Lab4 - Procedimientos almacenados y Tests SQL

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

Escuela Técnica Superior de
Ingeniería Informática

Índice

Info

Versión PDF disponible

1 Objetivo

El objetivo de esta práctica es aprender a crear procedimientos almacenados en MariaDB y diseñar tests automatizados para validar las reglas de negocio implementadas en la base de datos. El alumno aprenderá a:

- Crear y ejecutar procedimientos almacenados.
- Usar parámetros de entrada (IN) y salida (OUT).
- Implementar lógica de control con manejadores de excepciones (EXCEPTION HANDLER).
- Diseñar tests negativos para validar restricciones.
- Crear tablas de resultados para almacenar el estado de los tests.
- Implementar procedimientos orquestadores para ejecutar baterías de tests.

Los procedimientos almacenados permiten encapsular lógica de negocio en el servidor de base de datos. Los tests automatizados garantizan que las reglas de negocio se cumplan correctamente y facilitan la detección temprana de errores.

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` de los laboratorios anteriores.

Crea un nuevo archivo `tests.sql` en tu repositorio donde implementarás los tests de este laboratorio.

3 Control de versiones

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

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

```
git add tests.sql
git commit -m "Añadido archivo tests.sql para L4"
```

Al finalizar el laboratorio, haz push al repositorio remoto:

```
git add tests.sql
git commit -m "Completado L4: Tests SQL para validación de reglas de
negocio"
git push origin main
```

4

Variables y parámetros en SQL

Antes de trabajar con procedimientos almacenados, es fundamental comprender los diferentes tipos de variables y parámetros que podemos usar en SQL, así como las convenciones de nomenclatura que emplearemos en este laboratorio.

4.1. Variables locales (DECLARE)

Las variables locales se declaran dentro de procedimientos, funciones o bloques BEGIN...END mediante la palabra clave `DECLARE`. Estas variables:

- Solo existen dentro del bloque donde se declaran.
- Se destruyen al finalizar la ejecución del procedimiento o bloque.
- Deben declararse al principio del bloque, antes de cualquier instrucción ejecutable.
- Requieren especificar el tipo de dato (INT, VARCHAR, DECIMAL, etc.).

```
DELIMITER //
CREATE OR REPLACE PROCEDURE ejemplo_variables_locales()
BEGIN
    DECLARE v_contador INT DEFAULT 0;
    DECLARE v_nombre VARCHAR(100);
    DECLARE v_promedio DECIMAL(4,2);

    -- Asignar valores
    SET v_contador = 10;
    SELECT first_name INTO v_nombre FROM people WHERE person_id = 1;
```

```
    SELECT v_contador, v_nombre;
END //
DELIMITER ;
```

4.2. Variables de usuario (@variable)

Las variables de usuario se identifican con el prefijo `@` y tienen características diferentes:

- Existen durante toda la sesión del usuario (persisten entre llamadas).
- No necesitan ser declaradas explícitamente.
- Se crean automáticamente al asignarles un valor.
- Pueden usarse fuera de procedimientos almacenados.
- Útiles para pasar valores entre procedimientos o para almacenar resultados.

```
-- Asignar valor directamente
SET @resultado = 100;

-- Asignar desde una consulta
SELECT AVG(grade_value) INTO @promedio_general FROM grades;

-- Usar en procedimientos
CALL p_student_average(6, @promedio_estudiante);

-- Consultar el valor
SELECT @promedio_estudiante;
```

4.3. Parámetros de procedimientos

Los procedimientos pueden recibir parámetros de tres tipos:

IN (entrada): Valores que se pasan al procedimiento. No pueden ser modificados por el procedimiento (o si se modifican, los cambios no se reflejan fuera).

```
CREATE OR REPLACE PROCEDURE ejemplo_in(
    IN p_student_id INT
)
BEGIN
    SELECT * FROM students WHERE student_id = p_student_id;
END;
```

OUT (salida): Variables que el procedimiento usa para devolver valores. El valor inicial se ignora.

```
CREATE OR REPLACE PROCEDURE ejemplo_out(
    IN p_student_id INT,
    OUT p_average DECIMAL(4,2)
)
BEGIN
    SELECT AVG(grade_value) INTO p_average
    FROM grades
    WHERE student_id = p_student_id;
END;
```

INOUT (entrada/salida): Combinación de ambos. El procedimiento recibe un valor y puede modificarlo.

```
CREATE OR REPLACE PROCEDURE ejemplo_inout(
    INOUT p_contador INT
)
BEGIN
    SET p_contador = p_contador + 1;
END;
```

4.4. Convenciones de nomenclatura

Para mantener el código legible y organizado, seguiremos estas convenciones:

Prefijo	Uso	Ejemplo	Descripción
`v_`	Variables locales (DECLARE)	`v_student_id`, `v_total`, `v_name`	Variables declaradas dentro de procedimientos
`@`	Variables de usuario	`@result`, `@avg_grade`	Variables que persisten en la sesión
`p_`	Parámetros	`p_student_id`, `p_average`, `p_dni`	Parámetros IN, OUT o INOUT de procedimientos

Ejemplo completo aplicando las convenciones:

```
DELIMITER //
CREATE OR REPLACE PROCEDURE p_calculate_and_store(
    IN p_student_id INT,          -- Parámetro de entrada
    OUT p_result DECIMAL(4,2)     -- Parámetro de salida
)
BEGIN
    DECLARE v_count INT;           -- Variable local
    DECLARE v_sum DECIMAL(6,2);    -- Variable local

    -- Contar notas del estudiante
    SELECT COUNT(*), SUM(grade_value)
```

```
    INTO v_count, v_sum
    FROM grades
    WHERE student_id = p_student_id;

    -- Calcular promedio
    IF v_count > 0 THEN
        SET p_result = v_sum / v_count;
    ELSE
        SET p_result = 0;
    END IF;

    -- Guardar en variable de usuario para uso posterior
    SET @last_calculation = p_result;
END //
DELIMITER ;

-- Uso del procedimiento
CALL p_calculate_and_store(6, @average);
SELECT @average AS 'Promedio', @last_calculation AS 'Última calculada';
```

Ventajas de estas convenciones:

- **Claridad**: Se identifica fácilmente el origen y alcance de cada variable.
- **Mantenibilidad**: El código es más fácil de entender y modificar.
- **Prevención de errores**: Se evitan confusiones entre variables locales, parámetros y variables de usuario.
- **Consistencia**: Todo el código sigue el mismo estilo.

5

Procedimientos almacenados: Conceptos básicos

Un procedimiento almacenado es un conjunto de instrucciones SQL que se almacenan en el servidor de base de datos con un nombre específico y que pueden ser ejecutadas mediante una llamada `CALL`. Los procedimientos pueden:

- Recibir parámetros de entrada (IN), salida (OUT) o entrada/salida (INOUT).
- Ejecutar consultas SQL complejas.
- Implementar lógica de control de flujo (IF, WHILE, CASE, etc.).
- Devolver resultados mediante SELECT, aunque no los usaremos con este propósito en la asignatura.
- Lanzar excepciones y manejar errores.

5.1. Sintaxis básica

```
DELIMITER //
CREATE OR REPLACE PROCEDURE nombre_procedimiento(
    IN param1 TIPO,
    IN param2 TIPO,
    OUT resultado TIPO
)
BEGIN
    -- Cuerpo del procedimiento
    -- Instrucciones SQL
```

```
END //
DELIMITER ;
```

Observe lo siguiente:

- `DELIMITER //` cambia el delimitador de sentencias de `;` a `//` para permitir usar `;` dentro del procedimiento.
- `CREATE OR REPLACE PROCEDURE` crea el procedimiento o lo reemplaza si ya existe.
- `IN`, `OUT`, `INOUT` especifican el tipo de parámetro.
- El cuerpo va entre `BEGIN` y `END`.
- Se restaura el delimitador con `DELIMITER ;` al final.

5.2. Ejemplo 1: Procedimiento para borrar notas de un alumno (RF-002)

Este procedimiento borra todas las notas de un estudiante dado su DNI. Muestra cómo usar variables locales y consultas dentro de un procedimiento.

```
DELIMITER //
CREATE OR REPLACE PROCEDURE p_delete_student_grades(
    IN p_student_dni CHAR(9)
)
BEGIN
    DECLARE v_student_id INT;

    -- Buscar el ID del estudiante con el DNI proporcionado
    SELECT student_id INTO v_student_id
    FROM people p
    JOIN students s ON s.student_id = p.person_id
    WHERE p.dni = p_student_dni;

    -- Borrar todas las notas del estudiante
    DELETE FROM grades WHERE student_id = v_student_id;
END //
DELIMITER ;
```

Para ejecutarlo:

```
-- Borrar las notas del estudiante con DNI '12345678A'
CALL p_delete_student_grades('12345678A');

-- Verificar que se borraron
SELECT * FROM grades g
JOIN students s ON s.student_id = g.student_id
JOIN people p ON p.person_id = s.student_id
WHERE p.dni = '12345678A';
```

Observe lo siguiente:

- Se declara una variable local `v_student_id` mediante `DECLARE` incluyendo su tipo.
- Se le asigna un valor mediante una consulta `SELECT ... INTO`. El valor asignado es el resultado de la consulta.
- La variable se usa posteriormente en la instrucción `DELETE` para borrar las notas del estudiante.
- 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.

5.3. Ejemplo 2: Procedimiento para borrar todos los datos

Este procedimiento borra todos los datos de la base de datos respetando el orden de dependencias entre tablas. Es útil para limpiar la base de datos durante pruebas.

```
DELIMITER //
CREATE OR REPLACE PROCEDURE p_delete_all_data()
BEGIN
    -- Borrar datos en orden inverso de dependencias
    DELETE FROM grades;
    DELETE FROM teaching_loads;
    DELETE FROM group_enrollments;
    DELETE FROM subject_enrollments;
    DELETE FROM groups;
    DELETE FROM subjects;
    DELETE FROM students;
    DELETE FROM professors;
    DELETE FROM degrees;
    DELETE FROM people;
END //
DELIMITER ;
```

Para ejecutarlo:

```
-- Borrar todos los datos (¡cuidado con esta operación!)
CALL p_delete_all_data();

-- Verificar que las tablas están vacías
SELECT COUNT(*) AS total_grades FROM grades;
SELECT COUNT(*) AS total_students FROM students;
SELECT COUNT(*) AS total_people FROM people;
```

Observe lo siguiente:

- El procedimiento no recibe parámetros de entrada (los paréntesis están vacíos).
- Se ejecutan múltiples instrucciones `DELETE` en secuencia.
- Es fundamental respetar el orden de las dependencias: primero se borran los datos de las tablas que dependen de otras (como `grades` que depende de `students` y `groups`), y al final las tablas independientes (como `people` y `degrees`).
- Este procedimiento es muy útil durante el desarrollo y las pruebas, pero debe usarse con precaución en producción.

5.4. Ejemplo 3: Procedimiento para calcular el promedio de notas con parámetro OUT

Este procedimiento calcula la nota media de un estudiante y la devuelve mediante un parámetro de salida (OUT).

```
DELIMITER //
CREATE OR REPLACE PROCEDURE p_student_average(
    IN p_student_id INT,
    OUT p_average DECIMAL(4,2)
)
BEGIN
    SELECT AVG(grade_value) INTO p_average
    FROM grades
    WHERE student_id = p_student_id;
END //
DELIMITER ;
```

Para ejecutarlo:

```
-- Consultar la nota media del estudiante con ID 6
CALL p_student_average(6, @avg_student_6);
SELECT @avg_student_6 AS 'Promedio del estudiante 6';

-- Consultar para varios estudiantes
CALL p_student_average(7, @avg_student_7);
CALL p_student_average(8, @avg_student_8);
SELECT @avg_student_6, @avg_student_7, @avg_student_8;
```

Observe lo siguiente:

- Este procedimiento usa un parámetro `OUT` para devolver el resultado calculado.
- Los parámetros `OUT` permiten que el procedimiento devuelva valores al código que lo invoca.
- La consulta `SELECT ... INTO` asigna directamente el resultado al parámetro de salida.
- Para recibir el valor devuelto, se usa una variable de usuario (prefijada con `@`) al llamar al procedimiento.

- Las variables de usuario persisten durante toda la sesión, por lo que pueden consultarse después de la llamada.
- A diferencia de las funciones, los procedimientos no pueden usarse directamente en consultas SELECT, pero son más flexibles para operaciones complejas.

5.5. Ejemplo 4: Procedimiento para subir notas de un grupo

Este procedimiento incrementa en un 15% las notas de los estudiantes de un grupo específico en una convocatoria determinada, pero solo si la nota está entre 4.5 y 8. Demuestra el uso de `UPDATE` con condiciones y cálculos dentro de un procedimiento.

```
DELIMITER //
CREATE OR REPLACE PROCEDURE p_boost_group_grades(
    IN p_group_id INT,
    IN p_exam_call VARCHAR(20)
)
BEGIN
    DECLARE v_affected_rows INT;

    -- Obtener información del grupo
    SELECT group_name INTO v_group_name
    FROM groups
    WHERE group_id = p_group_id;

    -- Actualizar las notas que cumplen los criterios
    UPDATE grades
    SET grade_value = LEAST(grade_value * 1.15, 10.0)
    WHERE group_id = p_group_id
      AND exam_call = p_exam_call
      AND grade_value BETWEEN 4.5 AND 8.0;

END //
DELIMITER ;
```

Para ejecutarlo:

```
-- Ver las notas del grupo 1 antes de la actualización
SELECT g.grade_id, p.first_name, p.last_name, g.grade_value, g.exam_call
FROM grades g
JOIN students s ON s.student_id = g.student_id
JOIN people p ON p.person_id = s.student_id
WHERE g.group_id = 1 AND g.exam_call = 'Primera'
ORDER BY g.grade_value;

-- Subir las notas del grupo 1 en la convocatoria 'Primera'
```

```
CALL p_boost_group_grades(1, 'Primera');

-- Verificar las notas después de la actualización
SELECT g.grade_id, p.first_name, p.last_name, g.grade_value, g.exam_call
FROM grades g
JOIN students s ON s.student_id = g.student_id
JOIN people p ON p.person_id = s.student_id
WHERE g.group_id = 1 AND g.exam_call = 'Primera'
ORDER BY g.grade_value;
```

Observe lo siguiente:

- El procedimiento recibe dos parámetros `IN`: el identificador del grupo y la convocatoria.
- Se usa `UPDATE` con condiciones múltiples en la cláusula `WHERE` para seleccionar solo las notas que cumplen los criterios.
- `LEAST(grade_value * 1.15, 10.0)` garantiza que ninguna nota supere el máximo permitido (10.0), incluso después del incremento del 15%.
- `grade_value BETWEEN 4.5 AND 8.0` filtra solo las notas en el rango especificado.
- `ROW_COUNT()` es una función especial que devuelve el número de filas afectadas por la última operación DML.

5.6. Ejemplo 5: Procedimiento para asignar matrículas de honor

Este procedimiento asigna matrícula de honor al top 5% de estudiantes de un grupo en una convocatoria específica, respetando la restricción de que solo puede haber un máximo del 5% de MH y que la nota debe ser >= 9. Demuestra el uso de subconsultas, `LIMIT` y cálculos dentro de procedimientos.

```
DELIMITER //
CREATE OR REPLACE PROCEDURE p_assign_honors(
    IN p_group_id INT,
    IN p_exam_call VARCHAR(20)
)
BEGIN
    DECLARE v_total_students INT;
    DECLARE v_max_honors INT;

    -- Contar el total de estudiantes matriculados en este grupo
    -- (tengan o no calificación asignada)
    SELECT COUNT(*) INTO v_total_students
    FROM group_enrollments
    WHERE group_id = p_group_id;

    -- Calcular el 5% máximo de matrículas (truncado hacia abajo)
    SET v_max_honors = FLOOR(v_total_students * 0.05);
```

```
    -- Si el 5% es 0, no se pueden asignar matrículas
    IF v_max_honors = 0 THEN
        SELECT CONCAT('No se pueden asignar matrículas de honor: ',
                      'el 5% de ', v_total_students,
                      ' estudiantes matriculados es 0') AS mensaje;
    ELSE
        -- Primero quitar todas las matrículas de honor de este grupo/
convocatoria
        UPDATE grades
        SET with_honors = 0
        WHERE group_id = p_group_id
          AND exam_call = p_exam_call;

        -- Asignar MH a los mejores estudiantes con nota >= 9
        UPDATE grades
        SET with_honors = 1
        WHERE grade_id IN (
            SELECT grade_id
            FROM grades
            WHERE group_id = p_group_id
              AND exam_call = p_exam_call
              AND grade_value >= 9.0
            ORDER BY grade_value DESC
            LIMIT v_max_honors
        );
    END IF;
END //
DELIMITER ;
```

Para ejecutarlo:

```
-- Ver las notas del grupo 1 antes de asignar MH
SELECT g.grade_id, p.first_name, p.last_name, g.grade_value,
       g.with_honors, g.exam_call
FROM grades g
JOIN students s ON s.student_id = g.student_id
JOIN people p ON p.person_id = s.student_id
WHERE g.group_id = 1 AND g.exam_call = 'Primera'
ORDER BY g.grade_value DESC;

-- Asignar matrículas de honor al grupo 1 en la convocatoria 'Primera'
CALL p_assign_honors(1, 'Primera');

-- Verificar las matrículas asignadas
SELECT g.grade_id, p.first_name, p.last_name, g.grade_value,
       g.with_honors, g.exam_call
FROM grades g
JOIN students s ON s.student_id = g.student_id
JOIN people p ON p.person_id = s.student_id
```

```
WHERE g.group_id = 1 AND g.exam_call = 'Primera' AND g.with_honors = 1
ORDER BY g.grade_value DESC;
```

Observe lo siguiente:

- El procedimiento implementa la regla de negocio de que no puede haber más del 5% de matrículas de honor.
- **El 5% se calcula sobre el total de estudiantes matriculados en el grupo** (consultando `group_enrollments`), independientemente de si tienen o no calificación asignada.
- `FLOOR(v_total_students * 0.05)` trunca hacia abajo el cálculo del 5%, garantizando que nunca se supere el límite.
- Se usa `IF ... THEN ... ELSE ... END IF` para manejar el caso especial donde el 5% resulta en 0 estudiantes.
- Primero se quitan todas las MH existentes (`with_honors = 0`) para evitar inconsistencias.
- La subconsulta en el `UPDATE` selecciona los `grade_id` de los mejores estudiantes que cumplen los requisitos.
- `ORDER BY grade_value DESC` ordena las notas de mayor a menor.
- `LIMIT v_max_honors` limita la selección al número máximo calculado.
- Solo se consideran notas >= 9.0, cumpliendo con la restricción de matrícula de honor.

6

Tests automatizados: Estrategia

Los tests automatizados son fundamentales para garantizar que la base de datos cumple con todas las reglas de negocio definidas. En lugar de probar manualmente cada restricción, creamos procedimientos que intentan violar las reglas y verifican que el sistema las rechace correctamente.

6.1. Estrategia de testing

Utilizaremos **tests negativos** para validar que las restricciones funcionan:

1. **Test negativo**: Intenta realizar una operación que viola una regla de negocio.
2. **Resultado esperado**: La base de datos rechaza la operación con un error.
3. **Test PASS**: Si se captura la excepción esperada.
4. **Test FAIL**: Si la operación se ejecuta sin error (la restricción no funciona).

Los **tests positivos** (operaciones válidas) se asumen validados si `populateDB.sql` se ejecuta sin errores, ya que ese script contiene datos que cumplen todas las reglas.

7

Estructura del archivo tests.sql

El archivo `tests.sql` que vamos a crear contendrá:

```
--
-- Autor: [Tu Nombre]
-- Fecha: Enero 2026
-- Descripción: Tests negativos para la BD de Grados
--
USE GradesDB;

-- 1. Tabla de resultados (test_results)
-- 2. Procedimiento auxiliar (p_log_test)
-- 3. Tests individuales (p_test_rn001, p_test_rn002, ...)
-- 4. Procedimiento orquestador (p_run_grados_tests)
-- 5. Llamada al orquestador
```

8 Tabla de resultados

La tabla `test_results` almacena los resultados de cada test ejecutado:

```
-- ================================================================
-- TABLA DE RESULTADOS
-- ================================================================
CREATE OR REPLACE TABLE test_results (
    test_id VARCHAR(20) PRIMARY KEY,
    test_name VARCHAR(200) NOT NULL,
    test_message VARCHAR(500) NOT NULL,
    test_status ENUM('PASS','FAIL','ERROR') NOT NULL,
    execution_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Observe lo siguiente:

- `test_id` : Identificador único del test (ej: 'RN001', 'RN002').
- `test_name` : Nombre descriptivo extraído del mensaje.
- `test_message` : Mensaje completo que describe el test.
- `test_status` : Estado del test:
  - `PASS` : El test pasó (la restricción funcionó)
  - `FAIL` : El test falló (la restricción NO funciona)
  - `ERROR` : Error inesperado
- `execution_time` : Marca temporal de ejecución.

9 Procedimiento auxiliar

El procedimiento `p_log_test` facilita insertar resultados:

```
-- ================================================================
-- PROCEDIMIENTO AUXILIAR
-- ================================================================
DELIMITER //
CREATE OR REPLACE PROCEDURE p_log_test(
    IN p_test_id VARCHAR(20),
    IN p_message VARCHAR(500),
    IN p_status ENUM('PASS','FAIL','ERROR')
)
BEGIN
    INSERT INTO test_results(test_id, test_name, test_message, test_status)
    VALUES (p_test_id, SUBSTRING_INDEX(p_message, ':', 1), p_message,
p_status);
END //
DELIMITER ;
```

Observe lo siguiente:

- Recibe el ID, mensaje y estado del test.
- `SUBSTRING_INDEX(p_message, ':', 1)` extrae el nombre antes del primer `:`.
- Ejemplo: `'RN001: La MH requiere nota >= 9'` → extrae `'RN001'`.

10

Tests (RN001 - RN016)

A continuación implementaremos los 16 tests, uno para cada regla de negocio. Copia cada uno en tu archivo `tests.sql`.

10.1. Test RN001: Matrícula de honor requiere nota >= 9

```
-- ============================================================
-- TESTS (RN001 - RN016)
-- ============================================================

-- Test RN001: Matrícula de honor requiere nota >= 9
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn001_mh_requirement()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN001', 'RN001: La MH requiere nota >= 9',
'PASS');

    CALL p_populate_grades();

    UPDATE grades SET with_honors = 1 WHERE grade_id = 21;

    CALL p_log_test('RN001', 'ERROR: Se permitió MH con nota inferior a 9',
'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- `DECLARE EXIT HANDLER FOR SQLEXCEPTION`: Captura cualquier error SQL.
- Si el UPDATE falla (esperado), el handler registra PASS.
- Si el UPDATE tiene éxito (no debería), se registra FAIL.
- `grade_id = 21` tiene nota 6.2 (< 9), por lo que no puede ser MH.

10.2. Test RN002: No se permiten notas duplicadas

```
-- Test RN002: No se permiten notas duplicadas por asignatura/convocatoria
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn002_duplicate_grade()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN002', 'RN002: No se permiten notas duplicadas
por asignatura/convocatoria', 'PASS');

    CALL p_populate_grades();

    INSERT INTO grades (grade_id, student_id, group_id, grade_value,
exam_call, with_honors)
        VALUES (101, 6, 1, 6.0, 'Primera', 0);

    CALL p_log_test('RN002', 'ERROR: Se permitió duplicar nota en la misma
convocatoria', 'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- El estudiante 6 ya tiene nota en grupo 1, convocatoria 'Primera'.
- El trigger `t_biu_grades_rn01` debe detectar y rechazar esta duplicación.

10.3. Test RN003: Máximo 2 profesores por grupo

```
-- Test RN003: Un grupo no puede tener más de 2 profesores
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn003_professors_per_group()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN003', 'RN003: No se permite añadir un tercer
profesor al grupo', 'PASS');

    CALL p_populate_grades();
```

```
    INSERT INTO teaching_loads (professor_id, group_id, credits) VALUES (5,
1, 1.0);

    CALL p_log_test('RN003', 'ERROR: Se asignó un tercer profesor al
grupo', 'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- El grupo 1 ya tiene 2 profesores (IDs 1 y 2).
- Intentamos asignar el profesor 5 como tercero.
- El trigger `t_bi_teaching_loads_rn03` debe rechazar esto.

10.4. Test RN004: Límite de grupos por alumno

```
-- Test RN004: Un alumno no puede pertenecer a más de un grupo de teoría y
uno de laboratorio por asignatura
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn004_student_group_limit()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN004', 'RN004: Un alumno no puede pertenecer a
más grupos de los permitidos', 'PASS');

    CALL p_populate_grades();

    INSERT INTO group_enrollments (student_id, group_id) VALUES (6, 3);

    CALL p_log_test('RN004', 'ERROR: Se permitió un alumno en más grupos de
los permitidos', 'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- El estudiante 6 ya está en grupo 2 (L1 - laboratorio).
- El grupo 3 es L2 (también laboratorio) de la misma asignatura.
- Solo se permite 1 grupo de laboratorio por asignatura.

10.5. Test RN005: Cambios bruscos en notas

```
-- Test RN005: Una nota no puede alterarse en más de 4 puntos
DELIMITER //
```

```
CREATE OR REPLACE PROCEDURE p_test_rn005_grade_delta()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN005', 'RN005: No se puede modificar la nota en
más de 4 puntos', 'PASS');

    CALL p_populate_grades();

    UPDATE grades SET grade_value = 0.5 WHERE grade_id = 1;

    CALL p_log_test('RN005', 'ERROR: Se permitió modificar la nota en más
de 4 puntos', 'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- `grade_id = 1` tiene valor 9.8.
- Intentamos cambiarla a 0.5 (diferencia de 9.3 puntos).
- El trigger `t_bu_grades_rn05` valida que la diferencia sea ≤ 4.

10.6. Test RN006: Grupos por asignatura y año

```
-- Test RN006: No puede haber más de un grupo de teoría y dos de
laboratorio por asignatura
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn006_extra_group()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN006', 'RN006: No se permite crear un segundo
grupo de teoría para la asignatura', 'PASS');

    CALL p_populate_grades();

    INSERT INTO groups (group_id, subject_id, group_name, activity,
academic_year)
        VALUES (11, 11, 'T2', 'Teoría', 2024);

    CALL p_log_test('RN006', 'ERROR: Se creó un segundo grupo de teoría
para la misma asignatura', 'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- La asignatura 11 (IISSI-1) ya tiene grupo T1 de teoría en 2024.
- Intentamos crear T2 (segundo grupo de teoría).
- El trigger `t_bi_groups_rn06` debe rechazar esto.

10.7. Test RN007: Matrícula previa requerida

```
-- Test RN007: Un alumno sólo puede pertenecer a grupos de asignaturas en
las que está matriculado
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn007_subject_enrollment()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN007', 'RN007: No se puede añadir a un grupo sin
matrícula en la asignatura', 'PASS');

    CALL p_populate_grades();

    INSERT INTO people (person_id, dni, first_name, last_name, age, email)
        VALUES (102, '20000002B', 'Test', 'Alumno', 20,
'nuevo@alum.us.es');
    INSERT INTO students (student_id, access_method) VALUES (102,
'Selectividad');
    INSERT INTO group_enrollments (student_id, group_id) VALUES (102, 1);

    CALL p_log_test('RN007', 'ERROR: Se permitió unir a un grupo sin
matrícula en la asignatura', 'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- Creamos estudiante 102.
- NO lo matriculamos en la asignatura (`subject_enrollments`).
- Intentamos añadirlo directamente al grupo 1.
- El trigger debe rechazar esto.

10.8. Test RN008: Edad mínima para selectividad

```
-- Test RN008: Un alumno no puede acceder por selectividad con menos de 16
años
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn008_min_age()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN008', 'RN008: No se permite Selectividad con
menos de 16 años', 'PASS');

    CALL p_populate_grades();

    INSERT INTO people (person_id, dni, first_name, last_name, age, email)
```

```
        VALUES (104, '20000004D', 'Test', 'Menor', 15, 'menor@alum.us.es');
    INSERT INTO students (student_id, access_method) VALUES (104,
'Selectividad');

    CALL p_log_test('RN008', 'ERROR: Se permitió Selectividad para un
menor', 'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- Persona de 15 años.
- Método de acceso 'Selectividad'.
- El trigger `t_biu_students_rn08` debe rechazar esto.

10.9. Test RN009: Atributos NOT NULL

```
-- Test RN009: Los atributos obligatorios no pueden quedar a NULL
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn009_not_null_attributes()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN009', 'RN009: Los atributos obligatorios no
pueden quedar a NULL', 'PASS');

    CALL p_populate_grades();

    INSERT INTO people (person_id, dni, first_name, last_name, age, email)
        VALUES (103, '20000003C', NULL, 'Campos', 22, 'null@alum.us.es');

    CALL p_log_test('RN009', 'ERROR: Se permitió dejar atributos
obligatorios a NULL', 'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- `first_name` es NOT NULL.
- Intentamos insertar NULL.
- El constraint de la columna debe rechazar esto.

10.10. Test RN010: Créditos válidos

```
-- Test RN010: Los créditos de una asignatura pueden ser 6 o 12
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn010_subject_credits()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN010', 'RN010: Los créditos de una asignatura
pueden ser 6 o 12', 'PASS');

    CALL p_populate_grades();

    INSERT INTO subjects (subject_id, degree_id, subject_name, acronym,
credits, course, subject_type)
        VALUES (31, 3, 'Asignatura Créditos Inválidos', 'ACI', 8, 2,
'Obligatoria');

    CALL p_log_test('RN010', 'ERROR: Se permitió una asignatura con
créditos inválidos', 'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- Créditos = 8 (no permitido).
- El CHECK constraint `rn10_subjects_credits` valida `credits IN (6, 12)`.

10.11. Test RN011: Rango de notas

```
-- Test RN011: El valor de la nota está comprendido entre 0 y 10
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn011_grade_range()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN011', 'RN011: La nota debe estar entre 0 y 10',
'PASS');

    CALL p_populate_grades();

    INSERT INTO grades (grade_id, student_id, group_id, grade_value,
exam_call, with_honors)
        VALUES (201, 6, 1, 11.0, 'Primera', 0);

    CALL p_log_test('RN011', 'ERROR: Se permitió una nota fuera de rango',
'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- Valor 11.0 (fuera del rango 0-10).
- El CHECK constraint `rn11_grades_value` debe rechazar esto.

10.12. Test RN012: Edad de personas

```
-- Test RN012: La edad de las personas está entre 16 y 70 años
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn012_people_age()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN012', 'RN012: La edad de las personas debe estar
entre 16 y 70', 'PASS');

    CALL p_populate_grades();

    INSERT INTO people (person_id, dni, first_name, last_name, age, email)
        VALUES (105, '20000005E', 'Edad', 'Fuera', 80, 'edad@us.es');

    CALL p_log_test('RN012', 'ERROR: Se permitió una edad fuera de rango',
'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- Edad = 80 (fuera del rango 16-70).
- El CHECK constraint `rn12_people_age` debe rechazar esto.

10.13. Test RN013: Duración de grados

```
-- Test RN013: Los años de un grado están entre 3 y 6
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn013_degree_years()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN013', 'RN013: Los grados deben tener entre 3 y 6
años', 'PASS');

    CALL p_populate_grades();

    INSERT INTO degrees (degree_id, degree_name, duration_years)
        VALUES (10, 'Grado Experimental', 2);
```

```
    CALL p_log_test('RN013', 'ERROR: Se permitió un grado con años fuera de
rango', 'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- Duración = 2 años (fuera del rango 3-6).
- El CHECK constraint `rn13_degree_duration` debe rechazar esto.

10.14. Test RN014: Formato de DNI

```
-- Test RN014: El DNI está formado por 8 números y una letra
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn014_dni_format()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN014', 'RN014: El DNI debe tener 8 dígitos y una
letra', 'PASS');

    CALL p_populate_grades();

    INSERT INTO people (person_id, dni, first_name, last_name, age, email)
        VALUES (106, 'INVALIDO', 'DNI', 'Incorrecto', 30, 'dni@us.es');

    CALL p_log_test('RN014', 'ERROR: Se permitió un DNI con formato
inválido', 'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- DNI 'INVALIDO' no cumple el patrón.
- El CHECK constraint `rn14_people_dni` usa REGEXP: `'^[0-9]{8}[A-Za-z]$'`.

10.15. Test RN015: Rango de año académico

```
-- Test RN015: El año académico está entre 2000 y 2100
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn015_academic_year()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN015', 'RN015: El año académico debe estar entre
2000 y 2100', 'PASS');
```

```
    CALL p_populate_grades();

    INSERT INTO groups (group_id, subject_id, group_name, activity,
academic_year)
        VALUES (20, 12, 'MD-T2025', 'Teoría', 1999);

    CALL p_log_test('RN015', 'ERROR: Se permitió un año académico fuera de
rango', 'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- Año 1999 (fuera del rango 2000-2100).
- El CHECK constraint `rn15_groups_year` debe rechazar esto.

10.16. Test RN016: Rango de curso

```
-- Test RN016: El curso de una asignatura está entre 1 y 6
DELIMITER //
CREATE OR REPLACE PROCEDURE p_test_rn016_course_range()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        CALL p_log_test('RN016', 'RN016: El curso de una asignatura debe
estar entre 1 y 6', 'PASS');

    CALL p_populate_grades();

    INSERT INTO subjects (subject_id, degree_id, subject_name, acronym,
credits, course, subject_type)
        VALUES (30, 3, 'Asignatura Fuera de Curso', 'AFC', 6, 0,
'Obligatoria');

    CALL p_log_test('RN016', 'ERROR: Se permitió un curso fuera de rango',
'FAIL');
END //
DELIMITER ;
```

Observe lo siguiente:

- Curso = 0 (fuera del rango 1-6).
- El CHECK constraint `rn16_subjects_course` debe rechazar esto.

11 Procedimiento orquestador

El procedimiento `p_run_grados_tests` ejecuta todos los tests y muestra resultados:

```
-- ================================================================
-- ORQUESTADOR
-- ================================================================
DELIMITER //
CREATE OR REPLACE PROCEDURE p_run_grados_tests()
BEGIN
    -- Si los casos positivos fallan, no ejecutar los negativos
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        SELECT 'ERROR: El populate falló. No se ejecutaron los tests
positivos.';

    -- Ejecutar populate una sola vez para casos positivos
    CALL p_populate();

    -- Si llegamos aquí, el populate funcionó correctamente y se han pasado
los tests positivos
    DELETE FROM test_results;
    CALL p_test_rn001_mh_requirement();
    CALL p_test_rn002_duplicate_grade();
    CALL p_test_rn003_professors_per_group();
    CALL p_test_rn004_student_group_limit();
    CALL p_test_rn005_grade_delta();
    CALL p_test_rn006_extra_group();
    CALL p_test_rn007_subject_enrollment();
    CALL p_test_rn008_min_age();
    CALL p_test_rn009_not_null_attributes();
```

```
    CALL p_test_rn010_subject_credits();
    CALL p_test_rn011_grade_range();
    CALL p_test_rn012_people_age();
    CALL p_test_rn013_degree_years();
    CALL p_test_rn014_dni_format();
    CALL p_test_rn015_academic_year();
    CALL p_test_rn016_course_range();

    SELECT * FROM test_results ORDER BY execution_time, test_id;
    SELECT test_status, COUNT(*) AS total FROM test_results GROUP BY
test_status;
END //
DELIMITER ;
```

Observe lo siguiente:

- `p_populate` ejecuta los tests positivos, si falla no se deben ejecutar tests negativos
- `DELETE FROM test_results` : Limpia ejecuciones anteriores.
- Llama a los 16 tests en orden.
- Primera consulta: resultados detallados.
- Segunda consulta: resumen por estado.

12

Ejecución de los tests

Al final del archivo añade:

```
CALL p_run_grados_tests();
```

Para ejecutar todos los tests:

1. Abre `tests.sql` en HeidiSQL.
2. Presiona F9 o haz clic en “Ejecutar”.
3. Observa los resultados.

13 Interpretación de resultados

Obtendrás dos tablas de resultados:

13.1. Tabla 1: Resultados detallados

test_id	test_name	test_message	test_status	execution_time
RN001	RN001	RN001: La MH requiere nota >= 9	PASS	2026-01-14 10:30:01
RN002	RN002	RN002: No se permiten notas duplicadas...	PASS	2026-01-14 10:30:02
...	...	...	...	...

13.2. Tabla 2: Resumen

test_status	total
PASS	16

Interpretación:

- **PASS** ☒: La restricción funciona (rechazó operación inválida).
- **FAIL** ×: La restricción NO funciona (permitió operación inválida).
- **ERROR** ⚠: Error inesperado.

Resultado esperado: Los 16 tests deben mostrar PASS.

14

Push final

```
git add tests.sql
git commit -m "Completado L4: Tests SQL para validación de reglas de
negocio"
git push origin main
```

15

Resumen

En este laboratorio has aprendido:

- ✅ Crear procedimientos almacenados con `DELIMITER` y `CREATE OR REPLACE PROCEDURE`.
- ✅ Usar exception handlers (`DECLARE EXIT HANDLER FOR SQLEXCEPTION`).
- ✅ Diseñar tests negativos para validar restricciones.
- ✅ Crear tablas de resultados para tests automatizados.
- ✅ Implementar procedimientos orquestadores.
- ✅ Interpretar resultados de baterías de tests.

Los tests automatizados son herramientas fundamentales para mantener la calidad y consistencia de las bases de datos en entornos de producción.