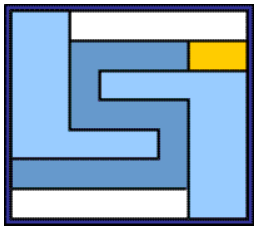

Lab. 06: Procedimientos, funciones y disparadores

Introducción a la Ingeniería del Software y los Sistemas de Información I

Curso 2020/21

David Ruiz, Inma Hernández, Agustín Borrego, Daniel Ayala

Índice

1. Objetivo

El objetivo de esta práctica es implementar disparadores y procedimientos en SQL. El alumno aprenderá a:

- Usar procedimientos y funciones para definir un conjunto de órdenes reutilizable.
- Usar disparadores para implementar restricciones complejas y reglas de negocio.

2. Preparación del entorno

Conéctese a la base de datos “grados” y ejecute en ella los scripts `tables.sql` y `populate.sql`.

Cree un archivo `triggers.sql` para la escritura de los disparadores y un archivo `procedures.sql` para los procedimientos y funciones.

3. Procedimientos

Un procedimiento es un conjunto de sentencias SQL a las que se les asigna un nombre y que reciben unos parámetros, de manera análoga a las funciones de otros lenguajes. La principal diferencia entre un procedimiento y una función en SQL es que los primeros no devuelven ningún valor, mientras que las segundas tienen un valor de retorno.

Generalmente, se emplean para definir un conjunto de instrucciones reutilizable que se espera emplear a menudo. Por ejemplo, para implementar el requisito funcional RF-006 podemos crear el siguiente procedimiento que borra todas las notas de un alumno con un DNI dado:

```
1 DELIMITER //
2 CREATE OR REPLACE PROCEDURE procDeleteGrades (studentDni CHAR(9))
3    BEGIN
4       DECLARE id INT;
5       SET id = (SELECT studentId FROM Students WHERE dni=studentDni);
6       DELETE FROM Grades WHERE studentId=id;
7    END //
8 DELIMITER ;
```

Observe lo siguiente:

- En las instrucciones de código que forman parte del procedimiento (entre `BEGIN` y `END`), los puntos y coma pueden ser problemáticos, ya que el intérprete puede confundirlos con el fin del procedimiento. Para evitar esto, durante su definición **cambiamos el símbolo usado para delimitar instrucciones** a `//` mediante la sentencia `DELIMITER`. Al terminar de definir el procedimiento, reestablecemos `;` como delimitador.
- La primera instrucción, `CREATE OR REPLACE PROCEDURE`, **declara el procedimiento que se va a definir** y lo reemplaza si ya existe uno con ese nombre.
- Por consistencia, y para distinguirlos visualmente más facilmente de las funciones, todos los nombres de procedimientos que definamos **empezarán por** `proc`.
- Se indican los parámetros de entrada **entre paréntesis, incluyendo el tipo de los mismos**. Si hay más de un parámetro, éstos son separados por comas.
- Dentro de un procedimiento **se pueden declarar variables** mediante `DECLARE` incluyendo su tipo, y se les puede asignar un valor mediante `SET`. El valor a asignar puede ser el resultado de una consulta SQL.
- En este procedimiento, buscamos la ID del estudiante que tiene el DNI proporcionado, la almacenamos en una variable y eliminamos todas las notas del estudiante cuya ID hemos almacenado.

Los procedimientos almacenados pueden ser llamados mediante `CALL`, por ejemplo:

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

A continuación, crearemos un procedimiento que borre todos los datos de la base de datos:

```
1  DELIMITER //
2  CREATE OR REPLACE PROCEDURE procDeleteData()
3     BEGIN
4        DELETE FROM Grades;
5        DELETE FROM GroupsStudents;
6        DELETE FROM Students;
7        DELETE FROM Groups;
8        DELETE FROM Subjects;
9        DELETE FROM Degrees;
10    END //
11 DELIMITER ;
```


4. Funciones

Las funciones son muy parecidas a los procedimientos, pero se diferencian de ellos en que las funciones sí pueden devolver valores, por lo que deben declarar su tipo de retorno. Las funciones SQL pueden usarse para obtener datos que requieran varias instrucciones SQL y se quieran consultar a menudo.

Mediante una función SQL podemos implementar el requisito funcional RF-011, para obtener la nota media de un alumno:

```
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 lo siguiente:

- El comienzo de la declaración es similar, sustituyendo `PROCEDURE` por `FUNCTION` e indicando los parámetros de entrada si los hay, pero se debe indicar el tipo que retorna la función mediante `RETURNS`.
- Al igual que en los procedimientos, se pueden declarar y asignar valores a variables mediante `DECLARE` y `SET`.
- Mediante la instrucción `RETURN` devolvemos el resultado. Puede devolverse una variable o el resultado de una consulta directamente.
- Como en los procedimientos, se debe realizar el cambio de delimitador para que el intérprete no confunda los ; del interior de la función con el final de la misma.

Al contrario que los procedimientos, las funciones se pueden usar en cualquier lugar en el que se podría usar una variable, como consultas, o el cuerpo de procedimientos/funciones/disparadores. Para consultar el valor de una función, en lugar de usar `CALL`, podemos hacer una consulta `SELECT`:

```
11 SELECT avgGrade(2);
```

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

También podemos consultarla como si fuera una columna más, por ejemplo, para obtener el nombre y los apellidos de un alumno junto con su nota media:

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

5. Disparadores

Mediante los disparadores (triggers) podemos asociar la ejecución de código a la inserción, modificación, o borrado de filas en una tabla. Esto nos puede servir, por ejemplo, para comprobar restricciones complejas e implementar reglas de negocio.

Como ejemplo, implementamos la regla de negocio RN-006, según la cual para obtener matrícula de honor la nota debe ser mayor o igual a 9:

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

Observe lo siguiente:

- Se debe cambiar el delimitador al igual que en los casos anteriores.

- Mediante `BEFORE INSERT ON Grades` indicamos que el disparador debe ejecutarse **justo antes de insertar filas** en la tabla Grades. Podríamos sustituir `BEFORE` por `AFTER`, pero en este caso, para cuando se lanzara el disparador, la nota ya habría sido insertada.
- En vez de `INSERT` podrían usarse `UPDATE` o `DELETE` para vincular disparadores a la actualización o borrado de filas, respectivamente.
- Con un `INSERT` podríamos insertar varias filas a la vez. Algo similar ocurre con `UPDATE` y `DELETE`. Con `FOR EACH ROW` indicamos que el disparador debe ejecutarse **por cada fila afectada**.
- Con `new` hacemos referencia a **la fila que está siendo insertada**, tanto si el disparador se ejecuta antes como después de insertarla.
- Mediante `SIGNAL` podemos hacer que se produzcan errores, **cancelándose la inserción de la fila**. El número después de `SQLSTATE` corresponde al código de error. Existe una gran cantidad de códigos de error, aunque el usual para los errores personalizados es 45000. Con `SET message_text` indicamos cuál es el mensaje del error. Es muy útil incluir un mensaje **tan descriptivo como sea posible**.
- Sería conveniente que se hiciera la comprobación no sólo al insertar una nota, sino al actualizarla. Para que sea así, habría que repetir el trigger, cambiando el nombre y sustituyendo `INSERT` por `UPDATE`.

El disparador anterior es simple, ya que solo contiene la comprobación de un valor y el lanzamiento de un error. Implementemos ahora un disparador que implementa la regla de negocio RN-009, lanzando un error si intenta introducirse una nota para un alumno en un grupo al que no pertenece:

```
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 ;
```

Pruebe el disparador anterior y observe lo siguiente:

- Se pueden declarar y asignar variables mediante `DECLARE` y `SET` al igual que en los procedimientos y funciones.
- En este caso, buscamos el número de asignaciones a grupos que coinciden con el estudiante y el grupo al que se está intentando asignar la nota. Si no hay ninguna, es porque el estudiante no está en ese grupo, y se lanza un error.

A continuación creamos un disparador que implementa la regla de negocio RN-010: cada vez que se actualice una nota, comprueba si ésta se ha subido en más de 4 puntos. En ese caso, se muestra un error con el nombre del estudiante y la diferencia de la nota nueva con respecto a la antigua:

```
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 lo siguiente:

- En este caso no se ha asignado el disparador a la inserción de filas, sino a la modificación de una fila, mediante `BEFORE UPDATE ON`.
- Una de las variables declaradas tiene como tipo una fila de la tabla Students, indicado mediante `ROW TYPE OF`. Así, podemos acceder a cualquier atributo del estudiante que almacenemos en esa variable.
- Se le asigna un valor mediante una consulta `SELECT` que sabemos que sólo devolverá una fila.
- La asignación en el caso de variables que representan filas se debe hacer de una forma diferente: incluyendo `INTO student` dentro de la consulta.
- Podemos hacer referencia a la fila tanto antes de la actualización (`new`) como después (`old`).
- Para crear un mensaje personalizado que requiera concatenar varias partes, usamos `CONCAT`. Como `CONCAT` no se puede usar en la misma instrucción en la que lanzamos el error, creamos primero el mensaje en una variable y luego lo usamos.
- La variable en la que hemos guardado el mensaje no se ha declarado antes, y tiene en su nombre el símbolo '@'. Si se usa una variable de esta forma, en vez de ser una variable local es una variable a nivel de usuario, que sigue existiendo y teniendo el mismo valor fuera del disparador. La hemos usado por comodidad a la hora de guardar y usar rápidamente el mensaje de error.

Podemos probar el disparador intentando subir una nota más de 4 puntos:

```
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. Ejercicios

Introduzca el código de creación de tablas e inserción de datos en procedimientos separados. Cree un procedimiento adicional que cree la base de datos usando los dos anteriores.

Implemente disparadores que hagan lo siguiente y pruébelos:

- Lanzar un error si una cita de teoría introducida o modificada tiene una hora (no fecha) fuera del horario de tutorías del profesor (RN-005).
- Lanzar un error si la introducción o modificación de un horario de tutorías haría que el profesor tuviera más de 6 horas semanales de tutorías (RN-002). Pista: al introducir o modificar una fila, los agregadores como `SUM` en un disparador `BEFORE` no incluyen la fila siendo insertada o los atributos actualizados.

Para realizar el ejercicio, cree con su usuario de GitHub una copia del repositorio correspondiente a esta sesión. Realice las modificaciones pertinentes en los archivos suministrados, creando nuevos archivos si es necesario, y suba esos cambios a su copia en GitHub. Recuerde de establecer la privacidad de su repositorio como “Private” y dar acceso como colaborador al usuario **iissi**.