Lab5 - Funciones y Disparadores SQL

Pepe Calderón, Fernando Sola, Daniel Ayala, Inma Hernández, Margarita Cruz, Carlos Arévalo, David Ruiz

Escuela Técnica Superior de
Ingeniería Informática

Índice

1 Objetivo

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

- Crear funciones SQL que retornan valores y pueden usarse en consultas.
- Implementar disparadores (triggers) para validar reglas de negocio automáticamente.
- Usar disparadores BEFORE INSERT/UPDATE para validar datos antes de modificarlos.
- Combinar funciones auxiliares con disparadores para código reutilizable.

Los procedimientos almacenados ya se han visto en el L4. En este laboratorio nos centraremos en funciones (que sí retornan valores) y disparadores (que se ejecutan automáticamente ante eventos).

2 Preparación del entorno

Abre HeidiSQL y conéctate con el usuario `iissi_user` a la base de datos `GradesDB`. Asegúrate de haber ejecutado previamente los scripts `createDB.sql` y `populateDB.sql`.

Crea un archivo `functions.sql` para las funciones que implementarás en este laboratorio.

3 Control de versiones

Continuaremos trabajando con el repositorio `GradesDB` creado en L1.

Al inicio del laboratorio, añade el archivo y haz commit:

```
git add functions.sql
git commit -m "Añadido archivo functions.sql para L5"
```

Al finalizar el laboratorio, haz push al repositorio remoto:

```
git add functions.sql
git commit -m "Completado L5: Funciones y disparadores SQL"
git push origin main
```

4

Funciones SQL

Las funciones son similares a los procedimientos pero tienen una diferencia fundamental: **las funciones devuelven un valor** mediante `RETURN` y pueden usarse directamente en consultas SELECT, WHERE, ORDER BY, etc.

4.1. Características de las funciones

- Deben declarar el tipo de dato que retornan mediante `RETURNS`.
- Usan `RETURN` para devolver el resultado.
- Pueden usarse en cualquier lugar donde se usaría una expresión o valor.
- Requieren especificar características como `DETERMINISTIC` o `READS SQL DATA`.
- No pueden modificar datos (no pueden usar INSERT, UPDATE, DELETE).

4.2. Sintaxis básica

```
DELIMITER //
CREATE OR REPLACE FUNCTION nombre_funcion(
    parametro1 TIPO,
    parametro2 TIPO
) RETURNS TIPO_RETORNO
DETERMINISTIC
READS SQL DATA
BEGIN
```

```
    DECLARE v_resultado TIPO_RETORNO;

    -- Lógica de la función
    -- ...

    RETURN v_resultado;
END //
DELIMITER ;
```

Características importantes:

- `DETERMINISTIC`: La función siempre devuelve el mismo resultado para los mismos parámetros.
- `READS SQL DATA`: Indica que la función lee datos pero no los modifica.
- Se debe usar `DELIMITER` igual que con los procedimientos.

4.3. Ejemplo 1: Función para calcular nota media de un estudiante

Esta función calcula el promedio de todas las notas de un estudiante.

```
DELIMITER //
CREATE OR REPLACE FUNCTION f_student_average(
    p_student_id INT
) RETURNS DECIMAL(4,2)
DETERMINISTIC
READS SQL DATA
BEGIN
    DECLARE v_average DECIMAL(4,2) DEFAULT 0;

    SELECT AVG(grade_value) INTO v_average
    FROM grades
    WHERE student_id = p_student_id;

    RETURN v_average;
END //
DELIMITER ;
```

Para ejecutarla:

```
-- Usar en una consulta SELECT
SELECT f_student_average(6) AS 'Promedio del estudiante 6';

-- Usarla junto con otros datos
SELECT p.person_id, p.first_name, p.last_name,
       f_student_average(s.student_id) AS promedio
```

```
FROM people p
JOIN students s ON s.student_id = p.person_id
ORDER BY promedio DESC;

-- Usar en una cláusula WHERE
SELECT p.first_name, p.last_name
FROM people p
JOIN students s ON s.student_id = p.person_id
WHERE f_student_average(s.student_id) >= 7.0;
```

Observe lo siguiente:

- La función retorna `DECIMAL(4,2)` especificado en `RETURNS`.
- Se inicializa `v_average` con `DEFAULT 0` para devolver 0 si el estudiante no tiene notas (en lugar de NULL).
- La función se invoca directamente en consultas, sin necesidad de `CALL`.
- Puede usarse en SELECT, WHERE, ORDER BY, etc.
- Es más concisa que un procedimiento con parámetro OUT para este caso.

4.4. Ejemplo 2: Función para verificar si un estudiante está matriculado en una asignatura

Esta es una función booleana (devuelve TRUE/FALSE) útil para validaciones.

```
DELIMITER //
CREATE OR REPLACE FUNCTION f_is_student_enrolled(
    p_student_id INT,
    p_subject_id INT
) RETURNS BOOLEAN
DETERMINISTIC
READS SQL DATA
BEGIN
    DECLARE v_exists INT;

    SELECT COUNT(*) INTO v_exists
    FROM subject_enrollments
    WHERE student_id = p_student_id
      AND subject_id = p_subject_id;

    RETURN v_exists > 0;
END //
DELIMITER ;
```

Para ejecutarla:

```
-- Listar asignaturas con indicador de matrícula del estudiante 6 del grado
3
SELECT s.subject_id, s.subject_name,
       f_is_student_enrolled(6, s.subject_id) AS matriculado
FROM subjects s
WHERE s.degree_id = 3;

-- Usar en un WHERE
SELECT s.subject_name
FROM subjects s
WHERE f_is_student_enrolled(6, s.subject_id) = TRUE;
```

Observe lo siguiente:

- Devuelve un valor `BOOLEAN` (TRUE o FALSE).
- La expresión `v_exists > 0` devuelve un booleano directamente.
- Las funciones booleanas son muy útiles para validaciones y filtros.
- Pueden usarse en cláusulas WHERE con comparaciones `= TRUE` o `= FALSE`.

4.5. Ejemplo 3: Función para verificar límite de grupos por asignatura

Esta función verifica si una asignatura ha alcanzado el número máximo de grupos permitidos para un tipo de actividad (1 para Teoría, 2 para Laboratorio) en un año académico específico.

```
DELIMITER //
CREATE OR REPLACE FUNCTION f_count_subject_groups(
    p_subject_id INT,
    p_activity VARCHAR(15),
    p_academic_year YEAR
) RETURNS BOOLEAN
DETERMINISTIC
READS SQL DATA
BEGIN
    DECLARE v_count INT;
    DECLARE v_result BOOLEAN DEFAULT FALSE;

    -- Contar grupos del tipo especificado
    SELECT COUNT(*) INTO v_count
    FROM groups
    WHERE subject_id = p_subject_id
      AND activity = p_activity
      AND academic_year = p_academic_year;

    -- Verificar si se alcanzó el límite
    IF p_activity = 'Teoría' THEN
        SET v_result = (v_count >= 1);  -- Máximo 1 grupo de teoría
```

```
    ELSEIF p_activity = 'Laboratorio' THEN
        SET v_result = (v_count >= 2);  -- Máximo 2 grupos de laboratorio
    END IF;

    RETURN v_result;
END //
DELIMITER ;
```

Para ejecutarla:

```
-- Verificar si la asignatura 1 ya tiene el máximo de grupos de teoría en
2025
SELECT f_count_subject_groups(1, 'Teoría', 2025, NULL) AS 'Límite
alcanzado';

-- Listar asignaturas que han alcanzado el límite de grupos de teoría
SELECT s.subject_id, s.subject_name, s.acronym
FROM subjects s
WHERE f_count_subject_groups(s.subject_id, 'Teoría', 2025, NULL) = TRUE;
```

Observe lo siguiente:

- Usa lógica condicional `IF ... THEN ... ELSEIF ... END IF` dentro de la función.
- Implementa lógica de negocio: diferentes límites para teoría (1) y laboratorio (2).
- Retorna un booleano indicando si se alcanzó el límite máximo.
- La condición `p_excluded_group IS NULL OR group_id <> p_excluded_group` permite usar la misma función tanto para INSERT (NULL) como UPDATE (grupo actual).

5 Funciones auxiliares para triggers

Antes de implementar los triggers, necesitamos definir una función auxiliar adicional que encapsula lógica de validación reutilizable. Esta función complementa las ya vistas en los ejemplos anteriores.

5.1. Función: f_student_group_limit

Verifica si un estudiante ha alcanzado el límite de grupos permitidos para una actividad en una asignatura (1 grupo de teoría o 1 grupo de laboratorio por asignatura).

```
DELIMITER //
CREATE OR REPLACE FUNCTION f_student_group_limit(
    p_student_id INT,
    p_subject_id INT,
    p_activity VARCHAR(15),
    p_excluded_group INT
)
RETURNS BOOLEAN
DETERMINISTIC
READS SQL DATA
BEGIN
    DECLARE v_count INT;
    DECLARE v_result BOOLEAN DEFAULT FALSE;

    SELECT COUNT(*) INTO v_count
    FROM group_enrollments ge
```

```
    JOIN groups g ON g.group_id = ge.group_id
    WHERE ge.student_id = p_student_id
      AND g.subject_id = p_subject_id
      AND g.activity = p_activity
      AND (p_excluded_group IS NULL OR ge.group_id <> p_excluded_group);

    IF p_activity = 'Teoría' THEN
        SET v_result = (v_count >= 1);
    ELSEIF p_activity = 'Laboratorio' THEN
        SET v_result = (v_count >= 1);
    END IF;

    RETURN v_result;
END//
DELIMITER ;
```

Observe lo siguiente:

- Cuenta los grupos en los que el estudiante está matriculado para una asignatura y actividad específicas.
- El parámetro `p_excluded_group` permite excluir un grupo del conteo (útil en UPDATEs).
- Retorna TRUE si el estudiante ya alcanzó el límite para esa actividad.
- Se usa en los triggers RN04 para validar límites de grupos por estudiante.

Resumen de funciones auxiliares para triggers:

- `f_is_student_enrolled` : Verifica matrícula en asignatura (vista en Ejemplo 2)
- `f_count_subject_groups` : Verifica límites de grupos por asignatura (vista en Ejemplo 3)
- `f_student_group_limit` : Verifica límites de grupos por estudiante (recién definida)

6

Disparadores (Triggers)

Los disparadores son bloques de código SQL que se ejecutan **automáticamente** cuando ocurre un evento específico en una tabla (INSERT, UPDATE o DELETE). Son fundamentales para implementar reglas de negocio y validaciones complejas.

6.1. Características de los disparadores

- Se ejecutan automáticamente, sin necesidad de llamarlos explícitamente.
- Pueden ejecutarse BEFORE (antes) o AFTER (después) del evento.
- Tienen acceso a los valores NEW (nuevos) y OLD (antiguos) de las filas afectadas.
- Pueden lanzar errores con `SIGNAL` para cancelar la operación.
- Se definen por cada operación (INSERT, UPDATE, DELETE) y momento (BEFORE, AFTER).

6.2. Sintaxis básica

```
DELIMITER //
CREATE OR REPLACE TRIGGER nombre_trigger
{BEFORE | AFTER} {INSERT | UPDATE | DELETE | INSERT OR UPDATE} ON
nombre_tabla
FOR EACH ROW
BEGIN
    -- Código del trigger
    -- Puede usar NEW.columna (valor nuevo)
```

```
    -- Puede usar OLD.columna (valor antiguo en UPDATE/DELETE)
END //
DELIMITER ;
```

Nota importante: MariaDB permite combinar operaciones con `OR`, por ejemplo `BEFORE INSERT OR UPDATE`, lo que evita tener que crear triggers separados para cada operación cuando la lógica es la misma.

6.3. Valores NEW y OLD

Operación	NEW	OLD
INSERT	Valores siendo insertados	No disponible
UPDATE	Valores nuevos después del UPDATE	Valores antiguos antes del UPDATE
DELETE	No disponible	Valores siendo borrados

6.4. Cuándo usar BEFORE vs AFTER

- **BEFORE**: Para validar datos y potencialmente cancelar la operación con SIGNAL.
- **AFTER**: Para realizar acciones después de que la operación se completó exitosamente.

En GradesDB, **todos los triggers son BEFORE** porque se usan para validar reglas de negocio.

7

Triggers en GradesDB

A continuación se presentan los disparadores implementados en `createDB.sql` para validar las reglas de negocio de GradesDB.

7.1. Trigger RN08: Edad mínima para selectividad

Regla de negocio: Un alumno que accede por selectividad debe tener al menos 16 años.

```
DELIMITER //
CREATE OR REPLACE TRIGGER t_biu_students_rn08
BEFORE INSERT OR UPDATE ON students
FOR EACH ROW
BEGIN
    DECLARE v_age TINYINT;
    IF NEW.access_method = 'Selectividad' THEN
        SELECT age INTO v_age FROM people WHERE person_id = NEW.student_id;
        IF v_age < 16 THEN
            SIGNAL SQLSTATE '45000'
                SET MESSAGE_TEXT = 'RN08: No se puede acceder por
Selectividad con menos de 16 años';
        END IF;
    END IF;
END//
DELIMITER ;
```

Observe lo siguiente:

- `BEFORE INSERT OR UPDATE`: Se ejecuta antes de insertar o actualizar en la tabla `students`.
- `FOR EACH ROW`: Se ejecuta por cada fila afectada.
- Se declara una variable local `v_age` para consultar la edad desde la tabla `people`.
- `NEW.access_method`: Accede al valor del método de acceso que se está insertando/actualizando.
- `SIGNAL SQLSTATE '45000'`: Lanza un error personalizado que cancela la operación.
- El error incluye un mensaje descriptivo con el prefijo de la regla (RN08).

7.2. Trigger RN02: Nota solo en grupos donde está matriculado

Regla de negocio: Un alumno solo puede tener notas en grupos a los que pertenece.

```
DELIMITER //
CREATE OR REPLACE TRIGGER t_biu_grades_rn02
BEFORE INSERT OR UPDATE ON grades
FOR EACH ROW
BEGIN
    DECLARE v_count INT;
    SELECT COUNT(*) INTO v_count
    FROM group_enrollments
    WHERE student_id = NEW.student_id
      AND group_id = NEW.group_id;

    IF v_count = 0 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'RN02: El alumno no pertenece al grupo
indicado';
    END IF;
END//
DELIMITER ;
```

Observe lo siguiente:

- Verifica la existencia del estudiante en el grupo mediante un COUNT.
- Si no existe matrícula en el grupo (`v_count = 0`), lanza un error.
- Previene inconsistencias: no pueden existir notas de estudiantes que no están en el grupo.

7.3. Trigger RN01: Una nota por asignatura, convocatoria y año

Regla de negocio: Un alumno no puede tener más de una nota para la misma asignatura, convocatoria y año académico.

```
DELIMITER //
CREATE OR REPLACE TRIGGER t_biu_grades_rn01
BEFORE INSERT OR UPDATE ON grades
FOR EACH ROW
BEGIN
    DECLARE v_subject_id INT;
    DECLARE v_academic_year YEAR;
    DECLARE v_exists INT;

    SELECT subject_id, academic_year INTO v_subject_id, v_academic_year
    FROM groups
    WHERE group_id = NEW.group_id;

    SELECT COUNT(*) INTO v_exists
    FROM grades g
    JOIN groups gr ON gr.group_id = g.group_id
    WHERE g.student_id = NEW.student_id
      AND g.exam_call = NEW.exam_call
      AND gr.subject_id = v_subject_id
      AND gr.academic_year = v_academic_year
      AND (NEW.grade_id IS NULL OR g.grade_id <> NEW.grade_id);

    IF v_exists > 0 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'RN01: Ya existe una nota para la misma
asignatura, convocatoria y año académico';
    END IF;
END//
DELIMITER ;
```

Observe lo siguiente:

- Primero obtiene la asignatura y año académico del grupo.
- Busca si ya existe otra nota con las mismas características.
- La condición `NEW.grade_id IS NULL OR g.grade_id <> NEW.grade_id` diferencia entre INSERT (NULL) y UPDATE (excluye la fila actual).
- Requiere un JOIN para relacionar notas con grupos y obtener la asignatura.

7.4. Trigger RN05: Límite de cambio en notas

Regla de negocio: Una nota no puede modificarse en más de 4 puntos.

```
DELIMITER //
CREATE OR REPLACE TRIGGER t_bu_grades_rn05
BEFORE UPDATE ON grades
FOR EACH ROW
BEGIN
```

```
    IF ABS(NEW.grade_value - OLD.grade_value) > 4 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'RN05: No se puede modificar la nota en más
de 4 puntos';
    END IF;
END//
DELIMITER ;
```

Observe lo siguiente:

- Este trigger es solo `BEFORE UPDATE` (no aplica a INSERT).
- Usa `OLD.grade_value` para el valor antiguo y `NEW.grade_value` para el nuevo.
- `ABS()` calcula el valor absoluto para manejar tanto subidas como bajadas.
- Validación simple pero efectiva para prevenir cambios drásticos en notas.

7.5. Trigger RN03: Máximo de profesores por grupo

Regla de negocio: Un grupo no puede tener más de 2 profesores asignados.

```
DELIMITER //
CREATE OR REPLACE TRIGGER t_bi_teaching_loads_rn03
BEFORE INSERT ON teaching_loads
FOR EACH ROW
BEGIN
    DECLARE v_count INT;

    SELECT COUNT(*) INTO v_count
    FROM teaching_loads
    WHERE group_id = NEW.group_id;

    IF v_count >= 2 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'RN03: El grupo ya tiene 2 profesores
asignados';
    END IF;
END//

CREATE OR REPLACE TRIGGER t_bu_teaching_loads_rn03
BEFORE UPDATE ON teaching_loads
FOR EACH ROW
BEGIN
    DECLARE v_count INT;

    SELECT COUNT(*) INTO v_count
    FROM teaching_loads
    WHERE group_id = NEW.group_id
      AND NOT (professor_id = OLD.professor_id AND group_id =
```

```
OLD.group_id);

    IF v_count >= 2 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'RN03: El grupo ya tiene 2 profesores
asignados';
    END IF;
END//
DELIMITER ;
```

Observe lo siguiente:

- Se definen **dos triggers separados**: uno para INSERT y otro para UPDATE, ya que la lógica de validación es ligeramente diferente.
- Aunque MariaDB permite `INSERT OR UPDATE`, en este caso se separan porque el trigger de UPDATE necesita excluir la fila actual del conteo.
- El trigger de UPDATE excluye la fila actual con `NOT (professor_id = OLD.professor_id AND group_id = OLD.group_id)`.
- Cuenta profesores existentes antes de permitir la asignación.
- Implementa el límite de 2 profesores por grupo.

7.6. Triggers RN04 y RN07: Límites de grupos por estudiante

Reglas de negocio:

- RN04: Un alumno solo puede estar en 1 grupo de teoría y 1 de laboratorio por asignatura.
- RN07: Un alumno debe estar matriculado en la asignatura antes de unirse a un grupo.

Estos triggers **usan funciones auxiliares** para código más limpio:

```
DELIMITER //
CREATE OR REPLACE TRIGGER t_bi_group_enrollments_rn04
BEFORE INSERT ON group_enrollments
FOR EACH ROW
BEGIN
    DECLARE v_subject_id INT;
    DECLARE v_activity VARCHAR(15);
    SELECT subject_id, activity INTO v_subject_id, v_activity
    FROM groups
    WHERE group_id = NEW.group_id;

    IF NOT f_is_student_enrolled(NEW.student_id, v_subject_id) THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'RN07: El alumno debe estar matriculado en
la asignatura';
    END IF;
```

```
    IF f_student_group_limit(NEW.student_id, v_subject_id, v_activity,
NULL) THEN
        IF v_activity = 'Teoría' THEN
            SIGNAL SQLSTATE '45000'
                SET MESSAGE_TEXT = 'RN04: Solo puede haber un grupo de
teoría por asignatura y alumno';
        ELSE
            SIGNAL SQLSTATE '45000'
                SET MESSAGE_TEXT = 'RN04: Solo puede haber dos grupos de
laboratorio por asignatura y alumno';
        END IF;
    END IF;
END//
DELIMITER ;
```

Observe lo siguiente:

- El trigger **llama a funciones** (`f_is_student_enrolled` y `f_student_group_limit`) en lugar de repetir la lógica.
- Esto hace el código más mantenible y reutilizable.
- Valida primero la matrícula en la asignatura (RN07).
- Luego valida el límite de grupos según el tipo de actividad (RN04).
- Las funciones fueron definidas previamente en `createDB.sql` .

7.7. Triggers RN06: Límites de grupos por asignatura

Regla de negocio: Solo puede haber 1 grupo de teoría y 2 de laboratorio por asignatura y año académico.

```
DELIMITER //
CREATE OR REPLACE TRIGGER t_bi_groups_rn06
BEFORE INSERT ON groups
FOR EACH ROW
BEGIN
    IF f_count_subject_groups(NEW.subject_id, NEW.activity,
NEW.academic_year, NULL) THEN
        IF NEW.activity = 'Teoría' THEN
            SIGNAL SQLSTATE '45000'
                SET MESSAGE_TEXT = 'RN06: Solo puede existir un grupo de
teoría por asignatura y año académico';
        ELSE
            SIGNAL SQLSTATE '45000'
                SET MESSAGE_TEXT = 'RN06: Solo pueden existir dos grupos de
laboratorio por asignatura y año académico';
        END IF;
    END IF;
```

```
END//

CREATE OR REPLACE TRIGGER t_bu_groups_rn06
BEFORE UPDATE ON groups
FOR EACH ROW
BEGIN
    IF f_count_subject_groups(NEW.subject_id, NEW.activity,
NEW.academic_year, OLD.group_id) THEN
        IF NEW.activity = 'Teoría' THEN
            SIGNAL SQLSTATE '45000'
                SET MESSAGE_TEXT = 'RN06: Solo puede existir un grupo de
teoría por asignatura y año académico';
        ELSE
            SIGNAL SQLSTATE '45000'
                SET MESSAGE_TEXT = 'RN06: Solo pueden existir dos grupos de
laboratorio por asignatura y año académico';
        END IF;
    END IF;
END//
DELIMITER ;
```

Observe lo siguiente:

- Usa la función `f_count_subject_groups` para verificar el límite.
- El trigger de INSERT pasa `NULL` como grupo a excluir.
- El trigger de UPDATE pasa `OLD.group_id` para excluir el grupo actual del conteo.
- Diferentes mensajes de error según el tipo de actividad.
- La función encapsula la lógica compleja de conteo y validación.

8

Buenas prácticas con Triggers y Funciones

8.1. Separación de responsabilidades

- **Funciones**: Encapsulan lógica de validación reutilizable.
- **Triggers**: Coordinan las validaciones y lanzan errores.

8.2. Mensajes de error descriptivos

- Incluir el código de la regla de negocio (ej: RN01, RN02).
- Explicar claramente qué violación se detectó.
- Facilitar el debugging y comprensión de problemas.

8.3. Nomenclatura consistente

- Triggers: `t_{bi|bu|bd}_{tabla}_{rn}`
  - `bi` = BEFORE INSERT
  - `bu` = BEFORE UPDATE
  - `bd` = BEFORE DELETE
  - `biu` = BEFORE INSERT OR UPDATE

- Funciones: `f_{descripcion}`
- Variables: `v_{nombre}`
- Parámetros: `p_{nombre}`

8.4. Uso de funciones auxiliares

Cuando la misma lógica se necesita en múltiples lugares:

1. Crear una función que encapsule la lógica.
2. Llamar a la función desde los triggers.
3. Mantener el código DRY (Don't Repeat Yourself).

9

Ejercicios propuestos

Implementa las siguientes funciones y triggers en tu archivo `functions.sql`:

1. **Función**: `f_count_student_grades(p_student_id INT, p_exam_call VARCHAR(20))` que cuente cuántas notas tiene un estudiante en una convocatoria específica.
2. **Función**: `f_professor_total_credits(p_professor_id INT)` que calcule el total de créditos que imparte un profesor.
3. **Trigger**: Validar que un grado no pueda tener más de 240 créditos en total sumando todas sus asignaturas.
4. **Trigger**: Validar que un estudiante no pueda estar matriculado en más de 60 créditos por año académico.

> **Info**
>
> Versión PDF disponible