERE instructor.dept\_name = department.dept\_name;

**Answer: 12, 2, 12**

1. Write a SQL query to find the names of all the instructors from the Biology department.

|  |
| --- |
| SELECT name  FROM instructor WHERE dept\_name = 'Biology'; |

1. Write a SQL query to find the names of courses in the Computer Science department which have 3 credits.

|  |
| --- |
| SELECT title  FROM course WHERE dept\_name = 'Computer Science' AND credits = 3; |

1. For the student with ID 12345, write a SQL query to show the course\_id and title of all courses registered for by the student.

|  |
| --- |
| SELECT t.course\_id, c.title FROM takes AS t JOIN course AS c ON t.course\_id = c.course\_id WHERE t.ID = '12345'; |

1. Write a SQL query to display the IDs of all instructors who have never taught a course.

|  |
| --- |
| SELECT ID FROM instructor AS i WHERE NOT EXISTS (  SELECT \*   FROM teaches AS t   WHERE i.ID = t.ID ); |

1. Write a SQL query to find the names of all students who have ever taken any Comp. Sci. course. (There should be no duplicate names.)

|  |
| --- |
| SELECT DISTINCT s.name FROM student AS s JOIN takes AS t ON s.ID = t.ID JOIN course AS c ON t.course\_id = c.course\_id WHERE c.dept\_name = 'Comp. Sci.'; |

1. Write a SQL query to find the maximum and minimum enrollment across all sections. Consider only sections that had some enrollment; don't worry about sections with no enrolled students.

|  |
| --- |
| SELECT MAX(enrollment\_count) AS max\_enrollment, MIN(enrollment\_count) AS min\_enrollment FROM (  SELECT course\_id, sec\_id, semester, year, COUNT(\*) AS enrollment\_count  FROM takes  GROUP BY course\_id, sec\_id, semester, year ) AS section\_enrollments; |

1. Write a SQL query to return the section(s) with the highest enrollment for all courses, and list the enrollment for those sections, using a subquery.

|  |
| --- |
| SELECT course\_id, sec\_id, semester, year, enrollment\_count FROM (  SELECT course\_id, sec\_id, semester, year, COUNT(\*) AS enrollment\_count,  RANK() OVER (PARTITION BY course\_id ORDER BY COUNT(\*) DESC) AS rank  FROM takes  GROUP BY course\_id, sec\_id, semester, year ) AS section\_enrollments WHERE rank = 1; |

1. Grades are mapped to a grade point as follows: A:10, B:8, C:6, D:4 and F:0. Create a table to store these mappings, and use it to write a query to find the cumulative grade point average (GPA). (This is calculated by averaging the grade points earned by a student in all courses. For example, if a student earns an A (4.0), a B (3.0), and a C (2.0) in three different courses, the average would be (4.0 + 3.0 + 2.0) / 3 = 3.0, resulting in a cumulative GPA of 3.0.) To ensure that no student receives a GPA before they have received all their final grades, make sure that, for students who have a null grade in any course they have taken, the cumulative GPA is shown as null.

|  |
| --- |
| --Create grade\_points table  CREATE TABLE grade\_points (  grade CHAR(1) PRIMARY KEY,  grade\_point INTEGER  );  -- Insert the grade mappings  INSERT INTO grade\_points (grade, grade\_point) VALUES  ('A', 10),  ('B', 8),  ('C', 6),  ('D', 4),  ('F', 0);    SELECT t.ID,   CASE   WHEN COUNT(t.grade) FILTER (WHERE t.grade IS NULL) > 0 THEN NULL  ELSE AVG(gp.grade\_point)   END AS cumulative\_gpa FROM takes AS t LEFT JOIN grade\_points AS gp ON t.grade = gp.grade GROUP BY t.ID; |

Insights

The constraints in a database design play a crucial role in maintaining data integrity and ensuring that the relationships between tables are logically consistent. For a university database, using primary keys is essential for uniquely identifying records like students, courses, or instructors, preventing duplication and errors. Foreign keys enforce referential integrity, ensuring that relationships (e.g., student enrollments in courses) are valid and correspond to actual data in related tables. For instance, without these constraints, it would be possible to have enrollments linked to non-existent courses or students, leading to inaccuracies in reporting and analysis.

Omitting or incorrectly implementing constraints can lead to significant issues in a university's data management. For example, without check constraints to validate data ranges (like grade points or credit hours), the database could contain incorrect or out-of-bounds values. This can corrupt the reliability of analytics, such as calculating GPA or analyzing course enrollments, making decision-making based on this data flawed. The database design provided in the DDL code effectively uses these constraints to enforce rules and maintain the quality of data, which is foundational for generating accurate insights and supporting future decision-making for the university.