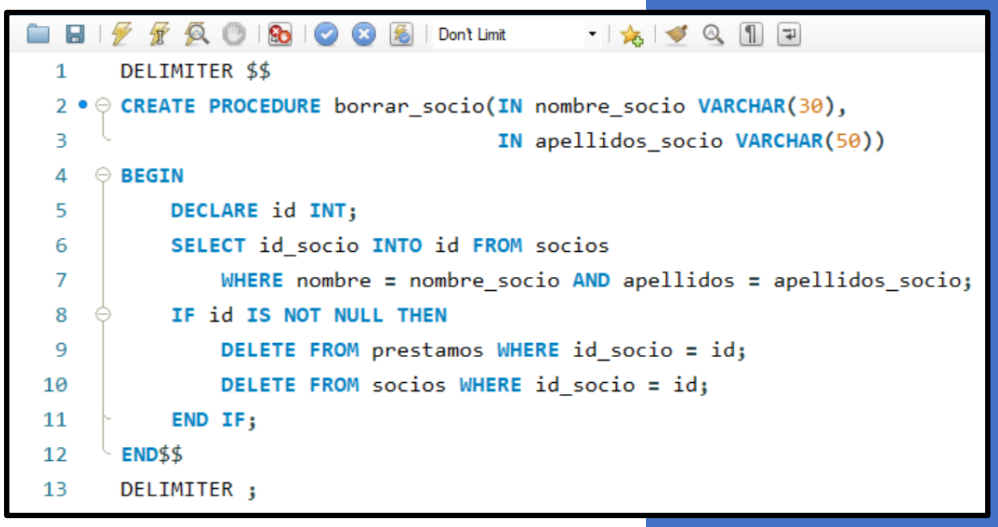
IES Playamar
Curso 2023 - 2024

Tema 6

Programación de bases de datos: procedimientos, funciones, eventos y triggers.



```
1   DELIMITER $$
2   CREATE PROCEDURE borrar_socio(IN nombre_socio VARCHAR(30),
3                                 IN apellidos_socio VARCHAR(50))
4   BEGIN
5       DECLARE id INT;
6       SELECT id_socio INTO id FROM socios
7           WHERE nombre = nombre_socio AND apellidos = apellidos_socio;
8       IF id IS NOT NULL THEN
9           DELETE FROM prestamos WHERE id_socio = id;
10          DELETE FROM socios WHERE id_socio = id;
11      END IF;
12  END$$
13  DELIMITER ;
```

Módulo: Bases de datos

Autor: Manuel Bautista

Este documento ha sido elaborado para el alumnado del módulo “Bases de datos” de los CFGS DAM y DAW del IES Playamar.

Contenido

1. Introducción

Como se ha visto en las unidades anteriores, SQL es un lenguaje muy potente y flexible que permite la creación y manipulación de una base de datos relacional. No obstante, al tratarse de un lenguaje declarativo, presenta limitaciones. Por ejemplo, imagina que se desea borrar un cliente de una base de datos y para ello antes hay que borrar todos los datos de sus pedidos y llamadas a atención al cliente (porque hay restricciones de borrado). Para borrar el cliente habría que hacer lo siguiente:

1. Comprobar que el cliente existe y localizar su ID.
2. Si existe ID:
    a. Localizar todos los datos de pedidos asociados a ese ID y borrarlos.
    b. Localizar todos los datos de atención al cliente a ese ID y borrarlos.
    c. Finalmente, borrar el cliente.

El código de programa que implemente esta funcionalidad debe realizar cuatro accesos a la base de datos. Sería más cómodo para el programa que la base de datos incorporara una funcionalidad que recibiera un nombre de cliente y realizara todas las operaciones necesarias para su borrado. Además de simplificar la operación, también se optimiza el rendimiento del programa ya que los accesos a la base de datos se reducen a uno.

Para estos supuestos y muchos más los SGBDR incorporan, además de SQL, la capacidad de añadir guiones (o *scripts*) a la base de datos que automatizan determinadas tareas. Estos guiones amplían el lenguaje SQL incorporando capacidades de lenguajes imperativos como Java: variables, estructuras de control, procedimientos, funciones, eventos y control de excepciones.

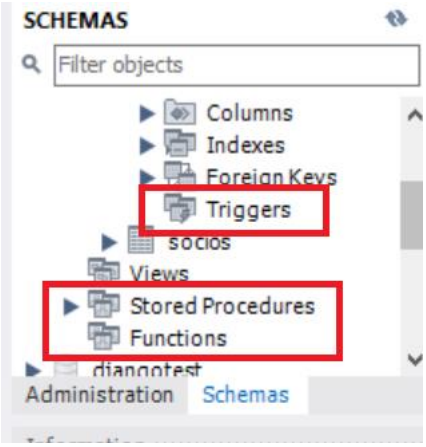
Aunque ISO define el lenguaje estándar SQL/PSM (SQL Persistent Stored Modules), la realidad es que cada SGBDR incorpora su propio lenguaje de programación (por ejemplo: Oracle tiene PL/SQL y SQL Server el lenguaje Transact-SQL, llamado también T-SQL). MySQL define su propio lenguaje muy cercano al estándar definido por ISO y sin nombre específico.

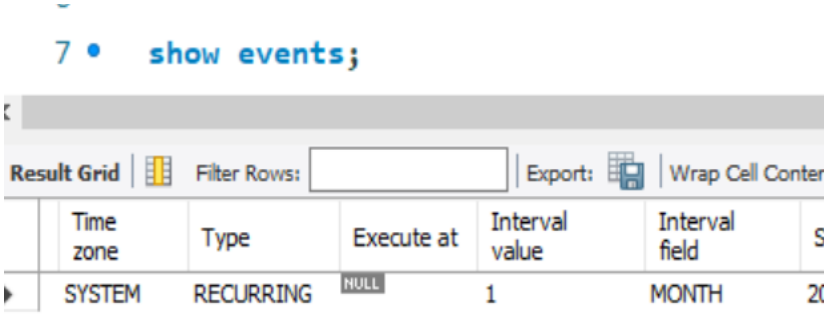
2. El lenguaje de programación de MySQL

El lenguaje de programación que incorpora MySQL permite crear guiones para programar y guardar en el servidor el siguiente código:

- Procedimientos almacenados.
- Funciones.
- Disparadores (o *triggers*).
- Eventos.

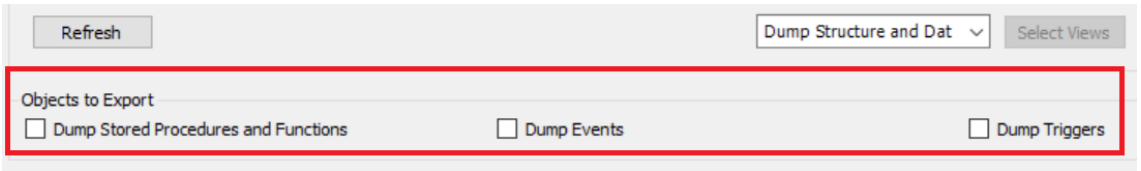
Desde MySQL Workbench se pueden consultar los guiones añadidos a la base de datos:





IMPORTANTE

Si se va a exportar una base de datos a la que se le han añadido guiones, es muy probable que sea necesario añadirlos en la exportación. Para ello hay que marcar las casillas de los guiones que se desean añadir a la exportación.



A continuación, se describen las características básicas del lenguaje de MySQL para llevar a cabo estas tareas de programación.

2.1. Configuración del delimitador

Cualquier guion que se programe en la base de datos consta de un número variable de instrucciones SQL (select, insert, update, delete…). MySQL espera que cada instrucción esté delimitada por el símbolo punto y coma (;). No obstante, todo el guion que se programa es realmente un único bloque que también debe ir delimitado. Esto puede llegar a confundir a la base de datos y provocar un comportamiento anómalo.

Para solucionar este problema, se utiliza la instrucción DELIMITER. Esta instrucción modifica el delimitador que separa instrucciones para darle un nuevo valor de forma temporal. Antes de comenzar a escribir el guion debe indicarse el delimitador provisional que será usado.

Por ejemplo, para usar el delimitador $$:

```
DELIMITER $$
```

IMPORTANTE

Al finalizar el guion debe restablecerse el valor por defecto del delimitador. Asegúrate siempre de añadir al final del guion la siguiente instrucción:

DELIMITER ;

¿POR QUÉ $$?

MySQL permite usar cualquier combinación de caracteres como delimitador ($$, //, @@...). Puedes utilizar la combinación que más te guste y **que no se vaya a repetir en el código del guion**. Es común utilizar $$ porque es el delimitador que obliga a utilizar phpMyAdmin. Se trata de una herramienta web de administración de base de datos MySQL con utilidades similares a MySQL Workbench.

2.2. Variables

Las variables se definen con la instrucción DECLARE:

```
DECLARE variable1 [, variable2...] <tipo> [DEFAULT valor]
```

En una misma instrucción DECLARE se pueden definir una o varias variables del mismo tipo. Opcionalmente se le puede dar un valor por defecto que inicializa todas las variables declaradas. Si no se indica valor por defecto, el valor inicial es NULL.

NOTA

Al igual que SQL, el nombre de una variable no es sensible a las mayúsculas (*case insensitive*). Es decir, los nombres de variable suma, Suma, SUMA hacen referencia a la misma variable.

Para asignar un valor a la variable se usa la instrucción SET.

```
SET variable = <expresión>;
```

La expresión puede ser un valor simple, una operación o una consulta que devuelve un dato. También es posible utilizar la instrucción INTO dentro de una consulta que devuelve una fila para asignar valor a una o varias variables, tantas como columnas devuelva la consulta:

```
SELECT campo1, campo2...
INTO variable1, variable2...
```

CASO PRÁCTICO 1

Para crear una variable total_habitantes con valor inicial cero.

```
DECLARE total_habitantes INT DEFAULT 0;
```

Para guardar en la variable el total de habitantes de la provincia con ID 3:

```
SET total_habitantes = (SELECT sum(habitantes) FROM municipios
                        WHERE id_provincia = 3);
```

También se puede hacer con la instrucción INTO:

```
SELECT sum(habitantes) INTO total_habitantes
FROM municipios WHERE id_provincia = 3);
```

2.3. Estructuras de selección

Las estructuras de selección permiten tomar decisiones en función de una determinada condición. Las estructuras incluidas en MySQL son IF y CASE. Con estas dos instrucciones se puede construir cualquier tipo de selección, ya sea simple, doble o múltiple.

La instrucción IF tiene la siguiente sintaxis:

```
IF <condición> THEN
     <instrucciones>
[ELSEIF <condición> THEN
     <instrucciones>] ...
[ELSE
     <instrucciones>]
END IF;
```

CASO PRÁCTICO 2

Imagina que en la tabla accesos(id_usuario, num_accesos) se guarda el número de veces que accede un usuario. Si es la primera vez que accede (no aparece en la tabla), hay que insertar una nueva fila para el usuario, en otro caso se actualiza el contador de accesos. Suponiendo que el ID del usuario está en una variable usuario:

```
SELECT id_usuario INTO fila FROM ACCESOS WHERE id_usuario = usuario;

IF fila IS NULL
        INSERT INTO accesos(id_usuario, num_accesos)
        VALUES (usuario, 1);
ELSE
        UPDATE accesos SET
                num_accesos = num_accesos + 1
        WHERE id_usuario = fila;
END IF;
```

La instrucción CASE tiene dos formas de uso:

```
CASE <valor>
WHEN v1 THEN
    <instrucciones>
[WHEN v2 THEN
    <instrucciones>] ...
[ELSE
    <instrucciones>]
END CASE;
```

```
CASE
WHEN <condición1> THEN
   <instrucciones>
[WHEN <condición2> THEN
   <instrucciones>] ...
[ELSE
   <instrucciones>]
END CASE;
```

CASO PRÁCTICO 3

Imagina que se necesita guardar en la variable resultado la calificación cualitativa del alumno a partir de su nota numérica, almacenada en la variable nota.

DECLARE resultado VARCHAR(20);

CASE

WHEN nota >= 0 AND nota < 5 THEN SET resultado = 'Suspenso';

WHEN nota >= 5 AND nota < 6 THEN SET resultado = 'Suficiente';

WHEN nota >= 6 AND nota < 7 THEN SET resultado = 'Bien';

WHEN nota >= 7 AND nota < 9 THEN SET resultado = 'Notable';

WHEN nota >= 9 AND nota < 10 THEN SET resultado = 'Sobresaliente';

WHEN nota = 10 THEN SET resultado = 'Matrícula de Honor';

ELSE SET resultado = 'Nota incorrecta';

END CASE;

2.4. Estructuras de repetición

Las estructuras de repetición permiten repetir un conjunto de instrucciones hasta que se cumpla una condición determinada. El conjunto de instrucciones recibe el nombre de bucle. Las estructuras incluidas en MySQL son WHILE y REPEAT, con las que se pueden ejecutar un bucle ninguna vez, una vez o varias veces.

NOTA

MySQL también incluye la instrucción LOOP. No obstante, esta instrucción no cumple con los principios de la programación estructurada por lo que es muy buena práctica evitar su uso.

La instrucción WHILE tiene la siguiente sintaxis:

```
WHILE <condición> DO
     <instrucciones>
END WHILE;
```

La condición (o invariante del bucle) es evaluada al inicio del bucle y, si se cumple, se ejecutan sus instrucciones. Una vez ejecutadas todas las instrucciones se vuelve a la situación inicial. Por tanto, el bucle se ejecuta *mientras* se cumpla la condición. Si la condición no se cumple la primera vez, el bucle no se ejecuta nunca.

CASO PRÁCTICO 4

Imagina que en la tabla numeros(id, valor) se necesita almacenar diez números aleatorios comprendidos entre 1 y 100:

```
DECLARE contador, numero INT DEFAULT 0;
WHILE contador < 10 DO
     SET numero = (SELECT FLOOR(1 + (RAND() * 100)));
     SET contador = contador + 1;
     INSERT INTO numeros(valor) VALUES(numero);
END WHILE;
```

El bucle se repite mientras no se hayan generado aun los diez números.

La instrucción REPEAT tiene la siguiente sintaxis:

```
REPEAT
     <instrucciones>
UNTIL <condición>
END REPEAT;
```

En este caso, la condición es evaluada después de ejecutar el bucle y, si no se cumple, se vuelve a la situación inicial. Por tanto, el bucle se ejecuta *hasta que* se cumpla la condición. El bucle siempre se ejecuta al menos una vez.

CASO PRÁCTICO 5

El ejemplo del caso 4 se puede implementar con un bucle REPEAT del siguiente modo:

```
DECLARE contador, numero INT DEFAULT 0;
REPEAT
        SET numero = (SELECT FLOOR(1 + (RAND() * 100)));
        SET contador = contador + 1;
        INSERT INTO numeros(valor) VALUES(numero);
UNTIL contador = 10
END REPEAT;
```

Ahora se repite el bucle hasta que el contador de inserciones llegue a diez.

2.5. Control de errores

Cuando se produce un error en una instrucción se eleva una excepción y se detiene la ejecución del código. MySQL permite controlar esos errores para evitar que el código se detenga, para ello se necesita definir un manejador de error. Se puede definir un manejador para uno o varios errores con la siguiente sintaxis:

```
DECLARE <acción> HANDLER
      FOR <condición> [, <condición>] ...
      <instrucciones>;
```

<acción> indica cómo se comporta el manejador de error. Las opciones son:

- EXIT: la ejecución del programa donde se define el manejador finaliza. Es la opción por defecto si no se definen manejadores de errores.
- CONTINUE: la ejecución del programa continua.

NOTA

El estándar SQL/PSM define una tercera opción UNDO que MySQL aun no soporta.

<condición> indica la condición de error que se maneja. Las opciones son:

- Código de error: indica el código de error numérico asociado al error. Los códigos de error que define MySQL se pueden consultar en el siguiente enlace: https://dev.mysql.com/doc/mysql-errors/8.0/en/server-error-reference.html#error_er_invalid_field_size.
- SQLSTATE error: el error también se representa con un código de texto de 5 caracteres. Si se desea indicar así, esta es la opción a utilizar.
- SQLWARNING: es un atajo para indicar todas las advertencias de MySQL.
- SQLEXCEPTION: es un atajo para indicar todas las excepciones de MySQL.
- NOT FOUND: indica que no se encontraron datos. Esta condición se usa en cursores (se verá en el siguiente apartado).

<instrucciones> indica el código que se ejecuta si se produce el error que se desea manejar. Si el código es un bloque de más de una instrucción debe ir encerrada entre los comandos BEGIN y END.

CASO PRÁCTICO 6

Se produce el error 1102 si se intenta crear una base de datos con un nombre incorrecto, y el error 1103 si lo que se intenta crear es una tabla. En ambos casos, se desea poner a 1 una variable error_sintaxis:

```
DECLARE CONTINUE HANDLER FOR 1102, 1103 SET error_sintaxis = 1;
```

Se desea poner a 1 una variable hay_excepcion, registrar el fallo en una tabla errores y finalizar el guion cada vez que hay una excepción:

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION BEGIN
        SET hay_excepcion = 1;
        INSERT INTO errores(fecha, descripcion)
        VALUES (current_timestamp(), 'Error al revisar los datos de cliente');
END;
```

2.6. Errores personalizados

Se pueden crear mensajes de error personalizados con la instrucción SIGNAL:

```
SIGNAL SQLSTATE <valor>
[SET <información>=<valor>][, <información>=<valor>]... ;
```

En cuanto al valor asociado a SQLSTATE, generalmente se usa el valor '45000' que es el valor reservado para las excepciones definidas por el usuario. No obstante, se puede utilizar cualquier valor siempre que no empiece por el prefijo '00' (los valores que empiezan con ese prefijo se reservan para ejecuciones que finalizan sin error).

Junto con el código de error se pueden añadir parámetros para dar más detalle del error, por ejemplo:

- TABLE_NAME, COLUMN_NAME: indica la tabla y columna relacionadas con el error.
- MESSAGE_TEXT: mensaje personalizado describiendo el error.
- MYSQL_ERRNO: número del mensaje de error.

Más información: https://dev.mysql.com/doc/refman/8.0/en/signal.html.

CASO PRÁCTICO 7

Se desea elevar una excepción informando de que un cliente no se ha podido borrar porque no existe:

```
DECLARE id INT;
SET id = (SELECT id_cliente FROM clientes WHERE nombre = 'cliente1'
                AND apellidos = 'apellido1');
IF id IS NULL THEN
        SIGNAL SQLSTATE '45000' SET
            MESSAGE_TEXT = 'Cliente no existe',
            TABLE_NAME = 'clientes';
END IF;
```

2.7. Cursores

Los cursores son estructuras que permiten recorrer las filas que son resultados de una consulta. El recorrido es de sólo lectura y en MySQL sólo es posible el recorrido de forma secuencial desde la primera fila obtenida hasta la última y sin realizar saltos. Para poder recorrer los resultados de una consulta es necesario combinar las instrucciones de cursor con una instrucción repetitiva.

En primer lugar, es necesario declarar el nombre del cursor y la consulta que va a recorrer:

```
DECLARE <nombre> CURSOR FOR <consulta>;
```

<consulta> es la instrucción SELECT a recorrer.

Cuando se recorre un cursor y se alcanza la última fila se lanza el error NOT FOUND (que se corresponde con el código de error 1329 y el valor SQLSTATE '02000'). Por eso cuando se trabaja con cursores es necesario manejar este error y debe indicarse después de su declaración de alguno de estos tres modos:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND ... ;
DECLARE CONTINUE HANDLER FOR 1329 ... ;
DECLARE CONTINUE HANDLER FOR SQLSATE '02000' ... ;
```

El siguiente paso es abrir el cursor:

```
OPEN <nombre>;
```

Una vez abierto el cursor ya se puede recorrer el resultado de la consulta:

```
FETCH <cursor> INTO var1 [, var2] ... ;
```

A continuación de INTO se deben indicar tantas variables como columnas devuelve la consulta. Los tipos de las variables deben coincidir (o ser compatibles) con los tipos de las columnas relacionadas.

Finalmente, el cursor debe ser cerrado:

```
CLOSE <cursor>;
```

CASO PRÁCTICO 8

Imagina que se necesita saber cuántos números pares hay en la tabla numeros(id, valor) utilizada en los casos anteriores. Se puede resolver con una consulta agrupada:

```
SELECT count(*) FROM numeros WHERE valor %2 = 0;
```

También es posible una solución basada en cursores. Se define un cursor para recorrer la tabla y se van contando las filas:

```
DECLARE contador, numero, fin INT DEFAULT 0;
DECLARE c_numeros CURSOR FOR SELECT valor FROM numeros;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin = 1;
OPEN c_numeros;
FETCH c_numeros INTO numero;
WHILE fin = 0 DO
        IF numero % 2 = 0 THEN
                SET contador = contador + 1;
        END IF;
        FETCH c_numeros INTO numero;
END WHILE;
CLOSE c_numeros;
```

2.8. Transacciones

Por defecto, MySQL se ejecuta con el modo autocommit activado. Esto aplica también a los guiones, cualquier operación de manipulación de datos que se realiza y finaliza correctamente se confirma de forma automática y no se puede deshacer. No obstante, es posible modificar este comportamiento por defecto si se necesita que todas las operaciones del guion formen parte de la misma transacción y se realicen de forma atómica.

Para iniciar una transacción en el guion, aunque el modo autocommit esté activado, se ejecuta la siguiente instrucción:

```
START TRANSACTION;
```

NOTA

Para iniciar una transacción también se puede utilizar la instrucción BEGIN o BEGIN WORK, que son alias soportados por MySQL. No obstante, es recomendable utilizar la instrucción START TRANSACTION ya que es la instrucción estándar que además permite indicar parámetros de configuración.

Si todas las operaciones se realizan correctamente, para dar por finalizada la transacción se ejecuta la instrucción:

```
COMMIT [WORK};
```

Si se produce un fallo y es necesario deshacer la transacción para volver al estado inicial de la base de datos, se ejecuta la instrucción:

```
ROLLBACK [WORK};
```

NOTA

Es posible desactivar el modo autocommit con la siguiente instrucción:

SET autocommit = 0;

Este cambio solo afecta a la sesión abierta. Es decir, si se cierra sesión y se vuelve a conectar a la base de datos, el modo autocommit volvería a estar activado. Si se desea desactivar autocommit de forma permanente, se debe revisar el fichero de configuración de MySQL:

[mysqld]

autocommit=0

CANCELACIONES PARCIALES

La instrucción ROLLBACK deshace todas las operaciones realizadas en la transacción devolviendo la base de datos al estado que había al inicio de la transacción. Es posible realizar cancelaciones parciales con la instrucción SAVEPOINT, aunque sólo está soportado en las tablas que utilizan el motor InnoDB (es el motor por defecto).

Para saber más sobre las cancelaciones parciales y sobre los motores de MySQL:

https://dev.mysql.com/doc/refman/8.0/en/savepoint.html.

https://dev.mysql.com/doc/refman/8.0/en/storage-engines.html.

CASO PRÁCTICO 9

El código del caso 4 inserta diez valores aleatorios en una tabla. Imagina que esas diez inserciones tienen que realizarse dentro de una transacción y, si se produce un error, se registra en una tabla errores y se finaliza el guion. Se puede hacer del siguiente modo:

```
DECLARE contador, numero INT DEFAULT 0;
DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        INSERT INTO errores(fecha, descripcion)
        VALUES (current_timestamp(), 'Error al insertar los valores aleatorios');
        ROLLBACK;
    END;
START TRANSACTION;
WHILE contador < 10 DO
        SET numero = (SELECT FLOOR(1 + (RAND() * 100)));
        SET contador = contador + 1;
        INSERT INTO numeros(valor) VALUES(numero);
END WHILE;
COMMIT;
```

3. Procedimientos

Un procedimiento (también llamado procedimiento almacenado) es un conjunto de instrucciones SQL almacenado en la base de datos. Se identifica con un nombre y puede incluir parámetros. El procedimiento se crea del siguiente modo:

```
CREATE PROCEDURE <nombre>([parámetros...])
BEGIN
    <instrucciones>
END;
```

Si se definen parámetros en el procedimiento, éstos actúan como variables dentro del procedimiento creado. Para cada parámetro que se indique, se debe informar:

- El tipo de parámetro, que puede ser de entrada (IN), de salida (OUT) o de entrada y salida (INOUT). Si se incluyen parámetros OUT o INOUT, el procedimiento puede devolver uno o varios valores por medio de estos parámetros.
- El nombre del parámetro.
- El tipo del parámetro.

Para ejecutar un procedimiento hay que utilizar la instrucción CALL, donde se debe indicar el nombre del procedimiento a ejecutar y, si tiene parámetros, los valores o variables de cada parámetro.

Un procedimiento se borra con la instrucción:

```
DROP PROCEDURE [IF EXISTS] <nombre>;
```

ALTER PROCEDURE

Si bien MySQL incluye la instrucción ALTER PROCEDURE, ésta no permite modificar los parámetros ni el cuerpo. Por esa razón, para modificar el procedimiento es mejor utilizar DROP y crearlo de nuevo con los cambios necesarios en el código.

DROP PROCEDURE IF EXISTS <nombre>;

CREATE PROCEDURE nombre(…)

BEGIN …

CASO PRÁCTICO 10

Cuando en una biblioteca se necesita borrar un socio, se deben borrar todos los préstamos que realizó además de sus datos. Esta operación supone el borrado en dos tablas y se puede simplificar con el siguiente procedimiento que recibe el nombre del socio:

```
DELIMITER $$
CREATE PROCEDURE borrar_socio(IN nombre_socio VARCHAR(30),
                              IN apellidos_socio VARCHAR(50))
BEGIN
    DECLARE id INT;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION ROLLBACK;
    SELECT id_socio INTO id FROM socios
        WHERE nombre = nombre_socio AND apellidos = apellidos_socio;
    IF id IS NOT NULL THEN
        DELETE FROM prestamos WHERE id_socio = id;
        DELETE FROM socios WHERE id_socio = id;
    END IF;
END$$
DELIMITER ;
```

Para ejecutar este procedimiento, se debe llamar a la instrucción CALL indicando el nombre y apellidos del socio. Por ejemplo:

```
CALL borrar_socio('Miguel', 'Sanchéz Aguilar');
```

4. Funciones

Al igual que un procedimiento, una función es un conjunto de instrucciones SQL almacenado en la base de datos identificado con un nombre y que puede incluir parámetros. Las diferencias que presenta respecto al procedimiento son:

- Una función se llama desde una sentencia SELECT o desde una expresión.
- Los parámetros solo pueden ser de entrada.
- La función devuelve un único valor.

Las funciones se crean con la instrucción:

```
CREATE FUNCTION <nombre>([parámetros...])
     RETURNS <tipo> <características>
BEGIN
     <instrucciones>
END;
```

Si se definen parámetros, éstos se definen del mismo modo que en los procedimientos con la salvedad de que no es necesario indicar el tipo de parámetro IN puesto que todos los parámetros de funciones son siempre de entrada.

A continuación de la definición de parámetros se indica la palabra RETURNS y el tipo del valor que devuelve la función. El valor que devuelve la función se indica en el cuerpo de la función con la instrucción RETURN.

Después de la definición del tipo de dato que devuelve la función, se deben indicar las características 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**: indica que la función no siempre devuelve el mismo resultado, aunque se utilicen los mismos parámetros de entrada. Esta es la opción por defecto cuando no se indica una característica de forma explícita.
- **CONTAINS SQL**: Indica que la función contiene sentencias SQL, pero no contiene sentencias de manipulación de datos.
- **NO SQL**: Indica que la función no contiene sentencias SQL.

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

Para poder crear una función en MySQL es necesario indicar al menos una de estas tres características: DETERMINISTIC, NO SQL, READS SQL DATA. Si no se indica al menos una de estas características se produce un error.

Para ejecutar una función hay que llamarla dentro de un SELECT o expresión, donde se debe indicar el nombre de la función a ejecutar y, si tiene parámetros, los valores o variables de cada parámetro.

Una función se borra con la instrucción:

```
DROP FUNCTION [IF EXISTS] <nombre>;
```

ALTER FUNCTION

Al igual que en procedimientos, existe la instrucción ALTER FUNCTION que sólo permite modificar las características de la función. Por esa razón, para modificar una función es mejor utilizar DROP y crear de nuevo la función con los cambios necesarios en el código.

DROP FUNCTION IF EXISTS <nombre>;

CREATE FUNCTION nombre(…) RETURNS …

CASO PRÁCTICO 11

El código del caso 7 se puede definir en una función:

```
DELIMITER $$
CREATE FUNCTION contar_pares() RETURNS INT READS SQL DATA
BEGIN
        DECLARE contador, num, fin INT DEFAULT 0;
        DECLARE c_numeros CURSOR FOR SELECT valor FROM numeros;
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin = 1;
```

```
        OPEN c_numeros;
        FETCH c_numeros INTO num;
        WHILE fin = 0 DO
                IF num % 2 = 0 THEN
                        SET contador = contador + 1;
                END IF;
                FETCH c_numeros INTO num;
        END WHILE;
        CLOSE c_numeros;
   RETURN contador;
END $$
DELIMITER ;
```

Para ejecutar la función se puede usar SELECT:

```
SELECT contar_pares();
```

También se puede llamar desde una expresión que admita un valor del tipo que devuelve la función:

```
IF contar_pares() > 0 THEN …
```

5. Disparadores (*triggers*)

Un disparador o *trigger* es un guion que se asocia a una tabla y que se activa (dispara) cuando ocurre un evento que modifica el contenido de una tabla (INSERT, UPDATE, DELETE). Se pueden crear disparadores sobre cualquier tabla que no sea temporal ni vista. Son herramientas muy útiles para implantar en la base de datos validaciones impuestas por las reglas de negocio.

IMPORTANTE

Los disparadores no se activan si los datos se borran con una instrucción TRUNCATE TABLE o DROP TABLE.

Un disparador se crea con la siguiente instrucción:

```
CREATE TRIGGER <nombre> <cuándo> <evento>
     ON <tabla> FOR EACH ROW [<orden>]
BEGIN
     <instrucciones>
END;
```

Donde:

- <cuando> indica si el disparador se activa antes (BEFORE) o después (AFTER) de que suceda el evento.
- <evento> indica la instrucción SQL que desencadena la activación del disparador (INSERT, UPDATE o DELETE).
- <orden> indica si el disparador se ejecuta antes (PRECEDES <trigger>) o después (FOLLOWS <trigger>) de otro disparador. Se puede definir más de un disparador para un mismo evento de una tabla.

NOTA

Es buena práctica indicar en el nombre del disparador el tiempo, evento y si se activa antes o después de otro. Por ejemplo, para indicar un disparador en la tabla Préstamos que se activa después de insertar, estas opciones son buenas:

prestamos_after_insert

prestamos_AI

Un disparador se borra cuando se elimina la tabla relacionada. También es posible borrarlo con la siguiente instrucción:

```
DROP TRIGGER <nombre>;
```

Para realizar las validaciones que se necesiten, en el código del disparador se puede hacer referencia al registro anterior al evento (OLD) o al nuevo registro (NEW), según la siguiente tabla:

EVENTO	OLD	NEW
INSERT	NO	SÍ
UPDATE	SÍ	SÍ
DELETE	SÍ	NO

CASO PRÁCTICO 12

Imagina que en la tabla de socios hay un campo retrasos que guarda el número de devoluciones que se entregan con retraso. Cada vez que se informe la fecha de devolución del libro, se debe comprobar si es mayor que la fecha límite. Si es así, hay que incrementar el contador de retrasos del socio:

```
delimiter $$
CREATE TRIGGER prestamos_after_update
        AFTER UPDATE
   ON prestamos FOR EACH ROW
BEGIN
        IF (NEW.fecha_devolucion IS NOT NULL) AND
          (NEW.fecha_limite < NEW.fecha_devolucion) THEN
                UPDATE socios SET
                        retrasos = retrasos + 1
                WHERE id_socio = NEW.id_socio;
        END IF;
END $$
DELIMITER ;
```

CASO PRÁCTICO 13

Imagina que no se permite darle un préstamo al socio de la biblioteca si éste ya acumula o supera cinco retrasos en la devolución de libros. Por tanto, al insertar un nuevo préstamo se debe consultar el número de retrasos y, si es cinco o superior, se lanza una excepción para impedir el préstamo:

```
delimiter $$
CREATE TRIGGER prestamos_before_insert
        BEFORE INSERT
   ON prestamos FOR EACH ROW
BEGIN
    DECLARE num_retrasos INT;
    SELECT retrasos INTO num_retrasos
    FROM socios
    WHERE id_socio = NEW.id_socio;
    IF num_retrasos >= 5 THEN
        SIGNAL SQLSTATE '45000' SET
            MESSAGE_TEXT = 'Se alcanzó el máximo de retrasos';
        END IF;
 END $$
DELIMITER  ;
```

IMPORTANTE

Los disparadores pueden tener un impacto enorme sobre el rendimiento de la base de datos ya que es un código que se activa tras cada inserción, actualización o borrado de datos de una tabla. Este código debe ser lo más reducido y óptimo posible para no ralentizar la base de datos.

6. Eventos

Un evento es una tarea que se programa para ser ejecutada de acuerdo a un horario, ya sea de forma puntual o de forma periódica. Se crean con la siguiente instrucción:

```
CREATE EVENT <nombre> ON SCHEDULE <cuándo>
DO
BEGIN
END;
```

<cuándo> permite indicar si el evento se ejecuta una única vez o de forma periódica.

EVENTO	CONFIGURACIÓN
Una sóla vez	AT <fecha_hora> [+ INTERVAL <intervalo>]
Periódico	EVERY <intervalo> [STARTS <intervalo>] [ENDS <intervalo>]

Más información: https://dev.mysql.com/doc/refman/8.0/en/create-event.html.

Para ver los eventos programados en la base de datos, y el detalle de un evento, se usan respectivamente las siguientes instrucciones:

```
SHOW EVENTS;
SHOW CREATE EVENT <nombre>;
```

Existe la posibilidad de modificar un evento ya creado (ALTER) y borrarlo si no se necesita más:

```
ALTER EVENT ... (sintaxis idéntica a CREATE EVENT)
DROP EVENT <nombre>;
```

NOTA

Para modificar un evento que ya existe, en realidad es más cómodo borrarlo y crearlo de nuevo. De este modo sólo se necesita un guion para la creación/modificación del evento.

CASO PRÁCTICO 14

Si el ejemplo del caso 4 se tuviera que hacer en un evento que debe activarse dentro de cinco minutos (contando el instante actual), quedaría así:

```
DELIMITER $$
CREATE EVENT insertar_numeros
        ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 5 MINUTE
DO
BEGIN
   DECLARE contador, numero INT DEFAULT 0;
   WHILE contador < 10 DO
        SET numero = (SELECT FLOOR(1 + (RAND() * 100)));
        SET contador = contador + 1;
        INSERT INTO numeros(valor) VALUES(numero);
   END WHILE;
END $$
DELIMITER ;
```

CASO PRÁCTICO 15

Algunos ejemplos más de configuraciones de eventos.

Cada hora:

```
ON SCHEDULE EVERY 1 HOUR
```

El día 1 de cada mes a las 5 de la mañana, comenzando el 1 de mayo:

```
ON SCHEDULE EVERY 1 MONTH STARTS '2024-05-01 05:00'
```

Cada 15 minutos durante las próximas 24 horas:

```
ON SCHEDULE EVERY 15 MINUTE
        STARTS CURRENT_TIMESTAMP
        ENDS CURRENT_TIMESTAMP + 24 HOUR
```

IMPORTANTE

Hay que tener cuidado con los eventos periódicos que se activan con mucha frecuencia ya que, al igual que los disparadores, pueden tener un impacto enorme sobre el rendimiento de la base de datos.

PLANIFICADOR DE EVENTOS (EVENT SCHEDULER)

El planificador de eventos se encarga de ejecutar los eventos programados. Para que un evento funcione el planificador debe estar activado. Su estado se puede consultar así:

```
SHOW VARIABLES WHERE variable_name = 'event_scheduler';
```

Los eventos se activarán si el valor de evento_scheduler es ON. Si no es así, debe activarse para que los eventos se ejecuten:

```
SET GLOBAL event_scheduler = ON;
```

Una vez creado un evento, es posible desactivarlo (sin borrarlo) asi:

ALTER EVENT *nombre_de_evento* DISABLE;

Un evento desactivado puede activarse asi:

ALTER EVENT *nombre_de_evento* ENABLE;

7. Resumen

La existencia de un lenguaje de programación integrado en los SGBD permite disminuir el tráfico de red entre la base de datos y los programas que se conectan con ella. También permite agrupar las funcionalidades relacionadas con los datos e independizarlas del resto de procesos.

Aunque existe un estándar llamado SQL/PSM, cada fabricante define su propio lenguaje de programación. La gestión que hacen estos lenguajes sobre variables, funciones, procedimientos y estructuras de control es similar a la de otros lenguajes de programación. Cabe destacar el uso de cursores (estructuras de almacenamiento en memoria que contienen el resultado de una consulta).

Un elemento muy importante en todo SGBD son los disparadores o *triggers*, bloques de código que se ejecutan a partir de una acción sobre los datos de una tabla (inserción, actualización o borrado) y ayudan a mantener la integridad de los datos almacenados.

Finalmente, la programación de eventos facilita las tareas de administración, ya que permite automatizar tareas periódicas como copias de seguridad, generación de estadísticas o gestión de usuarios.