Lab 06: Procedures, Functions and Triggers

Introduction to Software Engineering and Information Systems I

Course 2021/22

David Ruiz, Inma Hernández, Agustín Borrego, Daniel Ayala

Index

1. Objective

The goal of this lab is to implement triggers and procedures in SQL. The student will learn to:

- Use procedures and functions to define a reusable set of commands.
- Use triggers to implement complex constraints and business rules.

2 Environment setup

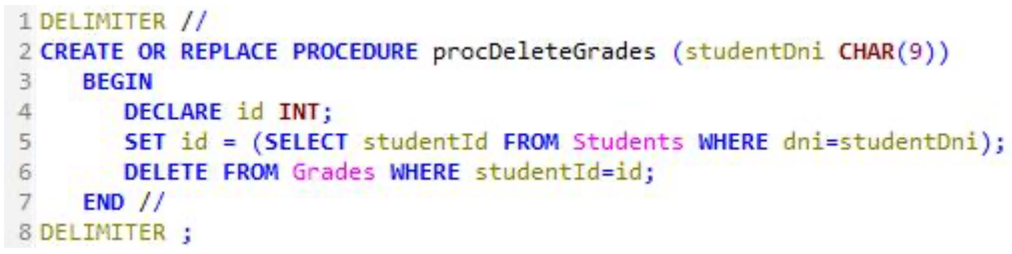
Connect to the "degrees" database and run the tables.sql and populate.sql scripts on it.

Create a triggers.sql file for writing triggers and a procedures.sql file for procedures and functions.

3.Procedures

A procedure is a set of SQL statements that may receive parameters and has a name, as in other programming languages. The main difference between a procedure and a function in SQL is that procedures do not return any value, while functions do.

Procedures are generally used to define a reusable set of instructions that will be used often. For example, we can create the following procedure that deletes all the grades of a student with a given ID:



```
DELIMITER //
CREATE OR REPLACE PROCEDURE procDeleteGrades (studentDni CHAR(9))
  BEGIN
    DECLARE id INT;
    SET id = (SELECT studentId FROM Students WHERE dni=studentDni);
    DELETE FROM Grades WHERE studentId=id;
  END //
DELIMITER ;
```

Observe the following:

- In the code statements that are part of the procedure (between BEGIN and END), semicolons can be problematic, as the interpreter may mistake them for the purpose of the procedure. To avoid this, during its definition we change the symbol used to delimit instructions to // through the DELIMITER statement. When we finish defining the procedure, we reestablish; as a delimiter.
- The first statement, CREATE OR REPLACE PROCEDURE, declares the procedure to be defined and replaces it if one with that name already exists.
- For consistency, and to visually distinguish them more easily from functions, all procedure names defined by us will begin with proc.
- Input parameters are indicated between parentheses, including their type. If there is more than one parameter, they will be separated by commas.
- Within a procedure, variables can be declared using DECLARE including their type, and the value can be assigned using the SET instruction. The value to be assigned can be the result of an SQL query. Notice that SQL queries in this case are wrapped between parentheses.
- In the procedure above, we retrieve the ID of the student with the given DNI, we store it in a variable and we delete all the grades of such student.

Stored procedures can be executed with the CALL instruction, as follows:

```sql
CALL procDeleteGrades('12345678A');
```

Next, we will create a procedure that deletes all the data from the database:

```sql
DELIMITER //
CREATE OR REPLACE PROCEDURE procDeleteData()
   BEGIN
      DELETE FROM Grades;
      DELETE FROM GroupsStudents;
      DELETE FROM Students;
      DELETE FROM Groups;
      DELETE FROM Subjects;
      DELETE FROM Degrees;
   END //
DELIMITER ;
```

4.Functions

Functions are very similar to procedures, but they differ from them in that functions can return values, so they must declare their return type. SQL functions can be used to obtain data that requires multiple SQL statements and are frequently queried.

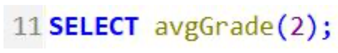
Through an SQL function we can obtain the average grade of a student:

```sql
DELIMITER //
CREATE OR REPLACE FUNCTION avgGrade(studentId INT) RETURNS DOUBLE
   BEGIN
      DECLARE avgStudentGrade DOUBLE;
      SET avgStudentGrade = (SELECT AVG(value) FROM Grades
                                WHERE Grades.studentId = studentId);
      RETURN avgStudentGrade;
   END //
DELIMITER ;
```

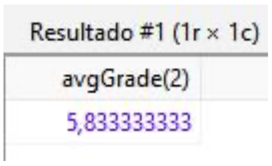
Observe the following:

- The beginning of the declaration is similar, substituting PROCEDURE for FUNCTION and indicating the input parameters if there are any, but the type returned by the function must be indicated with the RETURNS keyword.
- Like in procedures, variables can be declared and values assigned using the DECLARE and SET statements.
- With the RETURN instruction we return the result. A variable or the result of a query can be returned directly (remember the parentheses for SQL queries).
- Like in the procedures, the delimiter change must be made so that the interpreter does not confuse the; inside the function with the end of it.

Unlike procedures, functions can be used anywhere a variable could be used, such as queries, or the body of procedures / functions / triggers. To query the value of a function, instead of using CALL, we can do a SELECT query:

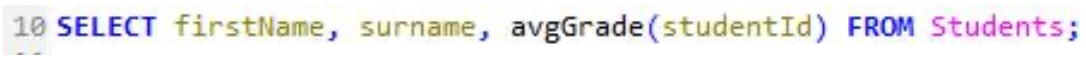


```sql
SELECT avgGrade(2);
```



```
Resultado #1 (1r × 1c)
avgGrade(2)
5,833333333
```

We can also retrieve it as if it were one more column, for example, to obtain the name and surname of a student along with their average grade:

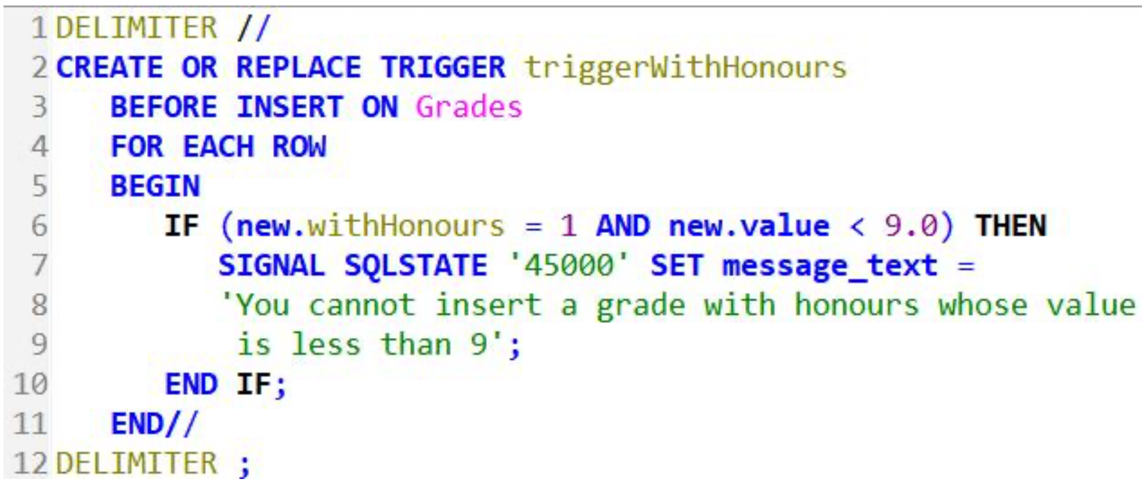


```sql
SELECT firstName, surname, avgGrade(studentId) FROM Students;
```

5.Triggers

Triggers are execute SQL code in the insertion, modification, or deletion of rows in a table. This can help us, for example, to check complex restrictions and implement business rules.

Business rules may be implemented with triggers. For example, if we need to ensure that a student awarded with a grade with honours his/her grade must be greater than or equal to 9:



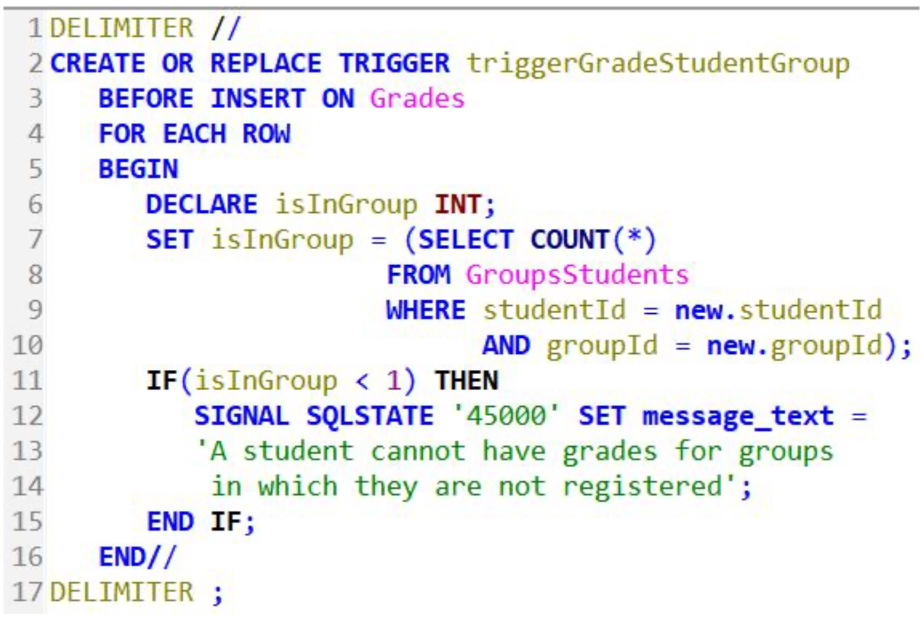
```sql
DELIMITER //
CREATE OR REPLACE TRIGGER triggerWithHonours
   BEFORE INSERT ON Grades
   FOR EACH ROW
   BEGIN
      IF (new.withHonours = 1 AND new.value < 9.0) THEN
         SIGNAL SQLSTATE '45000' SET message_text =
         'You cannot insert a grade with honours whose value
          is less than 9';
      END IF;
   END//
DELIMITER ;
```

Observe the following:

- The delimiter must be changed as in the previous cases.

- Using BEFORE INSERT ON Grades we indicate that the trigger must be executed just before inserting rows into the Grades table. We could substitute BEFORE for AFTER, but in this case, by the time the trigger fires, the grade would have already been inserted.
- Instead of INSERT, UPDATE or DELETE could be used to link triggers to update or delete rows, respectively.
- With only one INSERT operation we could add several records at the same time. Something similar happens with UPDATE and DELETE. With FOR EACH ROW we indicate that the trigger must be executed for each row affected.
- With new we refer to the row that is being inserted, whether the trigger is executed before or after inserting it.
- Through SIGNAL we can raise errors, canceling the insertion of the row. The number after SQLSTATE corresponds to the error code. There are a lot of error codes, although the usual for custom errors is 45000. With SET message_text we indicate what the error message is. It is very useful to include as descriptive a message as possible.
- It would be convenient to ensure that business rules are met not only on row insertion, but also on row updates. To do this, the trigger would have to be duplicated, changing the name and substituting the INSERT keyword for UPDATE.

The above trigger is simple as it only contains a value check and then it throws an error. Let's now implement a trigger that throws an error if you try to insert a grade for a student in a group to which they do not belong:



```
DELIMITER //
CREATE OR REPLACE TRIGGER triggerGradeStudentGroup
   BEFORE INSERT ON Grades
   FOR EACH ROW
   BEGIN
      DECLARE isInGroup INT;
      SET isInGroup = (SELECT COUNT(*)
                       FROM GroupsStudents
                       WHERE studentId = new.studentId
                             AND groupId = new.groupId);
      IF(isInGroup < 1) THEN
         SIGNAL SQLSTATE '45000' SET message_text =
         'A student cannot have grades for groups
          in which they are not registered';
      END IF;
   END//
DELIMITER ;
```

Try the above trigger and observe the following:

- Variables can be declared and assigned using DECLARE and SET just like in procedures and functions.
- In this case, we look for the number of group assignments that match the student and the group to which the grade is being assigned. If there are none, it is because the student is not in that group, and an error is thrown.

Next, we will create a trigger that implements a different business rule: each time a grade is updated, it checks if it has been raised by more than 4 points. In that case, an error is displayed with the student's name and the difference between the new grade and the old one:

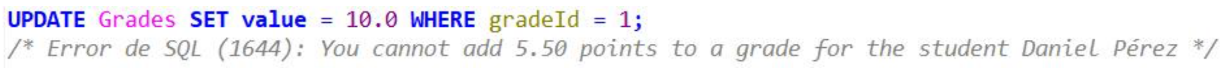
```
DELIMITER //
CREATE OR REPLACE TRIGGER triggerGradesChangeDifference
   BEFORE UPDATE ON Grades
   FOR EACH ROW
   BEGIN
      DECLARE difference DECIMAL(4,2);
      DECLARE student ROW TYPE OF Students;
      SET difference = new.value - old.value;

      IF(difference > 4) THEN
         SELECT * INTO student FROM Students WHERE studentId = new.studentId;
         SET @error_message = CONCAT('You cannot add ', difference,
               ' points to a grade for the student ',
               student.firstName, ' ', student.surname);
         SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = @error_message;
      END IF;
   END//
DELIMITER ;
```

Observe the following:

- In this case, the trigger has not been assigned to the row insertion, but to the row modification, by using BEFORE UPDATE ON.
- One of the declared variables has as a Students table row type, indicated by ROW TYPE OF. Thus, we can access any student attribute that we store in that variable.
- A value is then assigned by executing a SELECT query that only returns one row.
- The assignment in the case of variables that represent rows must be done in a different way: including INTO student within the SELECT query.
- We can refer to the row both before the update (old) and after the update (new).
- To create a custom message that requires multiple parts to be concatenated, we use CONCAT. Since CONCAT cannot be used in the same instruction in which we raised the error, we first create the message in a variable and then we use it.
- The variable in which we have saved the message has not been declared before, and has the symbol '@' in its name. If a variable is used in this way, instead of being a local variable it is a global variable at the user level, which continues to exist and has the same value outside the trigger. We have used it for time (not recommended).

We can test the trigger by trying to raise a note more than 4 points:



```
UPDATE Grades SET value = 10.0 WHERE gradeId = 1;
/* Error de SQL (1644): You cannot add 5.50 points to a grade for the student Daniel Pérez */
```

6.Exercises

Wrap the code for tables creation and data intertion in two separate procedures. Create an additional procedure that creates the database using the previous two.

Implement the following procedures, functions and triggers and use them to check the results obtained:

- Create three procedures: one for the table creation and another one to populate the tables, and then a third one with both merged. Test them isolatedly.
- Create one procedure called pass that, given a score threshold of the same type as the grade value column, changes the score of all grades between the threshold and 5 to 5. Extend the procedure to accept a subject id so that only the grades of that subject are affected. What if we need to treat that subject id as optional and perform the first version if a NULL value is passed, and the second version if the subject id is passed?
- Create one function called avgGradeDegree that, given a degree id, returns the average score of that degree. What if we needed the average score of each group in that degree. What if we wanted to get the average score for the degree and also that of each group in the degree?
- Implement one trigger called triggerLimitNumberStudentsInGroup that raises an error if we try to assign a student to a theory group with 60 or more students, or to a laboratory group with 30 or more students.

To do the exercise, create a copy of the repository for this session with your GitHub user. Make appropriate modifications to the supplied files, creating new files if necessary, and upload those changes to your copy on GitHub. Remember to set the privacy of your repository as "Private" and give dfernandez6 access as a collaborator.