

Tema 14

Programación en SGBD.

Cada gestor de BdD tiende a tener su propio [lenguaje](#):

- MySQL y otros incorporan el estándar [SQL/PSM](#). Nosotros nos centraremos en este.
- Está muy extendido también [PL/SQL](#) que es de Oracle.
- Una ventaja del *fork* de Mysql, [MariaDB](#) es que permite usar ambos.

14.1. Funciones, procedimientos y triggers.

Los *scripts* que realicemos se almacenan en un BdD como objetos de la misma. Pueden ser de tres tipos:

- **Función almacenada:** usadas para crear funciones similares a las usadas durante el curso (COUNT, MAX, etc.). Puede tener cero o muchos parámetros de entrada y siempre devuelve un valor (y solo un valor).
Generalmente se utilizan para realizar cálculos.
Se crea con la sentencia CREATE FUNCTION y se invoca con la sentencia SELECT o dentro de una expresión.
- **Procedimiento almacenado:** puede tener cero o muchos parámetros de entrada y cero o muchos parámetros de salida.
Generalmente se utilizan para realizar lógica de negocio.
Se crea con la sentencia CREATE PROCEDURE y se invoca con la sentencia CALL. **NO** pueden invocarse desde consultas.
- **Trigger (disparador):** se activa cuando ocurre un evento de inserción, actualización o borrado, sobre la tabla a la que está asociado.

Se crea con la sentencia `CREATE TRIGGER` y tiene que estar asociado a una tabla.

Ventajas:

- Reducir el tráfico de red: se envía la llamada al objeto en lugar de todas las instrucciones SQL.
- Centralizar la lógica de negocio: en lugar de tener que escribir el código para cada aplicación, se puede realizar una vez en la BdD e invocarse desde todas.
- Mejorar la seguridad: se pueden conceder privilegios sobre los objetos concretos (procedimiento o función) en lugar de sobre las tablas subyacentes.

Desventajas:

- Consumo de recursos: los objetos se almacenan en memoria para ejecutarse, para cada sesión, por lo que pueden llegar a consumir mucha memoria.
- Depuración (*Debugging*): no es fácil ya que MySQL no nos provee de herramientas para ello.

14.2. Procedimientos almacenados.

NOTA: en este apartado incluiremos, además de cómo crear un procedimiento, la sintaxis del lenguaje.

Sintaxis:

```

CREATE
  [DEFINER = user]
  PROCEDURE sp_name ([proc_parameter[, ... ]])
  [characteristic ...] routine_body

proc_parameter:
  [ IN | OUT | INOUT ] param_name type

type:
  Any valid MySQL data type

characteristic: {
  COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}

routine_body:
  Valid SQL routine statement

```

14.2.1. Parámetros de entrada, de salida y de entrada/salida.

Tres tipos de parámetros:

- **Entrada:** sirven para pasar un valor al procedimiento al llamarlo.
Aunque el valor del parámetro se modifique en el procedimiento, al finalizar tendrá el valor que tenía antes de que comenzara (la modificación no será visible para quien llama al procedimiento).
Se indican poniendo la palabra reservada IN delante del nombre del parámetro.
En programación sería equivalente al paso por valor de un parámetro.
- **Salida:** sirven para devolver parámetros desde el procedimiento a quien lo ha llamado. Inicialmente su valor será NULL.
Se indican poniendo la palabra reservada OUT delante del nombre del parámetro.
En programación sería equivalente al paso por referencia de un parámetro si no usamos el valor que nos pasan en él (o fuera vacío).
- **Entrada/Salida:** Es una combinación de los tipos IN y OUT.
Estos parámetros se indican poniendo la palabra reservada IN/OUT delante del

nombre del parámetro.

En programación sería equivalente al paso por referencia de un parámetro.

14.2.2. Crear procedimientos.

Usamos `CREATE PROCEDURE`.

`BEGIN` y `END` delimitan el código (equivalente a `{}` en Java).

Para llamar al procedimiento se utiliza `CALL`.

Cuando llamamos al procedimiento por primera vez en una sesión, se busca por nombre en el catálogo de MySQL, se compila y se almacena en memoria caché.

Las siguientes veces que se llame, en la misma sesión, se ejecutará sin tener que recompilarlo.

Desde un procedimiento almacenado se puede llamar a otros procedimientos o funciones, permitiéndonos escribir un código modular.

Cambiar el delimitador:

En SQL se utiliza el punto y coma para separar sentencias. Cuando queremos crear un *script* debemos modificarlo para que no se confunda el que usamos en el script con el que usamos para marcar dónde comienza y termina el procedimiento. Lo habitual es cambiarlo por `$$` o `//`

```
DELIMITER $$
```

Y luego devolverlo a su valor original:

```
DELIMITER ;
```

14.2.2.1. Ejemplos.

Sobre jardinería.

Parámetros de entrada:

Procedimiento llamado `listar_productos` que recibe como entrada el nombre de una gama y muestre un listado de todos los productos que existen dentro de esa gama.

Este procedimiento no devuelve ningún parámetro de salida, lo que hace es mostrar el listado de los productos.

```
DELIMITER $$

DROP PROCEDURE IF EXISTS listar_productos$$
CREATE PROCEDURE listar_productos(IN gama VARCHAR(50))
BEGIN
    SELECT *
    FROM producto
    WHERE producto.gama = gama;
END $$

DELIMITER ;
```

y lo llamamos:

```
CALL listar_productos('Herramientas');
```

Parámetros de salida:

Procedimiento llamado `contar_productos` que recibe como entrada el nombre de una gama y devuelve el número de productos que existen dentro de esa gama.

Utilizando SET y una variable:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS contar_productos$$
CREATE PROCEDURE contar_productos(IN gama VARCHAR(50),
    OUT total INT UNSIGNED)
BEGIN
    SET total = (
        SELECT COUNT(*)
        FROM producto
        WHERE producto.gama = gama);
END
$$

DELIMITER ;
CALL contar_productos('Herramientas', @total);
SELECT @total;
```

Usando SELECT INTO:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS contar_productos$$
CREATE PROCEDURE contar_productos(IN gama VARCHAR(50),
    OUT total INT UNSIGNED)
BEGIN
    SELECT COUNT(*)
    INTO total
    FROM producto
    WHERE producto.gama = gama;
END
$$

DELIMITER ;
CALL contar_productos('Herramientas', @total);
SELECT @total;
```

Otro procedimiento, que se llame `calcular_max_min_media`, que recibe como parámetro de entrada el nombre de la gama de un producto y devuelve como salida tres parámetros: el precio máximo, el precio mínimo y la media de los productos que existen en esa gama.

Utilizando SET

```
DELIMITER $$
DROP PROCEDURE IF EXISTS calcular_max_min_media$$
CREATE PROCEDURE calcular_max_min_media(
    IN gama VARCHAR(50),
    OUT maximo DECIMAL(15, 2),
    OUT minimo DECIMAL(15, 2),
    OUT media DECIMAL(15, 2)
)
BEGIN
    SET maximo = (
        SELECT MAX(precio_venta)
        FROM producto
        WHERE producto.gama = gama);

    SET minimo = (
        SELECT MIN(precio_venta)
        FROM producto
        WHERE producto.gama = gama);

    SET media = (
        SELECT AVG(precio_venta)
        FROM producto
        WHERE producto.gama = gama);
END $$

DELIMITER ;
CALL calcular_max_min_media('Herramientas', @maximo,
    @minimo, @media);
SELECT @maximo, @minimo, @media;
```

Usando SELECT ... INTO (observa que INTO puede ir al final)

```
DELIMITER $$
DROP PROCEDURE IF EXISTS calcular_max_min_media$$
CREATE PROCEDURE calcular_max_min_media(
    IN gama VARCHAR(50),
    OUT maximo DECIMAL(15, 2),
    OUT minimo DECIMAL(15, 2),
    OUT media DECIMAL(15, 2)
)
BEGIN
    SELECT
        MAX(precio_venta),
        MIN(precio_venta),
        AVG(precio_venta)
    FROM producto
    WHERE producto.gama = gama
    INTO maximo, minimo, media;
END $$

DELIMITER ;
CALL calcular_max_min_media('Herramientas', @maximo,
    @minimo, @media);
SELECT @maximo, @minimo, @media;
```

14.2.3. Consultar los procedimientos.

Podemos ver todos los que hay en el SGBD:

```
SHOW PROCEDURE STATUS;
```

o los que cumplan alguna condición:

```
SHOW PROCEDURE STATUS WHERE db = 'jardineria';
```

```
SHOW PROCEDURE STATUS WHERE db = 'sakila' AND definer LIKE 'ad%';
```

También podemos consultarlos de la tabla `information_schema.routines`:


```
SELECT
    routine_name
FROM
    information_schema.routines
WHERE
    routine_type = 'PROCEDURE'
AND routine_schema = 'jardineria';
```

Si usas * verás que tiene mucha más información.

Ejercicios 14.1

Debes incluir llamadas a los procedimientos que se piden.

1. Crea un procedimiento que reciba un parámetro de tipo VARCHAR que recibe un texto y lo muestre por pantalla.
2. Crea un procedimiento que muestre la información de cada empleado y el departamento donde trabaja.
3. Modifica el procedimiento anterior para que reciba un nombre de departamento y restrinja el resultado a los empleados que trabajan en él.
4. Crea un procedimiento que reciba un nombre de departamento y devuelva cuántos empleados trabajan en él.
5. Crea un procedimiento que reciba un nombre de departamento, lo cambie a mayúsculas en la tabla y lo devuelva en mayúsculas en el mismo parámetro.
6. Elimina los procedimientos creados.

14.3. Estructuras de programación.

14.3.1. Variables.

Para declarar una variable:

```
DECLARE var_name [, var_name] ... type [DEFAULT value]
```

El ámbito de una variable local será el bloque BEGIN y END del procedimiento o la función donde ha sido declarada.

Una restricción importante es que se deben declarar antes de los cursores y los handlers (los veremos más adelante).

Ejemplos:

```
DECLARE total INT UNSIGNED;
```

```
DECLARE ventaTotal DEC(10,2) DEFAULT 0.0;
```

```
DECLARE x, y INT DEFAULT 0;
```

Para asignarle un valor podemos hacerlo con SET

```
SET total = 10;
```

o SELECT INTO:

```
SELECT COUNT(*)  
INTO total  
FROM empleados;
```

IMPORTANTE: una variable que lleva @ delante del nombre es una variable de sesión que estará disponible hasta que la sesión termine.

IMPORTANTE:

Debes tener en cuenta que el procedimiento se puede crear en cualquier momento y se almacenará en la BdD.

Podrá ser llamado en cualquier momento desde la misma u otra sesión.

Aquí se incluye la llamada en los ejemplos para facilitar probarlo.

Un ejemplo más completo (ten en cuenta que el resultado de la operación se podría haber asignado directamente al parámetro de salida pero se utiliza una variable para mostrar su uso):

```
DELIMITER $$
DROP PROCEDURE IF EXISTS contar_productos$$
CREATE PROCEDURE contar_productos(
    IN gama VARCHAR(50),
    OUT numProd INT UNSIGNED
)
BEGIN
    -- Paso 1. Declaramos una variable local
    DECLARE total INT UNSIGNED;

    -- Paso 2. Contamos los productos
    SET total = (
        SELECT COUNT(*)
        FROM producto
        WHERE producto.gama = gama);

    -- Paso 3. Asignamos el resultado al parámetro de salida.
    SET numProd = total;
END
$$

DELIMITER ;
CALL contar_productos('Herramientas', @numero);
SELECT @numero;
```

14.3.2. Instrucciones condicionales.

14.3.2.1. IF.

Sintaxis:

```
IF search_condition THEN statement_list
    [ELSEIF search_condition THEN statement_list] ...
    [ELSE statement_list]
END IF
```

Ejemplos:

```
DELIMITER $$

DROP PROCEDURE IF EXISTS nivelCompositor$$
CREATE PROCEDURE nivelCompositor(
    IN idCompo INT,
    OUT nivelCompo VARCHAR(20))
BEGIN
    DECLARE numObras INT UNSIGNED DEFAULT 0;

    SELECT
        COUNT(*)
    INTO
        numObras
    FROM
        Compositor c,
        Obra o
    WHERE
        c.id_compositor = o.id_compositor
        AND c.id_compositor = idCompo;

    IF numObras ≥ 6 THEN
        SET nivelCompo = 'PLATINO';
    END IF;
END$$

DELIMITER ;
```

En el mismo procedimiento, podríamos ver un IF con ELSE.

```
IF numObras ≥ 6 THEN
    SET nivelCompo = 'PLATINO';
ELSE
    SET nivelCompo = 'OTRO';
END IF;
```

o ELSE IF

```
IF numObras ≥ 6 THEN
    SET nivelCompo = 'PLATINO';
ELSEIF numObras ≥ 4 THEN
    SET nivelCompo = 'ORO';
ELSE
    SET nivelCompo = 'OTRO';
END IF;
```

14.3.2.2. CASE.

Existen dos formas de utilizar CASE:

Comparamos una variable con posibles valores

```
CASE case_value
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

o

Establecemos comparaciones

```
CASE
    WHEN search_condition THEN statement_list
    [WHEN search_condition THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

Podemos ver ejemplos con un procedimiento similar a los anteriores:

```
CREATE PROCEDURE nivelCompositor(  
  IN idCompo INT,  
  OUT nivelCompo VARCHAR(20))  
BEGIN  
  DECLARE numObras INT UNSIGNED DEFAULT 0;  
  
  SELECT  
    COUNT(*)  
  INTO  
    numObras  
  FROM  
    Compositor c,  
    Obra o  
  WHERE  
    c.id_compositor = o.id_compositor  
    AND c.id_compositor = idCompo;  
  
  CASE numObras  
    WHEN 6 THEN  
      SET nivelCompo = 'PLATINO';  
    WHEN 5 THEN  
      SET nivelCompo = 'ORO';  
    WHEN 4 THEN  
      SET nivelCompo = 'PLATA';  
    ELSE  
      SET nivelCompo = 'OTRO';  
  END CASE;  
END$$
```

O con comparaciones:

```
CASE
  WHEN numObras ≥ 6 THEN
    SET nivelCompo = 'PLATINO';
  WHEN numObras ≥ 4 THEN
    SET nivelCompo = 'ORO';
  WHEN numObras ≥ 2 THEN
    SET nivelCompo = 'PLATA';
  ELSE
    SET nivelCompo = 'OTRO';
END CASE;
```

14.3.3. Bucles.

Hay tres tipos:

- LOOP
- REPEAT
- WHILE

14.3.3.1. WHILE

```
[begin_label:] WHILE search_condition DO
statement_list
END WHILE [end_label]
```

Ejemplo:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS ejemplo_bucle_while$$
CREATE PROCEDURE ejemplo_bucle_while(IN tope INT, OUT suma INT)
BEGIN
    DECLARE contador INT;

    SET contador = 1;
    SET suma = 0;

    WHILE contador ≤ tope DO
        SET suma = suma + contador;
        SET contador = contador + 1;
    END WHILE;
END
$$

DELIMITER ;
CALL ejemplo_bucle_while(10, @resultado);
SELECT @resultado;
```

14.3.3.2. LOOP

Sintaxis:

```
[begin_label:] LOOP
    statement_list
END LOOP [end_label]
```

Ejemplo:


```
DELIMITER $$
DROP PROCEDURE IF EXISTS ejemplo_bucle_loop$$
CREATE PROCEDURE ejemplo_bucle_loop(IN tope INT, OUT suma INT)
BEGIN
    DECLARE contador INT;

    SET contador = 1;
    SET suma = 0;

    bucle: LOOP
        IF contador > tope THEN
            LEAVE bucle;
        END IF;

        SET suma = suma + contador;
        SET contador = contador + 1;
    END LOOP;
END $$

DELIMITER ;
CALL ejemplo_bucle_loop(10, @resultado);
SELECT @resultado;
```

¿Dónde está la condición? se hace con LEAVE que tiene un comportamiento equivalente a break en Java.

Otro:

```
CREATE PROCEDURE ejemplo_bucle_loop2(IN tope INT,  
  OUT str VARCHAR(255))  
BEGIN  
  DECLARE contador INT;  
  
  SET contador = 1;  
  SET str = '';  
  
  bucle: LOOP  
    IF contador > tope THEN  
      LEAVE bucle;  
    END IF;  
  
    SET contador = contador + 1;  
    IF (contador mod 2) THEN  
      ITERATE bucle;  
    ELSE  
      SET str = CONCAT(str,contador,',');  
    END IF;  
  END LOOP;  
END$$
```

ITERATE es equivalente a continue aunque su colocación en el ejemplo no tiene mucho sentido, ya que ELSE hace lo mismo.

Podríamos haberlo hecho así también:

```
CREATE PROCEDURE ejemplo_bucle_loop2(IN tope INT,  
  OUT str VARCHAR(255))  
BEGIN  
  DECLARE contador INT;  
  
  SET contador = 1;  
  SET str = '';  
  
  bucle: LOOP  
    IF contador > tope THEN  
      LEAVE bucle;  
    END IF;  
  
    SET contador = contador + 1;  
    IF (contador mod 2) THEN  
      ITERATE bucle;  
    END IF;  
  
    SET str = CONCAT(str,contador,',');  
  END LOOP;  
END$$
```

bucle es una etiqueta; podríamos haber usado algo más significativo como por ejemplo cuenta.

La etiqueta es opcional pero para usar LEAVE o ITERATE es necesaria.

14.3.3.3. REPEAT

Sintaxis:

```
[begin_label:] REPEAT  
  statement_list  
UNTIL search_condition  
END REPEAT [end_label]
```

Aquí si podemos incluir la condición y tiene un comportamiento similar a un do-while:

```
use pruebas;

DELIMITER $$
DROP PROCEDURE IF EXISTS ejemplo_bucle_repeat$$
CREATE PROCEDURE ejemplo_bucle_repeat(IN tope INT, OUT suma INT)
BEGIN
    DECLARE contador INT;

    SET contador = 1;
    SET suma = 0;

    REPEAT
        SET suma = suma + contador;
        SET contador = contador + 1;
    UNTIL contador > tope
    END REPEAT;
END $$

DELIMITER ;
CALL ejemplo_bucle_repeat(10, @resultado);
SELECT @resultado;
```

Observa que no estamos usando etiqueta porque no vamos a necesitar usar LEAVE pero si lo necesitáramos (o ITERATE) tendríamos que añadirla.

Lo mismo sucedería en WHILE que vimos al principio.

Ejercicios 14.2

1. Crea un procedimiento que reciba un número entero que represente una temperatura y muestre si el agua se encontraría en estado sólido, líquido o gaseoso a dicha temperatura.
2. Realiza un procedimiento que reciba dos parámetros, uno con un número del uno al siete (día de la semana) y otro con un número del uno al doce (mes del año). Luego con estos dos datos debe componer un cadena que tenga la forma como "Un miércoles de marzo" si los valores de los parámetros fueran tres y tres respectivamente.
3. Crea un procedimiento que devuelva, en un parámetro de salida, los números primos hasta un número entero dado (se recibirá como argumento de entrada). Dicho número no puede ser mayor de 20 ni menor de 1 y debes comprobarlo al principio del procedimiento. En caso de no cumplirse devolverás "PARÁMETROS

INCORRECTOS”.

Utiliza WHILE.

4. Crea un procedimiento que reciba dos números entre 1 y 50 y cuente hacia atrás de dos en dos (mostrándolo) desde el mayor al menor (no se sabe si el mayor será el primero o el segundo).

Utiliza LOOP.

5. Crea un procedimiento que reciba dos números entre 10 y 100 y muestre todos los múltiplos de ambos (a la vez) menores de 1000.

Utiliza REPEAT.

6. Elimina los procedimientos creados.

14.4. Funciones almacenadas.

En cuanto a las posibilidades en el interior son (prácticamente) iguales a los procedimientos.

Principales diferencias con los procedimientos:

- Los parámetros siempre son de entrada y devuelve un valor (no se especifica IN al declararlos).
- Se usa RETURNS para declarar el tipo de dato a devolver.
- Se usa RETURN para devolver el valor.

```

CREATE
    [DEFINER = user]
    FUNCTION sp_name ([func_parameter[, ... ]])
    RETURNS type
    [characteristic ... ] routine_body

func_parameter:
    param_name type

type:
    Any valid MySQL data type

characteristic: {
    COMMENT 'string'
    | LANGUAGE SQL
    | [NOT] DETERMINISTIC
    | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
    | SQL SECURITY { DEFINER | INVOKER }
}

routine_body:
    Valid SQL routine statement

```

14.4.1. Características de las rutinas.

Podemos indicar algunas características del procedimiento o de la función. Las opciones disponibles son las siguientes:

- DETERMINISTIC: indica que la función siempre devuelve el mismo resultado cuando se utilizan los mismos parámetros de entrada.
- NOT DETERMINISTIC: la función no siempre devuelve el mismo resultado, aunque se utilicen los mismos parámetros de entrada. Esta es la opción que se selecciona por defecto cuando no se indica una característica de forma explícita.
- CONTAINS SQL: la función contiene sentencias SQL, pero no son de manipulación de datos (ni SELECT, ni INSERT, ni UPDATE). Algunos ejemplos de sentencias SQL que pueden aparecer en este caso son operaciones con variables (Ej: SET @x = 1) o uso de funciones de MySQL (Ej: SELECT NOW();) entre otras. Pero en ningún caso aparecerán sentencias de escritura o lectura de datos.
- NO SQL: la función no contiene sentencias SQL.
- READS SQL DATA: la función no modifica los datos de la base de datos y que

contiene sentencias de lectura de datos, como la sentencia SELECT.

- MODIFIES SQL DATA: la función sí modifica los datos de la base de datos y que contiene sentencias como INSERT, UPDATE o DELETE.

IMPORTANTE: aunque también se pueden indicar al crear un procedimiento, al crear una función es obligatorio indicar algunas, por ello se explican aquí y no antes.

Para poder crear una función en MySQL es necesario indicar al menos una de estas tres características:

- DETERMINISTIC
- NO SQL
- READS SQL DATA

Ejemplo de función:

```
CREATE FUNCTION contar_productos(gama VARCHAR(50))
  RETURNS INT UNSIGNED
  READS SQL DATA
BEGIN
  -- Paso 1. Declaramos una variable local
  DECLARE total INT UNSIGNED;

  -- Paso 2. Contamos los productos
  SET total = (
    SELECT COUNT(*)
    FROM producto
    WHERE producto.gama = gama);

  -- Paso 3. Devolvemos el resultado
  RETURN total;
END $$
```

Y para usarla no utilizamos CALL sino que la incorporamos en SELECT:

```
SELECT contar_productos('Herramientas');
```

Si no se indica al menos una de estas características obtendremos el siguiente mensaje de error:

```
Error Code: 1418. This function has none of DETERMINISTIC, NO SQL, or READS SQL DATA in its declaration and binary logging is enabled (you *might* want to use the less safe log_bin_trust_function_creators variable)
```

En el mensaje ya nos indican que podemos deshabilitar esta restricción. Para ello pondremos la variable global `log_bin_trust_function_creators` a 1.

Es necesario contar con el privilegio SUPER.

Requiere ciertos conocimientos de administración de máquinas Linux así que no entraremos en detalles. Ejecutaremos la sentencia siempre que nos haga falta.

```
SET GLOBAL log_bin_trust_function_creators = 1;
```

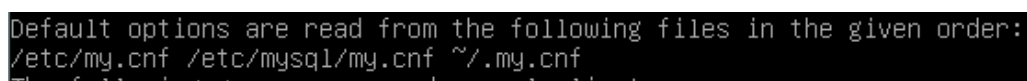
En lugar de configurar la variable global para la sesión, es posible modificarla en el archivo de configuración de MySQL.

14.4.1.1. Extra: ¿Cómo saber dónde está el archivo de configuración de MySQL?

Para encontrarlo, desde nuestra máquina servidor ejecutaremos:

```
mysqld --help --verbose
```

Y tendremos que buscar algo como (Figura 14.1)



```
Default options are read from the following files in the given order:  
/etc/my.cnf /etc/mysql/my.cnf ~/.my.cnf  
The following options are read from the file /etc/my.cnf:
```

Figura 14.1: Ubicación de los archivos de configuración.

La siguiente instrucción lo localiza por nosotros:

```
mysql --help | grep "Default options" -A 1
```

donde `-A 1` indica que nos muestre una línea más a partir de donde aparece el texto buscado.

El proceso es un poco más complejo porque en esos archivos solo se indica que se incluya los archivos que hay en dos directorios. En el archivo `/etc/mysql/conf.d/mysql.cnf`

podríamos añadir la línea

```
log-bin-trust-function-creators = 1
```

y reiniciar el servidor, pero esto son temas administrativos en los que no entraremos.

Ejercicios 14.3

1. Escribe una función que reciba un número entero de entrada y devuelva TRUE si el número es par o FALSE en caso contrario.
2. Crea una función que reciba un parámetros que será una nota numérica entre 0 y 10 y devuelva una cadena de texto con la calificación:

[0,5) = Insuficiente
[5,6) = Aprobado
[6, 7) = Bien
[7, 9) = Notable
[9, 10] = Sobresaliente
En cualquier otro caso la nota no será válida.

3. Vamos a crear un conversor de tiempo. Recibirá un tiempo en segundos y se debe devolver convertido en horas, minutos y segundos. ¿Es mejor usar una función o un procedimiento?
4. Escribe una función que devuelva como salida el número de años que han transcurrido entre dos fechas que se reciben como parámetros de entrada. Por ejemplo, si pasamos como parámetros de entrada las fechas 2018-01-01 y 2008-01-01 la función tiene que devolver que han pasado 10 años.

Para realizar esta función puede hacer uso de las siguientes funciones que nos proporciona MySQL:

- DATEDIFF
 - TRUNCATE
5. Realiza una función que redondee un número real a uno entero sin usar ninguna función excepto FLOOR(), solo con operadores básicos.
 6. Crea una función que compruebe si un número recibido como argumento es primo o no. Determina los límites de los parámetros de entrada y el tipo de la salida.
 7. Crea una función que devuelva el factorial de un número dado.
 8. Escribe una función que reciba una cadena de entrada y devuelva la misma cadena pero sin tildes. La función tendrá que reemplazar todas las vocales que tengan tilde por la misma vocal pero sin tilde. Por ejemplo, si la función recibe como parámetro de entrada la cadena María la función debe devolver la cadena Maria.

9. Crea una función que reciba dos parámetros y genere un número al azar entre un mínimo y máximo indicado. Te servirá la función `RAND` que devuelve un número `[0, 1)`. Para conseguir un número `[i, j]` debes usar
$$\text{FLOOR}(i + \text{RAND}() * (j - i + 1))$$

(Explicar yo por qué si no lo han visto en programación).
10. Crea un procedimiento que calcule si un año que reciba como argumento (entre el 1000 y el 2500) es bisiesto.
11. Escribe un procedimiento con un parámetro para el identificador de empleado, que nos muestre el tiempo que lleva el empleado en la empresa en años, meses y días.
12. Crea un procedimiento que reciba un número entero `[1, 9]` y muestre su tabla de multiplicar.
13. Crea un procedimiento que muestre todas las tablas de multiplicar de los números del 1 al 9 llamando al procedimiento del ejercicio anterior.
14. Elimina los procedimientos y funciones creadas.

Ejercicios 14.4

Ahora vamos a hacer procedimientos con sentencias SQL. Si no se indica lo contrario serán sobre la BdD **jardineria**.

1. Escribe un procedimiento que reciba el nombre de un país como parámetro de entrada y realice una consulta sobre la tabla `cliente` para obtener todos los clientes que existen en la tabla de ese país.
2. Escribe un procedimiento que reciba como parámetro de entrada una forma de pago, que será una cadena de caracteres (Ejemplo: PayPal, Transferencia, etc). Y devuelva como salida el pago de máximo valor realizado para esa forma de pago. Deberá hacer uso de la tabla `pago` de la base de datos `jardineria`.
3. Escribe un procedimiento que reciba como parámetro de entrada una forma de pago, que será una cadena de caracteres (Ejemplo: PayPal, Transferencia, etc). Y devuelva como salida los siguientes valores teniendo en cuenta la forma de pago seleccionada como parámetro de entrada:
 - el pago de máximo valor,
 - el pago de mínimo valor,
 - el valor medio de los pagos realizados,
 - la suma de todos los pagos,
 - el número de pagos realizados para esa forma de pago.

Deberá hacer uso de la tabla pago de la base de datos jardineria.

4. Crea una base de datos llamada procedimientos que contenga una tabla llamada cuadrados. La tabla cuadrados debe tener dos columnas de tipo INT UNSIGNED, una columna llamada numero y otra columna llamada cuadrado.

Una vez creada la base de datos y la tabla deberá crear un procedimiento llamado calcular_cuadrados con las siguientes características:

- El procedimiento recibe un parámetro de entrada llamado tope de tipo INT UNSIGNED con un valor máximo de 30.
- Calculará el valor de los cuadrados de los primeros números naturales hasta el valor introducido como parámetro.
- El valor de cada número y de su cuadrado deberán ser almacenados en la tabla cuadrados que hemos creado previamente.

Ten en cuenta que el procedimiento deberá eliminar el contenido actual de la tabla antes de insertar los nuevos valores de los cuadrados que va a calcular.

Realiza una versión con cada uno de los tres tipos de bucles que hemos visto.

5. En la base de datos llamada procedimientos crea una tabla llamada ejercicio. La tabla debe tener una única columna llamada numero y el tipo de dato de esta columna debe ser INT UNSIGNED.

Una vez creada la base de datos y la tabla deberás crear un procedimiento llamado calcular_numeros con las siguientes características:

- El procedimiento recibe un parámetro de entrada llamado valor_inicial de tipo INT UNSIGNED con valor máximo 200.
- Recibe otro parámetro de entrada llamado salto de tipo INT UNSIGNED con valor máximo 5.
- Deberá almacenar en la tabla ejercicio toda la secuencia de números desde el valor inicial pasado como entrada hasta el 1 de salto en salto. Por ejemplo desde el 47 contando hacia atrás de 3 en 3.

Ten en cuenta que el procedimiento deberá eliminar el contenido actual de la tabla antes de insertar los nuevos valores.

Realiza una versión con cada uno de los tres tipos de bucles que hemos visto.

6. En la base de datos llamada procedimientos crea una tabla llamada pares y otra tabla llamada impares. Las dos tablas deben tener única columna llamada numero y el tipo de dato de esta columna debe ser INT UNSIGNED.

Crea un procedimiento llamado calcular_pares_impares con las siguientes características:

- El procedimiento recibe dos parámetro de entrada llamados min y max de tipo INT UNSIGNED.

- Deberá almacenar en la tabla pares aquellos números pares que existan entre el valor min el valor max.
- Realizará la misma operación para almacenar los números impares en la tabla impares.

Ten en cuenta que el procedimiento deberá eliminar el contenido actual de la tabla antes de insertar los nuevos valores.

7. Escribe una función para la base de datos jardineria que devuelva el número total de productos que hay en la tabla producto.
8. Escribe una función que devuelva el valor medio del precio de los productos de una determinada gama que se recibirá como parámetro.
9. Escribe una función que devuelva el nombre del cliente que haya realizado más pedidos que incluyan productos de una gama que se recibirá como parámetro.

14.5. Manejo de errores.

¿Qué debe hacer el proceso si se produce un error?

Se utilizan manejadores (*handlers*).

```
DECLARE handler_action HANDLER
  FOR condition_value [, condition_value] ...
  statement

handler_action:
CONTINUE
| EXIT
| UNDO

condition_value:
mysql_error_code
| SQLSTATE [VALUE] sqlstate_value
| condition_name
| SQLWARNING
| NOT FOUND
| SQLEXCEPTION
```

Hay tres posibles acciones:

- CONTINUE: la ejecución del proceso continúa.
- EXIT: termina la ejecución del proceso.
- UNDO: no está soportado en MySQL.

Cada manejador se declara asociándolo a un **código de error de MySQL** determinado:

```
DECLARE CONTINUE HANDLER FOR 1051
BEGIN
    -- código del manejador
END;
```

También puede asociarse a un estado **SQLSTATE**:

```
DECLARE EXIT HANDLER FOR SQLSTATE '42S02'
BEGIN
    -- código del manejador
END;
```

NO se deben usar los valores de SQLSTATE que empiezan por '00' porque indican éxito en lugar de errores.

Hay tres palabras reservadas que se pueden usar para englobar varios SQLSTATE:

- SQLWARNING: códigos que comienzan con '01'.
- NOT FOUND: códigos que comienzan con '02'. Se utiliza en cursores (lo veremos más adelante) para detectar cuando no quedan más datos que procesar.
- SQLEXCEPTION: códigos que comienzan con '00', '01' o '02'.

```
DECLARE CONTINUE HANDLER FOR [SQLWARNING | NOT FOUND |
    SQLEXCEPTION]
BEGIN
    -- código del manejador
END;
```

Precedencia:

1. Un manejador de código de error MySQL tiene precedencia sobre uno de SQLSTATE.

2. Uno de SQLSTATE tiene precedencia sobre uno de SQLWARNING, SQLEXCEPTION o NOT FOUND.
3. Uno SQLEXCEPTION tiene precedencia sobre uno de SQLWARNING.

```
DROP DATABASE IF EXISTS test;
CREATE DATABASE test;
USE test;

DROP TABLE IF EXISTS test.t;
CREATE TABLE test.t (s1 INT, PRIMARY KEY (s1));
```

El código 23000 salta si se insertan claves duplicadas.

```
DELIMITER $$

DROP PROCEDURE IF EXISTS handlerdemo;
CREATE PROCEDURE handlerdemo ()
BEGIN
    DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x = 1;
    SET @x = 1;
    INSERT INTO test.t VALUES (1);
    SET @x = 2;
    INSERT INTO test.t VALUES (1);
    SET @x = 3;
END $$

DELIMITER ;
```

```
CALL handlerdemo();
SELECT @x;
```

¿Qué valor devolverá la sentencia SELECT @x? ¿y si usáramos EXIT en lugar de CONTINUE?

En el ejemplo anterior, el manejador solo necesita una sentencia. Si fuera necesario utilizar más se usaría BEGIN ... END:

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
    ROLLBACK;
    SELECT 'Se ha producido un error que ha devuelto al Bdd al
        estado inicial';
END;
```

14.5.1. Un ejemplo más completo.

Creamos una tabla que sería derivada de la relación N:M entre Proveedor y Producto. Por simplificar el ejemplo no se crean dichas tablas ni las claves ajenas.

```
CREATE TABLE ProveedorProducto (
    proveedorId INT,
    productoId INT,
    PRIMARY KEY (proveedorId , productoId)
);
```

Creamos un procedimiento para insertar valores en la tabla:

```
DELIMITER $$

CREATE PROCEDURE InsertProveedorProducto(
    IN inproveedorId INT,
    IN inProductoId INT
)
BEGIN
    -- sale si hay una clave duplicada.
    DECLARE EXIT HANDLER FOR 1062
    BEGIN
        SELECT CONCAT('PK duplicada: (',inproveedorId,',',
            ,inProductoId,')') AS Mensaje;
    END;

    INSERT INTO ProveedorProducto(proveedorId,productoId)
        VALUES(inproveedorId,inProductoId);

    SELECT COUNT(*)
        FROM ProveedorProducto
        WHERE proveedorId = inproveedorId;

END$$

DELIMITER ;
```

Y realizamos varias inserciones, la última de una clave repetida:

```
CALL InsertProveedorProducto(1,1);
CALL InsertProveedorProducto(1,2);
CALL InsertProveedorProducto(1,3);
-- Repetimos la última.
CALL InsertProveedorProducto(1,3);
```

¿Qué aparecerá por pantalla? ¿Se ejecuta SELECT COUNT(*) ... ? ¿por qué?

14.5.2. Manejadores y etiquetas.

Un manejador no puede utilizar una etiqueta del bloque principal:


```
CREATE PROCEDURE p ()
BEGIN
  DECLARE i INT DEFAULT 3;
  retry:
    REPEAT
      BEGIN
        DECLARE CONTINUE HANDLER FOR SQLWARNING
        BEGIN
          ITERATE retry; # no válido
        END;
        IF i < 0 THEN
          LEAVE retry; # válido
        END IF;
        SET i = i - 1;
      END;
    UNTIL FALSE END REPEAT;
END;
```

Para ello podemos usar *flags*:

```
DELIMITER $$

CREATE PROCEDURE p ( )
BEGIN
    DECLARE i INT DEFAULT 3;
    DECLARE done INT DEFAULT FALSE;
    retry:
        REPEAT
            BEGIN
                DECLARE CONTINUE HANDLER FOR SQLWARNING
                BEGIN
                    SET done = TRUE;
                END;
                IF done OR i < 0 THEN
                    LEAVE retry;
                END IF;
                SET i = i - 1;
            END;
        UNTIL FALSE END REPEAT;
END$$

DELIMITER ;
```

Si en algún caso es necesario ignorar una situación puedes declarar un manejador con el bloque vacío:

```
DECLARE CONTINUE HANDLER FOR SQLWARNING BEGIN END;
```

14.6. Lanzar errores.

¿Qué sucede si en nuestro proceso se produce una situación de error que no lo es para el SGBD? En Java lanzaríamos una excepción, aquí usamos SIGNAL o RESIGNAL.

Su uso tiene más posibilidades de las que veremos aquí. Puedes consultar la documentación:

- **SIGNAL**: lanza un error. Similar a lanzar una excepción en Java.
- **RESIGNAL**: es similar a propagar una excepción en Java pero pudiendo modificar parte de la información que lleva.

Con SIGNAL informamos de un error asociándolo a alguno de los códigos que hemos visto. Tiene sentido al realizar algún tipo de comprobación en nuestro código.

Un ejemplo:

```
CREATE PROCEDURE modifNomDepartamento(  
    IN id_departamento INT,  
    IN nombre_departamento VARCHAR(30))  
BEGIN  
    DECLARE num INT;  
  
    UPDATE departamentos D  
    SET D.nombre_departamento = nombre_departamento  
    WHERE D.id_departamento = id_departamento;  
  
    SELECT ROW_COUNT() INTO num;  
  
    IF(num ≠ 1) THEN  
        SIGNAL SQLSTATE '45000'  
        SET MESSAGE_TEXT = 'No existe el departamento';  
    END IF;  
END $$
```

RESIGNAL se utiliza de manera similar con un par de diferencias:

- Tiene que usarse obligatoriamente dentro del manejador de un error o aviso mientras que SIGNAL puede usarse en otros lugares también.
- Se pueden omitir todos atributos, ya que si no los definimos, usará los del error o aviso que estemos "manejando".

Podemos refactorizar el procedimiento que ya vimos para usar RESIGNAL. Ten en cuenta que solo cambiamos el texto pero no el código del error en este ejemplo.

```
CREATE PROCEDURE InsertProveedorProducto(  
    IN inproveedorId INT,  
    IN inProductoId INT  
)  
BEGIN  
    -- sale si hay una clave duplicada.  
    DECLARE EXIT HANDLER FOR 1062  
    BEGIN  
        DECLARE mensaje VARCHAR(30);  
        SET mensaje = CONCAT('PK duplicada:  
            (',inproveedorId,',',inProductoId,')');  
        RESIGNAL SET MESSAGE_TEXT = mensaje;  
    END;  
  
    INSERT INTO ProveedorProducto(proveedorId,productoId)  
        VALUES(inproveedorId,inProductoId);  
  
    SELECT COUNT(*)  
    FROM ProveedorProducto  
    WHERE proveedorId = inproveedorId;  
END$$
```

No vamos a ver otros pero, además de modificar el texto, podríamos hacerlo con otros elementos (tanto para SIGNAL como RESIGNAL):

- CLASS_ORIGIN
- SUBCLASS_ORIGIN
- CONSTRAINT_CATALOG
- CONSTRAINT_SCHEMA
- CONSTRAINT_NAME
- CATALOG_NAME
- SCHEMA_NAME
- TABLE_NAME
- COLUMN_NAME
- CURSOR_NAME
- MESSAGE_TEXT
- MYSQL_ERRNO

Para mostrar los errores podemos usar

```
SHOW ERRORS;
```

Ejercicios 14.5

1. Sustituyendo el manejador del procedimiento InsertProveedorProducto por los tres siguientes:

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
    SELECT 'SQLEXCEPTION' Mensaje;
DECLARE EXIT HANDLER FOR 1062
    SELECT 'Código MySQL' Mensaje;
DECLARE EXIT HANDLER FOR SQLSTATE '23000'
    SELECT 'SQLSTATE' Mensaje;
```

Llama al procedimiento insertando una clave duplicada.

¿cuál de los manejadores se aplica? ¿por qué?

2. Crea una base de datos llamada test que contenga una tabla llamada alumno. La tabla debe tener cuatro columnas:

- id: entero sin signo (clave primaria).
- nombre: cadena de 50 caracteres.
- apellido1: cadena de 50 caracteres.
- apellido2: cadena de 50 caracteres.

Una vez creada la base de datos y la tabla deberá crear un procedimiento llamado insertar_alumno con las siguientes características:

- Recibirá cuatro parámetros de entrada (id, nombre, apellido1, apellido2) y los insertará en la tabla alumno.
- Devolverá como salida un parámetro llamado error que tendrá un valor igual a 0 si la operación se ha podido realizar con éxito y un valor igual a 1 en caso contrario.
- Deberá manejar los errores que puedan ocurrir cuando se intenta insertar una fila que contiene una clave primaria repetida.

3. Escribe un procedimiento que inserte un nuevo departamento con los parámetros que se reciban, incluido el id.

¿Qué errores pueden darse al realizar la operación?

Introduce gestión de errores para controlarlos, decidiendo qué hacer en cada

caso.

4. Realiza un procedimiento para modificar el salario de un empleado. Se recibirá el identificador del empleado y el nuevo salario.
¿Qué sucede si nos pasan un tipo de dato incorrecto en el salario? ¿es necesario controlarlo dentro del procedimiento?
¿Puedes establecer un control de errores si el identificador de empleado no existe? La función `ROW_COUNT()` te permite saber el número de filas afectadas por la última operación (distinta de SELECT).
5. Establece control de errores en el ejercicio 14.2.2. Controla que los parámetros que nos pasan tengan valores válidos y lanza un mensaje en caso contrario.
6. Crea un procedimiento para introducir un nuevo departamento en la tabla departamentos de la Bdd empleados. Ten en cuenta los posibles errores que puedan producirse en la inserción y utiliza RESIGNAL para enviar un mensaje adecuado en español en cada caso.

14.7. Transacciones en procesos.

La ventaja es que podemos determinar si hacer COMMIT o ROLLBACK.

No hay nada de contenido nuevo, solo indicar cómo integrarlo así que incluyo aquí dos modelos genéricos:

Si queremos deshacer las operaciones si se produce un error o un aviso:

```
DELIMITER $$
CREATE PROCEDURE transaccion_en_mysql()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        -- ERROR
        ROLLBACK;
    END;

    DECLARE EXIT HANDLER FOR SQLWARNING
    BEGIN
        -- WARNING
        ROLLBACK;
    END;

    START TRANSACTION;
    -- Sentencias SQL
    COMMIT;
END $$
```

En lugar de tener un manejador para cada tipo de error, podemos tener uno común para todos los casos.

```
DELIMITER $$
CREATE PROCEDURE transaccion_en_mysql()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING
    BEGIN
        -- ERROR, WARNING
        ROLLBACK;
    END;

    START TRANSACTION;
    -- Sentencias SQL
    COMMIT;
END $$
```

También podemos usar **sentencias condicionales** para decidir si realizar un COMMIT o ROLLBACK.

Ejercicios 14.6

1. Crea una tabla cuentas con las siguientes columnas:

- id_cuenta: entero sin signo (clave primaria).
- saldo: real sin signo.

Inserta varias filas.

Realiza un procedimiento que reciba dos identificadores de cuenta y una cantidad. Debe realizar una transferencia de la cuenta del primer identificador a la segunda.

Realiza una gestión de errores completa y utiliza transacciones para asegurar el correcto funcionamiento de la operación.

2. Crea una base de datos llamada cine que contenga dos tablas con las siguientes columnas.

Tabla cuentas: como en el ejercicio anterior.

Tabla entradas:

- id_butaca: entero sin signo (clave primaria).
- nif: cadena de 9 caracteres.

Una vez creada la base de datos y las tablas deberá crear un procedimiento llamado comprar_entrada con las siguientes características:

- Recibe 3 parámetros de entrada (nif, id_cuenta, id_butaca).
- Devolverá como salida un parámetro llamado error que tendrá un valor igual a 0 si la compra de la entrada se ha podido realizar con éxito y un valor igual a 1 en caso contrario.

El procedimiento de compra debe realizar los siguientes pasos:

- a) Inicia una transacción.
- b) Actualiza la columna saldo de la tabla cuentas cobrando 5 euros a la cuenta con el id_cuenta adecuado.
- c) Inserta una una fila en la tabla entradas indicando la butaca (id_butaca) que acaba de comprar el usuario (nif).
- d) Comprueba si ha ocurrido algún error en las operaciones anteriores. En caso negativo aplica un COMMIT a la transacción y si ha ocurrido algún error aplica un ROLLBACK.

Deberá manejar los siguientes errores que puedan ocurrir durante el proceso:

- ERROR 1264 (Out of range value)
- ERROR 1062 (Duplicate entry for PRIMARY KEY)

3. ¿Qué ocurre cuando intentamos comprar una entrada y le pasamos como parámetro un número de cuenta que no existe en la tabla cuentas? ¿Ocurre algún

error o podemos comprar la entrada?

En caso de que se produzca algún error, ¿cómo podríamos resolverlo?.

14.8. Cursores.

¿Cómo podemos manejar un *resultset*¹ dentro de un proceso? Usando cursores.

Los cursores de MySQL tienen tres características:

- Solo lectura.
- Sin desplazamiento: solo se pueden leer los datos en el orden en el que fueron devueltos, empezando desde el primero, sin posibilidad de saltar ni volver hacia atrás.
- Sensibles a cambios: trabajan sobre los datos reales (otros SGBD permiten trabajar sobre una copia) por lo que cualquier cambio realizado en ellos, se reflejará inmediatamente en nuestro cursor.

Las operaciones que podemos hacer con los cursores son las siguientes:

DECLARE

El primer paso que tenemos que hacer para trabajar con cursores es declararlo. La sintaxis para declarar un cursor es:

```
DECLARE cursor_name CURSOR FOR select_statement;
```

Un cursor siempre tiene que ir asociado a una sentencia SELECT.

La declaración de cursores siempre debe realizarse **después** de la declaración de variables.

OPEN

Una vez que hemos declarado un cursor tenemos que abrirlo con OPEN.

¹Resultado de una sentencia SELECT.

```
OPEN cursor_name;
```

FETCH

Cuando el cursor está abierto podemos ir obteniendo cada una de las filas con FETCH. La sintaxis es la siguiente:

```
FETCH [[NEXT] FROM] cursor_name INTO var_name [, var_name] ... ;
```

Al estar recorriendo un cursor, si no quedan filas por recorrer se lanza el error NOT FOUND, que se corresponde con el valor SQLSTATE '02000'. Por eso cuando estemos trabajando con cursores será necesario declarar un manejador para gestionar este error.

```
DECLARE CONTINUE HANDLER FOR NOT FOUND ... ;
```

Por ejemplo:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;
```

CLOSE

Cuando hemos terminado de trabajar con un cursor tenemos que cerrarlo.

```
CLOSE cursor_name;
```

14.8.1. Ejemplos.

14.8.1.1. Ejemplo 1.

Sobre la BdD de empleados.

Creemos un procedimiento para obtener una lista con los correos de todos los empleados:

```
DELIMITER $$
CREATE PROCEDURE listaCorreos (
    INOUT lista varchar(4000)
)
BEGIN
    DECLARE fin INTEGER DEFAULT 0;
    DECLARE correo varchar(100) DEFAULT "";

    -- Declaramos el cursor
    DECLARE curEmail CURSOR FOR
        SELECT email FROM empleados;

    -- Declaramos el manejador para NOT FOUND
    DECLARE CONTINUE HANDLER
        FOR NOT FOUND SET fin = 1;

    -- Abrimos el cursor
    OPEN curEmail;

    getEmail: LOOP
        FETCH curEmail INTO correo;
        IF fin = 1 THEN
            LEAVE getEmail;
        END IF;
        SET lista = CONCAT(correo, "; ", lista);
    END LOOP getEmail;

    -- Cerramos el cursor
    CLOSE curEmail;

END$$
DELIMITER ;
```

Y lo probamos:

```
SET @emailLista = "";
CALL listaCorreos(@emailLista);
SELECT @emailLista;
```

14.8.1.2. Ejemplo 2.

Suponiendo la siguiente BdD:

```
DROP DATABASE IF EXISTS prueba;
CREATE DATABASE prueba;
USE prueba;

CREATE TABLE t1 (
    id INT UNSIGNED PRIMARY KEY,
    datos VARCHAR(16)
);

CREATE TABLE t2 (
    cont INT UNSIGNED,
    letra VARCHAR(16)
);

CREATE TABLE t3 (
    texto VARCHAR(16),
    num INT UNSIGNED
);

INSERT INTO t1 VALUES (1, 'AAAA');
INSERT INTO t1 VALUES (20, 'BBBB');

INSERT INTO t2 VALUES (10, 'Diez');
INSERT INTO t2 VALUES (2, 'Dos');
```

Podríamos tener el siguiente procedimiento que utiliza dos cursores:

```
DROP PROCEDURE IF EXISTS curdemo;

DELIMITER $$

CREATE PROCEDURE curdemo()
BEGIN
    DECLARE fin INT DEFAULT FALSE;
    DECLARE informacion, letras CHAR(16);
    DECLARE identificador, contador INT;
    DECLARE curT1 CURSOR FOR SELECT id, datos FROM prueba.t1;
    DECLARE curT2 CURSOR FOR SELECT cont, letra FROM prueba.t2;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin = TRUE;

    OPEN curT1;
    OPEN curT2;

    bucle: LOOP
        FETCH curT1 INTO identificador, informacion;
        FETCH curT2 INTO contador, letras;
        IF fin THEN
            LEAVE bucle;
        END IF;
        IF identificador < contador THEN
            INSERT INTO prueba.t3 VALUES (informacion, identificador);
        ELSE
            INSERT INTO prueba.t3 VALUES (letras, contador);
        END IF;
    END LOOP;

    CLOSE curT1;
    CLOSE curT2;
END $$

DELIMITER ;
```

Para probarlo:

```
CALL curdemo();
SELECT * FROM t3;
```

Ejercicios 14.7

1. Escribe un procedimiento que suba un 10 % el salario a los empleados con comisión y que ganen menos de 9000 . Para cada empleado se mostrará por pantalla el código de empleado, nombre, apellido, salario anterior y final. Utiliza un cursor. La transacción no puede quedarse a medias. Si por cualquier razón no es posible actualizar todos estos salarios, debe deshacerse el trabajo a la situación inicial.
2. Escribe un procedimiento que reciba dos parámetros (número de departamento, salario). Deberá crearse un cursor al que se le pasarán estos parámetros y que mostrará los datos de los empleados que pertenezcan al departamento y con el número de hijos indicados. Al final se indicará el número de empleados obtenidos.
3. Escribe las sentencias SQL necesarias para crear una base de datos llamada `test`, una tabla llamada `alumnos` y cuatro sentencias de inserción para inicializar la tabla. La tabla `alumnos` está formada por las siguientes columnas:

- `id` (entero sin signo y clave primaria)
- `nombre` (cadena de caracteres)
- `apellido1` (cadena de caracteres)
- `apellido2` (cadena de caracteres)
- `fecha_nacimiento` (fecha)

Una vez creada la tabla se decide añadir una nueva columna a la tabla llamada `edad` que será un valor calculado a partir de la columna `fecha_nacimiento`. Escribe la sentencia SQL necesaria para modificar la tabla y añadir la nueva columna.

Escribe una función llamada `calcular_edad` que reciba una fecha y devuelva el número de años (entero) que han pasado desde la fecha actual hasta la fecha pasada como parámetro.

Ahora escribe un procedimiento que permita calcular la edad de todos los alumnos que ya existen en la tabla. Para esto será necesario crear un procedimiento llamado `actualizar_columna_edad` que calcule la edad de cada alumno y actualice la tabla. Este procedimiento hará uso de la función `calcular_edad` que hemos creado en el paso anterior.

4. Modifica la tabla `alumnos` de los ejercicios anteriores para añadir una nueva columna `email`. Una vez que hemos modificado la tabla necesitamos asignarle una dirección de correo electrónico de forma automática.

Escribe un procedimiento llamado `crear_email` que dados los parámetros de entrada: `nombre`, `apellido1`, `apellido2` y `dominio`, cree una dirección de email y la devuelva como salida.

La dirección de correo electrónico tendrá el siguiente formato:

- El primer carácter del parámetro nombre.
- Los tres primeros caracteres del parámetro apellido1.
- Los tres primeros caracteres del parámetro apellido2.
- El carácter @.
- El dominio pasado como parámetro.

Ahora escriba un procedimiento que permita crear un email para todos los alumnos que ya existen en la tabla. Para esto será necesario crear un procedimiento llamado `actualizar_columna_email` que actualice la columna `email` de la tabla `alumnos`. Este procedimiento hará uso del procedimiento `crear_email` que hemos creado en el paso anterior.

5. Escribe un procedimiento llamado `crear_lista_emails_alumnos` que devuelva la lista de emails de la tabla `alumnos` separados por un punto y coma. Ejemplo: `juan@iescelia.org;maria@iescelia.org;pepe@iescelia.org;lucia@iescelia.org`.

14.9. Triggers.

Un *trigger*² es un objeto de la base de datos que está asociado con una tabla y que se activa cuando ocurre un evento sobre la tabla.

Nos permiten establecer comprobaciones más complejas sobre los datos de una tabla. Si es posible, es preferible usar `CHECK` en la creación de la tabla.

El estándar SQL define dos tipos:

- A nivel de fila: cuando se actúa sobre una fila. Si la operación afecta a 100 filas, se invoca el *trigger* 100 veces.
- A nivel de sentencia: el *trigger* se invoca una sola vez independientemente del número de filas afectadas.

MySQL solo soporta los *triggers* a nivel de fila.

²disparador, gatillo.

```
CREATE
  [DEFINER = { user | CURRENT_USER }]
  TRIGGER trigger_name
  trigger_time trigger_event
  ON tbl_name FOR EACH ROW
  [trigger_order]
  trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

Los eventos que podemos monitorizar son:

- INSERT: el *trigger* se activa cuando se inserta una nueva fila sobre la tabla asociada.
- UPDATE: el *trigger* se activa cuando se actualiza algún valor de una fila sobre la tabla asociada.
- DELETE: el *trigger* se activa cuando se elimina una fila sobre la tabla asociada.

Para diferenciar los valores de las columnas antes y después de que se inicie la sentencia DML se utilizan las palabras reservadas NEW y OLD:

- OLD: no existe en *triggers* INSERT.
- NEW: no existe en *triggers* DELETE.

Para consultar los triggers que hay en la BdD actual:

```
SHOW TRIGGERS;
```

14.9.1. Ejemplo.

Creamos una base de datos llamada test que contenga una tabla llamada alumnos con las siguientes columnas:

- id (entero sin signo)

- nombre (cadena de caracteres)
- apellido1 (cadena de caracteres)
- apellido2 (cadena de caracteres)
- nota (número real)

```
DROP DATABASE IF EXISTS test;
CREATE DATABASE test;
USE test;

CREATE TABLE alumnos (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(50) NOT NULL,
  apellido1 VARCHAR(50) NOT NULL,
  apellido2 VARCHAR(50),
  nota FLOAT
);
```

Vamos a escribir dos *triggers* con las siguientes características:

Trigger 1: trigger_check_nota_before_insert

- Se ejecuta sobre la tabla alumnos.
- Se ejecuta antes de una operación de inserción.
- Si el nuevo valor de la nota que se quiere insertar es negativo, se guarda como 0.
- Si el nuevo valor de la nota que se quiere insertar es mayor que 10, se guarda como 10.

```
DELIMITER $$
DROP TRIGGER IF EXISTS trigger_check_nota_before_insert$$
CREATE TRIGGER trigger_check_nota_before_insert
    BEFORE INSERT
    ON alumnos FOR EACH ROW
BEGIN
    IF NEW.nota < 0 THEN
        set NEW.nota = 0;
    ELSEIF NEW.nota > 10 THEN
        set NEW.nota = 10;
    END IF;
END $$

DELIMITER ;
```

Trigger 2 : trigger_check_nota_before_update

- Se ejecuta sobre la tabla alumnos.
- Se ejecuta antes de una operación de actualización.
- Si el nuevo valor de la nota que se quiere actualizar es negativo, se mantiene el valor anterior.
- Si el nuevo valor de la nota que se quiere actualizar es mayor que 10, se mantiene el valor anterior.

```
CREATE TRIGGER trigger_check_nota_before_update
    BEFORE UPDATE
    ON alumnos FOR EACH ROW
BEGIN
    IF NEW.nota < 0 THEN
        set NEW.nota = OLD.nota;
    ELSEIF NEW.nota > 10 THEN
        set NEW.nota = OLD.nota;
    END IF;
END $$
```

Una vez creados los *triggers* probamos varias sentencias de inserción y actualización sobre la tabla *alumnos* para verificar que los *triggers* se están ejecutando correctamente.

```
INSERT INTO alumnos VALUES (1, 'Pepe', 'López', 'López', -1);
INSERT INTO alumnos VALUES (2, 'María', 'Sánchez', 'Sánchez', 11);
INSERT INTO alumnos VALUES (3, 'Juan', 'Pérez', 'Pérez', 8.5);

SELECT * FROM alumnos;
```

```
UPDATE alumnos SET nota = 14 WHERE id = 1;
UPDATE alumnos SET nota = -4 WHERE id = 2;
UPDATE alumnos SET nota = 9.5 WHERE id = 3;

SELECT * FROM alumnos;
```

14.9.2. Otros ejemplos.

14.9.2.1. Pagos.

Sobre una tabla de pagos:

```
CREATE TABLE billings (
    billingNo INT AUTO_INCREMENT,
    customerNo INT,
    billingDate DATE,
    amount DEC(10, 2),
    PRIMARY KEY (billingNo)
);
```

Creamos un trigger que no permita actualizar la cuantía en un valor superior a 10 veces el anterior:

```
DELIMITER $$
CREATE TRIGGER before_billing_update
  BEFORE UPDATE
  ON billings FOR EACH ROW
BEGIN
  IF new.amount > old.amount * 10 THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'New amount cannot be 10 times greater
                        than the current amount.';
  END IF;
END$$
DELIMITER ;
```

14.9.2.2. Centros de trabajo. BEFORE INSERT.

Sobre las tablas:

```
DROP TABLE IF EXISTS WorkCenters;

CREATE TABLE WorkCenters (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  capacity INT NOT NULL
);
```

y

```
DROP TABLE IF EXISTS WorkCenterStats;

CREATE TABLE WorkCenterStats(
  totalCapacity INT NOT NULL
);
```

Creamos un trigger que si existía alguna fila, aumenta la capacidad y en caso contrario inserta la fila con la capacidad indicada:

```
DELIMITER $$

CREATE TRIGGER before_workcenters_insert
  BEFORE INSERT
  ON WorkCenters FOR EACH ROW
BEGIN
  DECLARE rowcount INT;

  SELECT COUNT(*)
  INTO rowcount
  FROM WorkCenterStats;

  IF rowcount > 0 THEN
    UPDATE WorkCenterStats
    SET totalCapacity = totalCapacity + new.capacity;
  ELSE
    INSERT INTO WorkCenterStats(totalCapacity)
    VALUES(new.capacity);
  END IF;

END $$

DELIMITER ;
```

Podemos probar:

```
INSERT INTO WorkCenters(name, capacity)
VALUES('Mold Machine',100);

SELECT * FROM WorkCenterStats;
```

```
INSERT INTO WorkCenters(name, capacity)
VALUES('Packing',200);

SELECT * FROM WorkCenterStats;
```

14.9.2.3. Notas (recordatorios). AFTER INSERT.

Teniendo las tablas:

```
DROP TABLE IF EXISTS members;

CREATE TABLE members (
  id INT AUTO_INCREMENT,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(255),
  birthDate DATE,
  PRIMARY KEY (id)
);
```

y

```
DROP TABLE IF EXISTS reminders;

CREATE TABLE reminders (
  id INT AUTO_INCREMENT,
  memberId INT,
  message VARCHAR(255) NOT NULL,
  PRIMARY KEY (id , memberId)
);
```

Creamos un trigger que compruebe si la fecha insertada es NULL para insertar un recordatorio de que el usuario debe actualizarla:

```
DELIMITER $$

CREATE TRIGGER after_members_insert
  AFTER INSERT
  ON members FOR EACH ROW
BEGIN
  IF NEW.birthDate IS NULL THEN
    INSERT INTO reminders(memberId, message)
      VALUES(new.id, CONCAT('Hi ', NEW.name, ', please update your
        date of birth.'));
  END IF;
END$$

DELIMITER ;
```

Lo probamos:

```
INSERT INTO members(name, email, birthDate)
VALUES
  ('John Doe', 'john.doe@example.com', NULL),
  ('Jane Doe', 'jane.doe@example.com', '2000-01-01');

SELECT * FROM members;

SELECT * FROM reminders;
```

14.9.2.4. Ventas. BEFORE UPDATE.

Con la tabla:

```
DROP TABLE IF EXISTS sales;

CREATE TABLE sales (
  id INT AUTO_INCREMENT,
  product VARCHAR(100) NOT NULL,
  quantity INT NOT NULL DEFAULT 0,
  fiscalYear SMALLINT NOT NULL,
  fiscalMonth TINYINT NOT NULL,
  CHECK(fiscalMonth ≥ 1 AND fiscalMonth ≤ 12),
  CHECK(fiscalYear BETWEEN 2000 and 2050),
  CHECK (quantity ≥ 0),
  UNIQUE(product, fiscalYear, fiscalMonth),
  PRIMARY KEY(id)
);
```

Insertamos algunos datos:

```
INSERT INTO sales(product, quantity, fiscalYear, fiscalMonth)
VALUES
  ('2003 Harley-Davidson Eagle Drag Bike', 120, 2020, 1),
  ('1969 Corvair Monza', 150, 2020, 1),
  ('1970 Plymouth Hemi Cuda', 200, 2020, 1);
```

Creamos un *trigger* que compruebe que no se puede actualizar la cantidad a una que sea tres veces superior que la anterior:


```
DELIMITER $$

CREATE TRIGGER before_sales_update
  BEFORE UPDATE
  ON sales FOR EACH ROW
BEGIN
  DECLARE errorMessage VARCHAR(255);
  SET errorMessage = CONCAT('The new quantity ',
    NEW.quantity,
    ' cannot be 3 times greater than the current quantity ',
    OLD.quantity);

  IF new.quantity > old.quantity * 3 THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = errorMessage;
  END IF;
END $$

DELIMITER ;
```

Y lo probamos:

```
UPDATE sales
  SET quantity = 150
  WHERE id = 1;
```

```
UPDATE sales
  SET quantity = 500
  WHERE id = 1;
```

Lanza el error:

```
Error Code: 1644. The new quantity 500 cannot be 3 times greater
than the current quantity 150
```

Se puede consultar con

```
SHOW ERRORS;
```

14.9.2.5. Ventas. AFTER UPDATE.

Con la tabla:

```
DROP TABLE IF EXISTS Sales;

CREATE TABLE Sales (
  id INT AUTO_INCREMENT,
  product VARCHAR(100) NOT NULL,
  quantity INT NOT NULL DEFAULT 0,
  fiscalYear SMALLINT NOT NULL,
  fiscalMonth TINYINT NOT NULL,
  CHECK(fiscalMonth ≥ 1 AND fiscalMonth ≤ 12),
  CHECK(fiscalYear BETWEEN 2000 and 2050),
  CHECK (quantity ≥ 0),
  UNIQUE(product, fiscalYear, fiscalMonth),
  PRIMARY KEY(id)
);
```

Con datos:

```
INSERT INTO Sales(product, quantity, fiscalYear, fiscalMonth)
VALUES
  ('2001 Ferrari Enzo',140, 2021,1),
  ('1998 Chrysler Plymouth Prowler', 110,2021,1),
  ('1913 Ford Model T Speedster', 120,2021,1);
```

Creamos otra tabla para registrar los cambios en cantidad de la tabla anterior:

```
DROP TABLE IF EXISTS SalesChanges;

CREATE TABLE SalesChanges (
    id INT AUTO_INCREMENT PRIMARY KEY,
    salesId INT,
    beforeQuantity INT,
    afterQuantity INT,
    changedAt TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

Creamos un *trigger* que lo gestione:

```
DELIMITER $$

CREATE TRIGGER after_sales_update
AFTER UPDATE
ON sales FOR EACH ROW
BEGIN
    IF OLD.quantity <> new.quantity THEN
        INSERT INTO SalesChanges(salesId,beforeQuantity,
            afterQuantity)
            VALUES(old.id, old.quantity, new.quantity);
    END IF;
END$$

DELIMITER ;
```

Y lo probamos:

```
UPDATE Sales
SET quantity = 350
WHERE id = 1;

SELECT * FROM SalesChanges;
```

```
UPDATE Sales
SET quantity = CAST(quantity * 1.1 AS UNSIGNED);

SELECT * FROM SalesChanges;
```

14.9.2.6. Registro de salarios. BEFORE DELETE.

Con las tablas:

```
DROP TABLE IF EXISTS Salaries;

CREATE TABLE Salaries (
    employeeNumber INT PRIMARY KEY,
    validFrom DATE NOT NULL,
    amount DEC(12 , 2 ) NOT NULL DEFAULT 0
);
```

```
INSERT INTO salaries(employeeNumber,validFrom,amount)
VALUES
    (1002, '2000-01-01', 50000),
    (1056, '2000-01-01', 60000),
    (1076, '2000-01-01', 70000);
```

Y otra en la que queremos registrar los datos que se borren:

```
DROP TABLE IF EXISTS SalaryArchives;

CREATE TABLE SalaryArchives (
    id INT PRIMARY KEY AUTO_INCREMENT,
    employeeNumber INT PRIMARY KEY,
    validFrom DATE NOT NULL,
    amount DEC(12 , 2 ) NOT NULL DEFAULT 0,
    deletedAt TIMESTAMP DEFAULT NOW()
);
```

Creamos el *trigger* que lo haga:

```
DELIMITER $$

CREATE TRIGGER before_salaries_delete
  BEFORE DELETE
  ON salaries FOR EACH ROW
BEGIN
  INSERT INTO SalaryArchives(employeeNumber,validFrom,amount)
  VALUES(OLD.employeeNumber,OLD.validFrom,OLD.amount);
END$$

DELIMITER ;
```

Para probarlo:

```
DELETE FROM salaries
  WHERE employeeNumber = 1002;

SELECT * FROM SalaryArchives;
```

```
DELETE FROM salaries;

SELECT * FROM SalaryArchives;
```

14.9.2.7. Presupuesto salarial. AFTER DELETE.

Con la tabla:

```
DROP TABLE IF EXISTS Salaries;

CREATE TABLE Salaries (
  employeeNumber INT PRIMARY KEY,
  salary DECIMAL(10,2) NOT NULL DEFAULT 0
);
```

```
INSERT INTO Salaries(employeeNumber,salary)
VALUES
    (1002,5000),
    (1056,7000),
    (1076,8000);
```

Creamos otra para almacenar el presupuesto salarial que será la suma de los salarios:

```
DROP TABLE IF EXISTS SalaryBudgets;

CREATE TABLE SalaryBudgets(
    total DECIMAL(15,2) NOT NULL
);
```

```
INSERT INTO SalaryBudgets(total)
SELECT SUM(salary)
FROM Salaries;
```

Lo comprobamos:

```
SELECT * FROM SalaryBudgets;
```

Creamos el trigger para actualizar:

```
CREATE TRIGGER after_salaries_delete
AFTER DELETE
ON Salaries FOR EACH ROW
BEGIN
    UPDATE SalaryBudgets
    SET total = total - old.salary;
END$$
```

Para probarlo:

```
DELETE FROM Salaries
  WHERE employeeNumber = 1002;

SELECT * FROM SalaryBudgets;
```

```
DELETE FROM Salaries;

SELECT * FROM SalaryBudgets;
```

14.9.3. Consultar *triggers*.

Podemos hacerlo de manera análoga a cómo hemos consultado procedimientos:

```
SHOW TRIGGERS
[FROM | IN] database_name]
[LIKE 'pattern' | WHERE search_condition];
```

Por ejemplo:

```
SHOW TRIGGERS;
```

```
SHOW TRIGGERS
FROM Conciertos;
```

```
SHOW TRIGGERS
FROM empleados
WHERE table = 'departamentos';
```

14.9.4. Orden de los *triggers*.

Este punto se puede complicar porque puede haber varios *triggers* actuando sobre los mismos datos en el mismo momento (tiempo: BEFORE o AFTER).

Para gestionarlo, existe la posibilidad de establecer un orden con PRECEDES o FOLLOWS. No entraremos en detalles, simplemente pongo un ejemplo.

En este caso, el *trigger* `ins_transaction` se ejecuta a la vez que `ins_sum` en la misma tabla (ambos `BEFORE INSERT ON account`).

Observa también que al tener una única instrucción no es necesario usar `BEGIN` ni `END`.

```
CREATE TRIGGER ins_transaction
  BEFORE INSERT ON account
  FOR EACH ROW PRECEDES ins_sum
SET
  @deposits = @deposits + IF(NEW.amount>0,NEW.amount,0),
  @withdrawals = @withdrawals + IF(NEW.amount<0,-NEW.amount,0);
```

Si te interesa el tema, puedes ver una explicación [aquí](#).

14.10. Programar (en el tiempo) procesos.

Este apartado no se incluye en el temario de DAM/DAW pero lo nombro para que sepáis que se puede hacer.

Podemos programar procesos para que se realicen en algún [momento](#).

Ejercicios 14.8

1. Crea una base de datos llamada `test` que contenga una tabla llamada `alumnos` con las siguientes columnas:

- `id` (entero sin signo)
- `nombre` (cadena de caracteres)
- `apellido1` (cadena de caracteres)
- `apellido2` (cadena de caracteres)
- `nota` (número real)
- `nota_texto` (cadena de caracteres)

Una vez creada la tabla escriba dos *triggers* con las siguientes características:

Trigger 1: `trigger_check_nota_before_insert`:

- Se ejecuta sobre la tabla `alumnos`.
- Se ejecuta antes de una operación de inserción.
- Si el nuevo valor de la nota que se quiere insertar es negativo, se guarda como 0.
- Si el nuevo valor de la nota que se quiere insertar es mayor que 10, se guarda

como 10.

- Insertará en `nota_texto` el número en letra (1 → 'UNO', etc.).

Trigger 2: `trigger_check_nota_before_update`:

- Se ejecuta sobre la tabla `alumnos`.
- Se ejecuta antes de una operación de actualización.
- Si el nuevo valor de la nota que se quiere actualizar es negativo, se mantiene el valor anterior.
- Si el nuevo valor de la nota que se quiere actualizar es mayor que 10, se mantiene el valor anterior.
- Si el valor era correcto se actualizará el valor en `nota_texto`.

Una vez creados los *triggers* escribe varias sentencias de inserción y actualización sobre la tabla `alumnos` y verifica que los *triggers* se están ejecutando correctamente.

2. Sobre lo creado en el ejercicio 14.7.5

Escribe un *trigger* con las siguientes características:

- Nombre: `trigger_crear_email_before_insert`.
- Se ejecuta sobre la tabla `alumnos`.
- Se ejecuta antes de una operación de inserción.
- Si el nuevo valor del email que se quiere insertar es `NULL`, entonces se le creará automáticamente una dirección de email y se insertará en la tabla.
- Si el nuevo valor del email no es `NULL` se guardará en la tabla el valor del email.

Nota: Para crear la nueva dirección de email se deberá hacer uso del procedimiento `crear_email`.

3. Modifica el ejercicio anterior y añade un nuevo *trigger* que las siguientes características:

Trigger: `trigger_guardar_email_after_update`:

- Se ejecuta sobre la tabla `alumnos`.
- Se ejecuta después de una operación de actualización.
- Cada vez que un alumno modifique su dirección de email se deberá insertar un nuevo registro en una tabla llamada `log_cambios_email`.

La tabla `log_cambios_email` debe tener los siguientes campos:

- `id`: clave primaria (entero autoincrementado)
- `id_alumno`: id del alumno (entero)
- `fecha_hora`: marca de tiempo con el instante del cambio (fecha y hora)
- `old_email`: valor anterior del email (cadena de caracteres)

- new_email: nuevo valor con el que se ha actualizado
4. Modifica el ejercicio anterior y añade un nuevo *trigger* que tenga las siguientes características:

Trigger: trigger_guardar_alumnos_eliminados:

- Se ejecuta sobre la tabla alumnos.
- Se ejecuta después de una operación de borrado.
- Cada vez que se elimine un alumno de la tabla alumnos se deberá insertar un nuevo registro en una tabla llamada log_alumnos_eliminados.

La tabla log_alumnos_eliminados contiene los siguientes campos:

- id: clave primaria (entero autoincrementado)
- id_alumno: id del alumno (entero)
- fecha_hora: marca de tiempo con el instante del cambio (fecha y hora)
- nombre: nombre del alumno eliminado (cadena de caracteres)
- apellido1: primer apellido del alumno eliminado (cadena de caracteres)
- apellido2: segundo apellido del alumno eliminado (cadena de caracteres)
- email: email del alumno eliminado (cadena de caracteres)

LOS SIGUIENTES EJERCICIOS SON SOBRE LA BDD EMPLEADOS.

5. Crea un *trigger* para cambiar a un empleado de departamento. Debe recibir el código del empleado y el del nuevo departamento. En caso de que se produzca algún problema al asignarle el nuevo departamento, permanecerá en el anterior.
6. Crea un *trigger* para actualizar el salario y la comisión de un empleado.

Después de actualizarlos a los nuevos valores, se debe comprobar si están dentro de los rangos para su trabajo. En caso contrario se desharán los cambios. (Soy consciente de que se podría hacer la comprobación antes de realizar los cambios pero quiero que uses transacciones).

Si se han producido los cambios, se mostrará un mensaje en el que se incluya el nuevo salario, la nueva comisión y la categoría profesional en la que se encuentra el empleado tras el cambio.

Si no se ha podido actualizar, se mostrará un mensaje de error y una lista de los trabajos que podría desempeñar con el salario que se le quería asignar.

Para añadir la lista al mensaje, utiliza un cursor.

Ejercicios 14.9

LOS SIGUIENTES EJERCICIOS SON SOBRE LA BDD JARDINERÍA.

1. Crea una función `calcular_precio_total_pedido` que, dado un código de pedido, calcule la suma total del pedido. Ten en cuenta que un pedido puede contener varios productos diferentes y varias cantidades de cada producto.
2. Crea una función `calcular_suma_pedidos_cliente` que, dado un código de cliente, calcule la suma total de todos los pedidos realizados por el cliente. Deberá hacer uso de la función `calcular_precio_total_pedido` que has desarrollado en el apartado anterior.
3. Crea una función `calcular_suma_pagos_cliente` que, dado un código de cliente, calcule la suma total de los pagos realizados por ese cliente.
4. Crea una función `calcular_pagos_pendientes` que calcule los pagos pendientes de todos los clientes.

Para saber si un cliente tiene algún pago pendiente deberemos calcular cuál es la cantidad de todos los pedidos y los pagos que ha realizado. Si la cantidad de los pedidos es mayor que la de los pagos entonces ese cliente tiene pagos pendientes.

Deberá insertar en una tabla llamada `clientes_con_pagos_pendientes` los siguientes datos:

- `id_cliente`
 - `suma_total_pedidos`
 - `suma_total_pagos`
 - `pendiente_de_pago`
5. Escribe un procedimiento llamado `obtener_numero_empleados` que reciba como parámetro de entrada el código de una oficina y devuelva el número de empleados que tiene.

Escribe una sentencia SQL que realice una llamada al procedimiento realizado para comprobar que se ejecuta correctamente.
 6. Escribe una función llamada `cantidad_total_de_productos_vendidos` que reciba como parámetro de entrada el código de un producto y devuelva la cantidad total de productos que se han vendido con ese código.

Escribe una sentencia SQL que realice una llamada a la función realizada para comprobar que se ejecuta correctamente.
 7. Crea una tabla que se llame `productos_vendidos` que tenga las siguientes columnas:
 - `id` (entero sin signo, auto incremento y clave primaria)

- `codigo_producto` (cadena de caracteres)
- `cantidad_total` (entero)

Escribe un procedimiento llamado `estadísticas_productos_vendidos` que para cada uno de los productos de la tabla `producto` calcule la cantidad total de unidades que se han vendido y almacene esta información en la tabla `productos_vendidos`.

El procedimiento tendrá que realizar las siguientes acciones:

- Borrar el contenido de la tabla `productos_vendidos`.
- Recorrer cada uno de los productos de la tabla `producto`. Será necesario usar un cursor.
- Calcular la cantidad total de productos vendidos. En este paso será necesario utilizar la función `cantidad_total_de_productos_vendidos` desarrollada en el ejercicio 14.9.6.
- Insertar en la tabla `productos_vendidos` los valores del código de producto y la cantidad total de unidades que se han vendido para ese producto en concreto.

14.11. Bibliografía.

Por falta de tiempo, en este tema he incluido algunas partes obtenidas del material de José Juan Sánchez Hernández al que agradezco que comparta su trabajo.

Los últimos ejemplos de *triggers* se han obtenido del tutorial [MySQL Triggers](#).

- José Juan Sánchez Hernández (2021/2022). [Bases de datos / Gestión de Bases de datos](#)
- [MySQL 8.0 Reference Manual](#).
- [MySQL Stored Procedures](#). MySQL Tutorial.
- [MySQL Triggers](#). MySQL Tutorial.
- [How to find out the location of currently used MySQL configuration file in linux](#). StackOverflow.