DAW/DAM. UD 7. USUARIOS Y EXTENSIONES PARTE 2. EXTENSIONES DE MYSQL

DAW/DAM. Bases de datos (BD)

UD 7. USUARIOS Y EXTENSIONES

Parte 2. Extensiones de MySQL

Abelardo Martínez y **Pau Miñana**
Basado y modificado de Sergio Badal (www.sergiobadal.com) y Raquel Torres.
Curso 2023-2024

1. Lenguaje de programación en MySQL

Aunque el lenguaje de programación por excelencia para bases de datos es PL/SQL, otros SGBD como MySQL también permiten crear procedimientos y funciones. En esta unidad vamos a estudiar los **PROGRAMAS** que podemos escribir con MySQL, que son objetos que contienen código SQL y se almacenan asociados a una base de datos concreta. Los programas pueden ser de 3 tipos:

Tipo	Funcionalidad
Procedimientos almacenados	- Son objetos que se crean con la sentencia CREATE PROCEDURE, se invocan/llaman con la sentencia CALL y se eliminan con la sentencia DROP PROCEDURE. - Un procedimiento puede tener cero o muchos parámetros de entrada, cero o muchos parámetros de salida o cero o muchos parámetros de entrada y salida.
Funciones	- Son objetos que se crean con la sentencia CREATE FUNCTION, se invocan/llaman dentro de una sentencia SELECT o dentro de una expresión y se eliminan con la sentencia DROP FUNCTION. - Una función puede tener cero o muchos parámetros de entrada y siempre devuelve un valor, asociado al nombre de la función.
Triggers o disparadores	- Son objetos que se crean con la sentencia CREATE TRIGGER, tienen que estar asociados a una tabla concreta y se eliminan con la sentencia DROP TRIGGER. - Un trigger se activa cuando ocurre un evento de inserción, actualización o borrado (lenguaje DML), sobre la tabla a la que está asociado.

Como toda tecnología, el uso de PROGRAMAS tiene sus pros y sus contras.

a) VENTAJAS

- **Menos tráfico de red**
  - Al reducir la carga en las capas superiores de la aplicación, se reduce el tráfico de red y, con el uso de los programas, puede mejorar el rendimiento.
  - Piensa que si un programa tiene mil líneas, se envía una sola línea a la BD y no mil líneas, ahorrándose una gran cantidad de tráfico de datos.
- **Más seguridad**
  - En cuanto a seguridad, es posible limitar los permisos de acceso de usuario a los programas y no a las tablas directamente.
  - De este modo, evitamos problemas derivados de una aplicación mal programada que haga un mal uso de las tablas.

b) INCONVENIENTES

- **Complejidad**
  - Al poder contener elementos de programación como bucles, variables, etc., requerimos cierta formación, por lo que existe una curva de aprendizaje.
- **Incompatibilidad**
  - Problema con migraciones. No todos los SGBD (Oracle, PostgreSQL, etc.) usan los programas del mismo modo, por lo que se reduce la portabilidad del código.

1.1. Crear y modificar programas

En MySQL **no está contemplada la opción de modificar un PROGRAMA**: si queremos hacer cambios sobre un programa tendremos que eliminarlo y crearlo de nuevo o bien, incluir la sentencia "DROP xxx IF EXISTS". Por ejemplo:

```
DROP PROCEDURE IF EXISTS obtenerProductosPorEstado;
CREATE PROCEDURE obtenerProductosPorEstado (…);

DROP FUNCTION IF EXISTS calcularMedia;
CREATE FUNCTION calcularMedia(…);

DROP TRIGGER IF EXISTS comprobarStock;
CREATE TRIGGER comprobarStock(…);
```

CREATE PROCEDURE|FUNCTION|TRIGGER IF NOT EXISTS se ha incorporado en versiones recientes de MySQL y puede fallar para versiones anteriores. Se recomienda la estructura anterior por ese motivo; además de este modo se asegura que se reescribe el procedimiento, mientras que con el anterior si el procedimiento existe simplemente se ignora la orden y no se modifica nada.

Para poder crear y manipular procedimientos, funciones y triggers es necesario que tengas permisos INSERT y DELETE sobre la base de datos en la que deseas crearlos.

Codificación de programas

Para implementar programas, **es recomendable usar un EDITOR GRÁFICO** que nos ayudará a colorear el código:

- Puedes usar un editor de código genérico como Notepadqq (Linux) o Notepad++ (Windows), indicando en el menú superior LANGUAGE → SQL
- También puedes usar un gestor de bases de datos MySQL como WorkBench, Navicat, Heidi, etc.

Para ejecutar los PROGRAMAS una vez creados en el editor o el gestor, puedes hacerlo directamente desde el gestor MySQL (si has escogido esta opción) o desde la consola como hemos hecho hasta ahora (si preferiste usar el editor de código genérico).

> Asegúrate de que usas la **comilla SIMPLE RECTA**. **Ésta NO: ‘** **Ésta SÍ: '**

1.2. Delimitadores

En MySQL el carácter ";" (punto y coma) se utiliza por defecto para terminar con una instrucción. A la hora de definir programas tendremos que usar delimitadores para indicar al SGBD que se trata de un bloque independiente y evitar la ejecución de cada instrucción independientemente. Debemos establecer el delimitador de comienzo y fin del procedimiento, que normalmente es el **$$**, // o ||.

Fíjate bien, en los ejemplos que verás más adelante, en cómo usamos los DELIMITADORES para decirle a MySQL que NO inicie la ejecución de los comandos hasta que no vuelva a ver el delimitador que indicamos al principio ($$). En dichos ejemplos, **DELIMITER $$** frena la ejecución de MySQL, que se retomará de nuevo en la sentencia **DELIMITER ;** del final. Con la orden “DELIMITER ;” volvemos a establecer el delimitador al punto y coma, que es el delimitador por defecto.

Más info: https://tecnobreros.wordpress.com/2009/07/15/delimitadores-en-mysql/

1.3. Variables y parámetros

Para manipular información dentro de los PROGRAMAS que vayamos a crear (procedimientos, triggers, funciones), necesitaremos contenedores de información (variables y parámetros). Estos contenedores de información pueden ser:

a) Variables de usuario

- Pueden usarse FUERA o DENTRO de los programas que vayamos a usar (sean éstos procedimientos, funciones o triggers).
- SÍ llevan una arroba delante. Por ejemplo: **@variablex**
- NO es necesario declararlas.
- Para asignarles un valor usamos la orden **INTO** si es dentro de un SELECT, o la orden **SET @variablex = valor**; en otro caso.
  - SELECT a, b, c **INTO** (@variablea, @variableb, @variablec) FROM …
  - **SET** @variablea = 34;

b) Variables locales

- Se usan SIEMPRE DENTRO de los programas que vayamos a usar (sean éstos procedimientos, funciones o triggers).
- NO llevan una arroba delante. ~~**@variablex**~~
- SÍ es necesario declararlas con la orden **DECLARE variablex TIPO;**
- Para asignarles un valor usamos la orden **INTO** si es dentro de un SELECT, o la orden **SET variablex = valor**; en otro caso.
  - SELECT a, b, c **INTO** (variablea, variableb, variablec) FROM …
  - **SET** variablea = 34;

c) Parámetros

- Se usan SIEMPRE DENTRO de los programas que vayamos a usar (sean éstos procedimientos, funciones o triggers).
- No llevan una arroba delante. ~~**@param**~~
- Es necesario declararlos en la definición del programa.
- Para asignarles un valor usamos la orden **INTO** si es dentro de un SELECT, o la orden

SET param = valor; en otro caso.

- SELECT a, b, c **INTO** (parama, paramb, paramc) FROM …
- **SET** parama = 34;

Aquí tienes una tabla resumen:

	VARIABLES DE USUARIO	VARIABLES LOCALES	PARÁMETROS
DECLARACIÓN	No es necesario	DECLARE nombre_variable TIPO;	IN/OUT/INOUT nombre_param;
SINTAXIS	@nombre_variable ej. @suma	nombre_variable_o_parametro ej. suma	
ÁMBITO	FUERA/DENTRO de los programas	Siempre DENTRO de los programas	
ASIGNACIÓN	SELECT campo INTO xxx o SET xxx = valor		

1.4. Estructuras de control

1.4.1. Alternativas

Estas instrucciones representan qué hacer en función de una expresión o una condición. Hay 2 tipos:

a) Alternativa simple

La sintaxis es:

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

b) Alternativa compuesta

Esta instrucción es muy versátil, más que los CASE de otros lenguajes de programación. Puede comparar el valor de una variable o el resultado de una expresión con los valores que siguen a la palabra reservada WHEN, o bien puede omitir ese valor o expresión inicial y plantear condiciones independientes en cada uno de los WHEN. La sintaxis es:

```
CASE [expresion]
WHEN {condicion1|valor1} THEN
    bloque_instrucciones_1
WHEN {condicion2|valor2} THEN
    bloque_instrucciones_2
...
[ELSE
    bloque_instrucciones_por_defecto]
END CASE;
```

Más info: https://stackhowto.com/case-when-in-mysql-with-multiple-conditions/

1.4.2. Repetitivas

Estas instrucciones repiten una serie de instrucciones en función de una expresión o una condición. Hay 2 tipos:

a) Bucle WHILE

El bucle WHILE realizará las instrucciones que contiene, mientras la condición sea verdadera. Su sintaxis es:

```
WHILE condición DO
    instrucciones ...
END WHILE;
```

b) Bucle REPEAT

El bucle REPEAT realizará las instrucciones que contiene, hasta que la condición sea verdadera. Su sintaxis es:

```
REPEAT
    instrucciones ...
UNTIL condición
END REPEAT;
```

La diferencia radica en cuando se ejecuta la comprobación de la condición, si antes de ejecutar las instrucciones contenidas o después. Con REPEAT se garantiza que las instrucciones se ejecuten al menos una vez, mientras que WHILE solo las ejecutará si se cumple la condición antes de empezar.

Más info: https://donnierock.com/2013/10/08/bucles-y-condicionales-en-procedimientos-almacenados-de-mysql/

1.5. Operadores

a) Operadores matemáticos

Operador	Significado
+	Suma
-	Resta
*	Producto
/	División
DIV	División entera
MOD (Dividendo, divisor)	Resto de la división entera
POW (Base, exponente)	Elevado a (base elevado a exponente)

b) Operadores de relación

Operador	Significado
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
=	Igual a
<>	Distinto de

c) Operadores lógicos

Operador	Significado
AND	Y lógico
OR	O lógico
NOT	Negación

d) Operador de concatenación

Operador	Significado
CONCAT (Valor1, Valor2, Valor3, ...)	Concatena los valores en una cadena

1.6. BD a utilizar

En los ejemplos del tema, usaremos esta base de datos de productos:

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

CREATE TABLE productos (
prodID  INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
nombre  VARCHAR(20) NOT NULL,
estado  VARCHAR(20) NOT NULL DEFAULT 'disponible',
coste   DECIMAL(10,2) NOT NULL DEFAULT 0.0,
precio  DECIMAL(10,2) NOT NULL DEFAULT 0.0
);

INSERT INTO productos (nombre, estado, coste, precio) VALUES
('Producto A','disponible', 4, 8),
('Producto B', 'disponible', 1, 1.5),
('Producto C', 'agotado', 50, 80);
```

2. Procedimientos almacenados

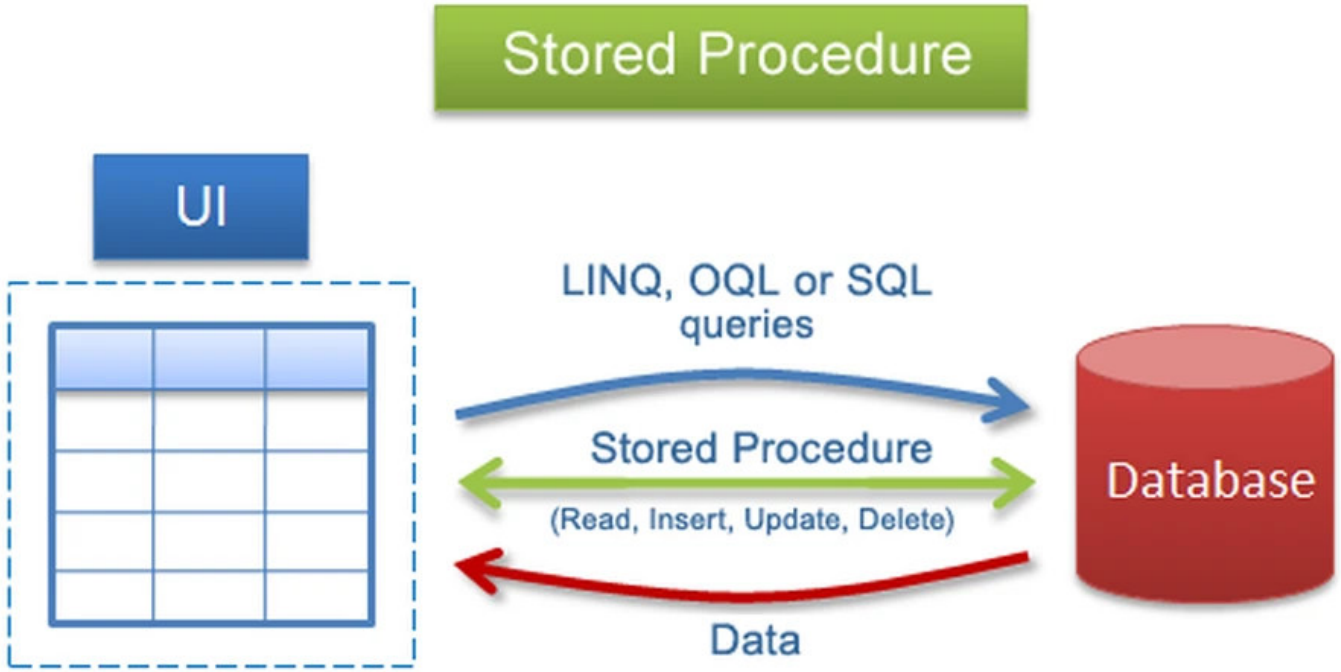
Un procedimiento almacenado MySQL no es más que una porción de código que puedes guardar y reutilizar. Ese código puede tener variables, bucles, iteraciones, etc., y es muy útil cuando repites la misma tarea muchas veces, siendo un buen método para encapsular el código. En Oracle, la sintaxis es muy similar.

En *scripts* de BBBD que encontrarás en la red o en la propia empresa en la que trabajes, es muy común encontrar estos comandos:

a) Bien un **DROP xxx IF EXISTS** antes de cada **CREATE xxx**.

b) Bien un **CREATE xxx IF NOT EXISTS** en lugar del **CREATE xxx**.

Esto es muy útil para poder ejecutar el *script* varias veces y se considera como una BUENA PRAXIS, sobre todo en ámbitos académicos y en entornos de prueba/desarrollo donde el "prueba y error" está a la orden del día.



Más info: https://www.cablenaranja.com/como-crear-y-usar-procedimientos-almacenados-en-mysql/

2.1. Sintaxis

La sintaxis de un procedimiento almacenado es la siguiente:

```
CREATE PROCEDURE [IF NOT EXISTS] nombre_procedimiento
([[IN|OUT|INOUT] nombre_parametro tipo_parametro,....])
BEGIN
    ... instrucciones;
END
```

Para **ejecutar un procedimiento** almacenado lo invocamos así:

```
CALL nombre_procedimiento (param1, param2, ...);
```

2.2. Parámetros

Los parámetros se definen separados por una coma y pueden ser de tres tipos:

- **Parámetro de entrada (IN)**. Es el tipo de parámetro que se usa por defecto. La aplicación o código que invoque al procedimiento tendrá que pasar un argumento para este parámetro. El procedimiento trabajará con una copia de su valor, teniendo el parámetro su valor original al terminar la ejecución del procedimiento.
- **Parámetro de salida (OUT)**. El valor de este parámetro puede ser cambiado en el procedimiento y, además, su valor modificado será enviado de vuelta al código o programa que invoca el procedimiento. Dentro del procedimiento el parámetro no empieza con su valor original.
- **Parámetro de entrada y salida (INOUT)**. Es una mezcla de los dos conceptos anteriores. La aplicación o código que invoca al procedimiento puede pasarle un valor a éste, devolviendo el valor modificado al terminar la ejecución. En caso de resultarte confuso, echa un vistazo al ejemplo que verás más adelante.

2.3. Ejemplos

2.3.1. Procedimientos con parámetros de entrada IN

Queremos obtener los productos de un determinado estado. Para ello, pasamos el nombre_estado como parámetro **IN (solo de entrada)**. Para parámetros IN, manipulamos el valor como si de un campo más de la tabla se tratara.

```
\! clear; -- Limpiar la consola (Linux)
\! cls; -- Limpiar la consola (Windows)
USE bd_productos;
```

Creamos el procedimiento:

```
DROP PROCEDURE IF EXISTS obtenerProductosPorEstado;
DELIMITER $$
CREATE PROCEDURE obtenerProductosPorEstado(
    IN nombre_estado VARCHAR(20)
)
BEGIN
    SELECT *
    FROM productos
    WHERE estado = nombre_estado;
END$$
DELIMITER ;
```

Vuelve al punto de (Variables y parámetros) si tienes dudas. Tenemos en este ejemplo:

- Un parámetro de entrada (tipo IN)
- Ninguna variable local
- Ninguna variable de usuario

Suponiendo que quieras obtener los productos con estado no disponible (agotado), tendrías que invocar al procedimiento de este modo:

```
CALL obtenerProductosPorEstado('agotado');
```

```
mysql> CALL obtenerProductosPorEstado('agotado');
+--------+------------+---------+-------+--------+
| prodID | nombre     | estado  | coste | precio |
+--------+------------+---------+-------+--------+
|      3 | Producto C | agotado | 50.00 |  80.00 |
+--------+------------+---------+-------+--------+
1 row in set (0,00 sec)
```

Si no te funciona, asegúrate de que usas la comilla SIMPLE RECTA.

2.3.2. Procedimientos con parámetros de salida OUT

Ahora queremos contar el número de productos que tienen un determinado estado y devolver ese contador en un **parámetro OUT (solo de salida)** llamado numero. Para parámetros OUT, usamos el comando INTO para grabar y devolver el dato que buscamos.

```
\! clear; -- Limpiar la consola (Linux)
USE bd_productos;
```

Creamos el procedimiento:

```
DROP PROCEDURE IF EXISTS contarProductosPorEstado;
DELIMITER $$
CREATE PROCEDURE contarProductosPorEstado(
    IN nombre_estado VARCHAR(20),
    OUT numero INTEGER)
BEGIN
    SELECT COUNT(prodID) INTO numero
    FROM productos
    WHERE estado = nombre_estado;
END$$
DELIMITER ;
```

Para ejecutarlo, pasamos ahora el "nombre_estado" definido como IN y la variable "numero" como parámetro OUT. Suponiendo que quieras obtener los productos “disponibles”, sería así:

-- Usamos una variable (@numero) para almacenar la salida. No es necesario declararla antes.

CALL contarProductosPorEstado('disponible', @numero);

-- Para mostrar en pantalla la variable @numero hacemos un SELECT de esa variable

SELECT @numero **AS** disponibles;

La variable de usuario @numero fuera, pasa a ser el parámetro numero dentro.

Vuelve al punto de (Variables y parámetros) si tienes dudas. Tenemos en este ejemplo:

- Dos parámetros, uno de entrada (tipo IN) y otro de salida (tipo OUT)
- Ninguna variable local
- Una variable de usuario (@numero)

```
mysql> CALL contarProductosPorEstado('disponible', @numero);
Query OK, 1 row affected (0,00 sec)

mysql> SELECT @numero AS disponibles;
+-------------+
| disponibles |
+-------------+
|           2 |
+-------------+
1 row in set (0,00 sec)
```

2.3.3. Procedimientos con parámetros de entrada/salida IN/OUT

Vamos a crear un procedimiento que incremente un parámetro **INOUT (entrada/salida)** llamado "total_ventas" cuando se vende un producto. Para parámetros INOUT, tenemos que usar los comandos INTO, DECLARE, y SET.

```
DROP PROCEDURE IF EXISTS venderProducto;
DELIMITER $$
CREATE PROCEDURE venderProducto(
    INOUT total_ventas DECIMAL(10,2),
    IN id_producto DECIMAL(10,2))
BEGIN
    DECLARE PVP DECIMAL(10,2);
    SELECT precio INTO PVP
    FROM productos
    WHERE prodID = id_producto;
   -- los parámetros no llevan @ y total_ventas es un parámetro
    SET total_ventas = total_ventas + PVP;
END$$
DELIMITER ;
```

Vamos a “vender” algunos productos para ver cómo cambia la variable:

```
SET @total_ventas = 0;
CALL venderProducto(@total_ventas, 1);
CALL venderProducto(@total_ventas, 2);
CALL venderProducto(@total_ventas, 2);
SELECT CONCAT(@total_ventas,' euros') AS ventas;
```

La variable de usuario @total_ventas fuera, pasa a ser el parámetro total_ventas dentro.

Vuelve al punto de (Variables y parámetros) si tienes dudas. Tenemos en este ejemplo:

- Dos parámetros, uno de entrada/salida (tipo INOUT) y otro de entrada (tipo

DECIMAL(10,2))

- Una variable local PVP (declarada dentro del procedimiento)
- Una variable de usuario (@total_ventas)

```
mysql> SET @total_ventas = 0;
Query OK, 0 rows affected (0,00 sec)

mysql> CALL venderProducto(@total_ventas, 1);
Query OK, 1 row affected (0,00 sec)

mysql> CALL venderProducto(@total_ventas, 2);
Query OK, 1 row affected (0,00 sec)

mysql> CALL venderProducto(@total_ventas, 2);
Query OK, 1 row affected (0,00 sec)

mysql> SELECT CONCAT(@total_ventas,' euros') AS ventas;
+-------------+
| ventas      |
+-------------+
| 11.00 euros |
+-------------+
1 row in set (0,00 sec)
```

2.3.4. Procedimientos que modifican datos

Cualquier PROGRAMA (procedimiento, función o trigger) puede modificar los datos de la BD. En este ejemplo pondremos los productos con **estado "agotado"** a **estado "en oferta"** y los dejaremos a precio de coste (precio=coste), mostrando el resultado. Fíjate que este procedimiento no tiene parámetros, aunque es mera casualidad.

```
DROP PROCEDURE IF EXISTS aplicarOfertasyMostrarlas;
DELIMITER $$
CREATE PROCEDURE aplicarOfertasyMostrarlas()
BEGIN
   UPDATE productos
   SET precio = coste, estado = 'en oferta'
   WHERE estado = 'agotado';
   SELECT *
   FROM productos
   WHERE estado = 'en oferta';
END$$
DELIMITER ;
```

Para modificar todos los productos, debemos invocar al procedimiento de este modo:

```
CALL aplicarOfertasyMostrarlas();
```

```
mysql> CALL aplicarOfertasyMostrarlas();
+--------+------------+-----------+-------+--------+
| prodID | nombre     | estado    | coste | precio |
+--------+------------+-----------+-------+--------+
|      3 | Producto C | en oferta | 50.00 |  50.00 |
+--------+------------+-----------+-------+--------+
1 row in set (0,01 sec)
```

VUELVE A EJECUTAR EL SCRIPT INICIAL PARA DEVOLVER LA BD A SU ESTADO INICIAL.

3. Funciones

Las funciones almacenadas de MySQL, o simplemente funciones, nos permiten procesar y manipular datos de forma procedural de un modo muy eficiente. Básicamente son idénticas a los procedimientos, pero **sus parámetros son de entrada (si tienen) y deben devolver un único valor**.

Como pasaba con los procedimientos, en Oracle la sintaxis es muy similar. De igual modo, no podrás definir una función dos veces y, en caso de que quieras redefinirla, tendrás que eliminarla con la sentencia DROP y luego volver a definirla.

A diferencia de los procedimientos, que solo se pueden llamar con CALL, las funciones pueden intercalarse en las sentencias SQL como si de una "función del sistema" se tratara, de la misma manera que hacemos con las conocidas funciones de agregado MAX(), SUM() o las de concatenación de cadenas CONCAT(), etc.

Las funciones pueden ser:

- **DETERMINISTAS**: si para unos mismos parámetros devuelven siempre el mismo valor.
- **NO DETERMINISTAS**: en caso contrario, siendo el caso NO DETERMINISTA más común el uso de funciones aleatorias.

> **Para evitar errores, debes indicar en cada función antes del BEGIN si ésta es DETERMINISTIC (99% de los casos) o NOT DETERMINISTIC**.

3.1. Sintaxis

La sintaxis de una función es la siguiente:

```
CREATE FUNCTION [IF NOT EXISTS] nombre_funcion
([ nombre_parametro tipo_parametro,... ])
RETURNS tipo_de_dato

[NOT] DETERMINISTIC
BEGIN
    ... instrucciones;

    RETURN valor_variable; -- el tipo debe coincidir con tipo_de_dato
END
```

Fíjate que incluimos las palabras reservadas **RETURNS** y **RETURN**:

- **RETURNS tipo_de_dato**: Declaración indicando que va a devolver un tipo "tipo_de_dato"
- **RETURN valor_variable**: Orden para devolver un valor o variable

3.2. Ejemplo

Vamos a crear una función que calcule el beneficio que se obtiene por cada producto, que se llamará calcularBeneficio. Esta función aceptará dos parámetros: el precio de compra (coste) y el precio de venta de un producto. El resultado de la función simplemente será la resta del precio de venta y el de compra, dando como resultado el beneficio obtenido con su venta.

```
\! clear; -- Limpiar la consola (Linux)
USE bd_productos;
```

Creamos la función:

```
DROP FUNCTION IF EXISTS calcularBeneficio;
DELIMITER $$
CREATE FUNCTION calcularBeneficio(
    p_coste DECIMAL(10,2),
    p_precio DECIMAL(10,2))
RETURNS  DECIMAL(10,2)
DETERMINISTIC
BEGIN
    DECLARE beneficio DECIMAL(10,2);
    SET beneficio = p_precio - p_coste;
    RETURN beneficio;
END$$
DELIMITER ;
```

Una vez hayas creado la función, podrás usarla directamente en cualquier consulta. A modo de ejemplo, vamos a ejecutar esta consulta sobre la tabla productos de la base de datos ejemplo:

```
SELECT *, calcularBeneficio(coste, precio) AS beneficio FROM productos;
```

En este ejemplo, son los propios campos de la tabla escritos en la llamada a la función (fuera) los que pasan a ser los parámetros p_coste, p_precio en el cuerpo de

la función (dentro).

Tenemos en este ejemplo:

- Dos parámetros, ambos de entrada (las funciones solo pueden tener de entrada)
- Una variable local beneficio (declarada dentro de la función)
- Dos campos de la tabla que se pasan a la función en su llamada

```
mysql> SELECT *, calcularBeneficio(coste, precio) AS beneficio FROM productos;
+--------+------------+------------+-------+--------+-----------+
| prodID | nombre     | estado     | coste | precio | beneficio |
+--------+------------+------------+-------+--------+-----------+
|      1 | Producto A | disponible |  4.00 |   8.00 |      4.00 |
|      2 | Producto B | disponible |  1.00 |   1.50 |      0.50 |
|      3 | Producto C | agotado    | 50.00 |  80.00 |     30.00 |
+--------+------------+------------+-------+--------+-----------+
3 rows in set (0,00 sec)
```

4. Ejemplos de funciones y procedimientos

En este apartado vamos a ver varios ejemplos prácticos con funciones y procedimientos.

4.1. Función "sumaDosNumeros"

Realiza una función que sume dos números reales y devuelva el resultado. Para el cuerpo de programa usa variables locales y, para la llamada, prueba con valores y variables de usuario. Los pasos que debemos seguir se indican a continuación:

- Primero comprobamos si existe una función con el mismo nombre y si existe la borramos. Hay que tener en cuenta que la primera vez que ejecutemos el *script*, como la función no existe nos mostrará un warning (aviso).
- Tenemos que indicar los delimitadores que utlizaremos para indicar donde termina la función o el procedimiento, tal como dijimos los más habituales son $$, //, ||. Aquí hemos elegido $$.
- Hay que cambiar el delimitador porque las instrucciones de las funciones y procedimientos deben terminar con punto y coma, con lo cual MySQL no sabría identificar cuando acaba la función o el procedimiento. Para cambiar el delimitador se emplea la palabra reservada DELIMITER. Además, al fnal, cuando haya terminado el *script* volveremos a colocar como delimitador el punto y coma para volver a la normalidad.
- Para ejecutar la función emplearemos una instrucción SELECT con el nombre de la función seguida de un paréntesis con los valores de los parámetros que deseemos utilizar.

Creamos la función:

```
DROP FUNCTION IF EXISTS sumaDosNumeros;
DELIMITER $$
CREATE FUNCTION sumaDosNumeros (
    num1 DECIMAL(10,2),
    num2 DECIMAL(10,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    DECLARE suma DECIMAL(10,2) DEFAULT 0;
    SET suma = num1 + num2;
    RETURN suma;
END$$
DELIMITER ;
```

La manera más sencilla de ejecutarla será poniendo directamente los valores:

```
SELECT sumaDosNumeros(3, 5);
```

```
mysql> SELECT sumaDosNumeros(3, 5);
+----------------------+
| sumaDosNumeros(3, 5) |
+----------------------+
|                 8.00 |
+----------------------+
1 row in set (0,00 sec)
```

Con VARIABLES DE USUARIO (recuerda que las LOCALES solo sirven dentro de un programa):

~~DECLARE a INT;~~ ~~DECLARE b INT;~~ ~~SET a=3;~~ ~~SET b=5;~~ ~~SELECT sumaDosNumeros(a, b);~~	SET @a=3; SET @b=5; SELECT sumaDosNumeros(@a, @b);

Prácticamente cualquier procedimiento se puede reescribir como una función y viceversa. ¿Cuál sería el procedimiento equivalente y cómo probar que funciona?

4.2. Procedimiento "esParImpar"

Realiza un procedimiento que reciba un número y nos diga si es par o impar. Para el cuerpo de programa usa variables de usuario y, para la llamada, prueba con valores y variables de usuario. Los pasos que debemos seguir se indican a continuación:

- Primero comprobamos si existe un procedimiento con el mismo nombre y si existe lo borramos.
- Igual que antes, establecemos el delimitador que vamos a utilizar y escribimos nuestro procedimiento. Ahora en los parámetros sí podemos indicar si son IN, OUT o INOUT.
- Para mostrar el resultado de la variable en pantalla utlizamos SELECT.
- Este procedimiento usa una variable de usuario, lo que no es habitual en estos casos (lo habitual seria usar un parametro de salida), esta se mantiene después de la ejecución del mismo.
- Para llamar al procedimiento utlizaremos, desde la línea de comandos, CALL seguido del nombre del procedimiento y los valores de los parámetros entre paréntesis.
- Después de llamarlo se puede seguir consultando el valor de @comoes con SELECT.

Creamos el procedimiento:

```
DROP PROCEDURE IF EXISTS esParImpar;
DELIMITER $$
CREATE PROCEDURE esParImpar (IN num INTEGER)
BEGIN
    IF MOD(num,2) = 0 THEN
        SET @comoes = 'PAR';
    ELSE
        SET @comoes = 'IMPAR';
    END IF;
    SELECT @comoes AS 'PAR o IMPAR';
END$$
DELIMITER ;
```

La manera más sencilla de ejecutarlo será poniendo directamente los valores:

> **CALL** esParImpar(3);

```
mysql> CALL esParImpar(3);
+-------------+
| PAR o IMPAR |
+-------------+
| IMPAR       |
+-------------+
1 row in set (0,00 sec)
```

Con VARIABLES DE USUARIO (recuerda que las LOCALES solo sirven dentro de un programa):

~~DECLARE a INT;~~ ~~SET a=3;~~ ~~CALL esParImpar(a);~~	SET @a=3; CALL esParImpar(a);

Tras ejecutar el procedimiento la primera vez, se ha creado la **variable de usuario @comoes**, se puede consultar su valor (e incluso modificarlo).

```
mysql> SELECT @comoes;
+---------+
| @comoes |
+---------+
| IMPAR   |
+---------+
1 row in set (0,00 sec)
```

4.3. Función "notaCalificacion"

Realiza una función que reciba una nota numérica **entera** entre 0 y 10 y devuelva si es un suspenso, suficiente, bien, etc. Para el cuerpo de programa usa variables locales y, para la llamada, prueba con valores y variables de usuario.

```
DROP FUNCTION IF EXISTS notaCalificacion;
DELIMITER $$
CREATE FUNCTION notaCalificacion(nota_num INTEGER)
RETURNS VARCHAR(15)
DETERMINISTIC
BEGIN
    DECLARE nota_texto VARCHAR(15);
    CASE
        WHEN nota_num BETWEEN 0 AND 4 THEN SET nota_texto = 'Insuficiente';
        WHEN nota_num = 5 THEN SET nota_texto = 'Suficiente';
        WHEN nota_num = 6 THEN SET nota_texto = 'Bien';
        WHEN nota_num BETWEEN 7 AND 8 THEN SET nota_texto = 'Notable';
        WHEN nota_num BETWEEN 9 AND 10 THEN SET nota_texto = 'Sobresaliente';
        ELSE SET nota_texto = '¡Nota incorrecta!';
    END CASE;
    RETURN nota_texto;
END$$
DELIMITER ;
```

La manera más sencilla de ejecutarla será poniendo directamente los valores:

```
SELECT notaCalificacion(4) AS 'CALIFICACIÓN';
```

```
mysql> SELECT notaCalificacion(4) AS 'CALIFICACIÓN';
+---------------+
| CALIFICACIÓN  |
+---------------+
| Insuficiente  |
+---------------+
1 row in set (0,00 sec)
```

Con VARIABLES DE USUARIO (recuerda que las LOCALES solo sirven dentro de un programa):

```
SET @a=4;
SELECT notaCalificacion(@a);
```

4.4. Función "notaRealCalificacion"

Realiza una función que reciba una nota numérica real (DECIMAL(10,2)) entre 0 y 10 y devuelva si es un suspenso, suficiente, bien, etc. Para el cuerpo de programa usa variables locales y, para la llamada, prueba con valores y variables de usuario. Utiliza **|| como delimitador**. Es la misma función que la anterior pero con la nota como un número real.

```
DROP FUNCTION IF EXISTS notaRealCalificacion;
DELIMITER ||
CREATE FUNCTION notaRealCalificacion(nota_num DECIMAL(10,2))
RETURNS VARCHAR(15)
DETERMINISTIC
BEGIN
    DECLARE nota_texto VARCHAR(15);
    CASE
        WHEN NOTA_NUM < 5 THEN SET NOTA_TEXTO = 'Insuficiente';
        WHEN NOTA_NUM < 6 THEN SET NOTA_TEXTO = 'Suficiente';
        WHEN NOTA_NUM < 9 THEN SET NOTA_TEXTO = 'Notable';
        WHEN NOTA_NUM <= 10 THEN SET NOTA_TEXTO = 'Sobresaliente';
        ELSE SET nota_texto = '¡Nota incorrecta!';
    END CASE;
    RETURN nota_texto;
END||
DELIMITER ;
```

La manera más sencilla de ejecutarla será poniendo directamente los valores:

```
SELECT notaRealCalificacion(4.9) AS 'CALIFICACIÓN';
```

```
mysql> SELECT notaRealCalificacion(4.9) AS 'CALIFICACIÓN';
+---------------+
| CALIFICACIÓN  |
+---------------+
| Insuficiente  |
+---------------+
1 row in set (0,00 sec)
```

Con VARIABLES DE USUARIO (recuerda que las LOCALES solo sirven dentro de un programa):

```
SET @a=4.9;
SELECT notaRealCalificacion(@a);
```

4.5. Procedimiento "mostrar1_N"

Realiza un procedimiento que reciba un número entero y muestre en pantalla los números desde el 1 hasta el número recibido incluido. Utiliza un **bucle WHILE** para este ejercicio. Para el cuerpo de programa usa variables de usuario y, para la llamada, prueba con valores y variables de usuario.

```
DROP PROCEDURE IF EXISTS mostrar1_N;
DELIMITER $$
CREATE PROCEDURE mostrar1_N (IN num INTEGER)
BEGIN
    SET @ii = 0;
    WHILE @ii < num DO
        SET @ii = @ii + 1;
        SELECT @ii AS VALOR;
    END WHILE;
END$$
DELIMITER ;
```

La manera más sencilla de ejecutarlo será poniendo directamente los valores:

> **CALL** mostrar1_N(3);

```
mysql> CALL mostrar1_N(3);
+-------+
| VALOR |
+-------+
|     1 |
+-------+
1 row in set (0,00 sec)

+-------+
| VALOR |
+-------+
|     2 |
+-------+
```

```
1 row in set (0,00 sec)

+-------+
| VALOR |
+-------+
|     3 |
+-------+
1 row in set (0,00 sec)
```

Con VARIABLES DE USUARIO (recuerda que las LOCALES solo sirven dentro de un programa):

```
SET @a=3;
CALL mostrar1_N(@a);
```

4.6. Procedimiento "mostrarImpares1_N"

Realiza un procedimiento que reciba un parámetro de tipo entero y muestre en pantalla los números impares entre el 1 y el número recibido. Para este ejercicio utilizaremos un **bucle REPEAT** y como **delimitador #**. Para el cuerpo de programa usa variables locales y, para la llamada, prueba con valores y variables de usuario.

```
DROP PROCEDURE IF EXISTS mostrarImpares1_N;
DELIMITER #
CREATE PROCEDURE mostrarImpares1_N (IN num INTEGER)
BEGIN
    DECLARE ii INTEGER DEFAULT 1;
    IF num >= 1 THEN
    REPEAT
        SELECT ii AS IMPAR;
        SET ii = ii+2;
    UNTIL ii > num
    END REPEAT;
END IF;
END#
DELIMITER ;
```

La manera más sencilla de ejecutarlo será poniendo directamente los valores:

> **CALL** mostrarImpares1_N(6);

```
mysql> CALL mostrarImpares1_N(6);
+-------+
| IMPAR |
+-------+
|     1 |
+-------+
1 row in set (0,01 sec)

+-------+
| IMPAR |
```

```
+-------+
|     3 |
+-------+
1 row in set (0,01 sec)

+-------+
| IMPAR |
+-------+
|     5 |
+-------+
1 row in set (0,01 sec)
```

Con VARIABLES DE USUARIO (recuerda que las LOCALES solo sirven dentro de un programa):

```
SET @a=6;
CALL mostrarImpares1_N(@a);
```

5. Triggers

Un TRIGGER o disparador es un objeto con nombre dentro de una base de datos el cual se asocia con una tabla y se activa cuando ocurre en ésta un evento en particular. A partir de MySQL 5.0.2 se incorporó el soporte BÁSICO para disparadores (*triggers*) y se mejoró con las versiones siguientes.

Los TRIGGER son **procedimientos que se ejecutarán según nuestras indicaciones cuando en una tabla ocurre un evento de tipo** actualización (**UPDATE**), inserción (**INSERT**) y borrado (**DELETE**).

Ventajas e inconvenientes de usar triggers:

a) VENTAJAS

- Con los triggers seremos capaces de validar todos aquellos valores que no pudieron ser validados mediante “constraints”, asegurando así la integridad de los datos.
- Los triggers nos permitirán ejecutar reglas de negocios.
- Utilizando la combinación de eventos podemos realizar acciones sumamente complejas.
- Los triggers nos permitirán llevar un control de los cambios realizados en una tabla. Para ello debemos apoyarnos en una segunda tabla (Comúnmente una tabla log).

b) INCONVENIENTES

- Los triggers, al ejecutarse de forma automática, pueden dificultar llevar un control sobre qué sentencias SQL fueron ejecutadas.
- Los triggers incrementan la sobrecarga del servidor. Un mal uso de triggers puede derivar en respuestas lentas por parte del servidor.

> **Más info:**
>
> Cómo crear y utilizar Triggers en MySQL. https://www.neoguias.com/como-crear-y-utilizar-triggers-en-mysql/
>
> Programador clic. MySQL-trigger. https://programmerclick.com/article/8726910429/

5.1. Sintaxis

Esta es la sintaxis de un trigger en MySQL:

```
CREATE TRIGGER [IF NOT EXISTS] 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
```

5.2. Ejemplo

Vamos a crear un trigger que actualice automáticamente el precio de los productos de la tabla "productos" cada vez que se actualice su coste. Le llamaremos actualizarPrecioProducto. El trigger comprobará si el coste del producto ha cambiado y, en caso afirmativo, establecerá el precio del producto con el doble del valor de su coste.

```
\! clear; -- Limpiar la consola (Linux)
USE bd_productos;
```

Creamos el trigger:

```
DROP TRIGGER IF EXISTS actualizarPrecioProducto;
DELIMITER $$
CREATE TRIGGER actualizarPrecioProducto
BEFORE UPDATE ON productos
FOR EACH ROW
BEGIN
  IF NEW.coste <> OLD.coste
    THEN
      SET NEW.precio = NEW.coste * 2;
  END IF;
END$$
DELIMITER ;
```

Lo que hacemos es crear un trigger que se ejecute antes de la actualización del registro, algo que indicamos con la sentencia “BEFORE UPDATE ON”. Luego comprobamos si el coste antiguo del producto difiere del nuevo y, si es así, actualizamos el precio con el doble del valor de su nuevo coste.

¿Cómo utilizar un trigger?

Una vez hayamos creado el trigger, no tendremos que hacer absolutamente nada para llamarlo, puesto que el motor de la base de datos lo invocará automáticamente cada vez que se actualice un registro de la tabla de productos.

Sin embargo, sí podemos comprobar el resultado del trigger actualizando un registro con una sentencia UPDATE como ésta:

```
UPDATE productos
SET coste = 5
WHERE prodID = 1;
SELECT * FROM productos;
```

Cuando se ejecuta la actualización con la sentencia UPDATE, se activa también el trigger, que actualizará el precio con el doble de su valor, según lo hemos definido. El resultado será:

```
mysql> SELECT * FROM productos;
+--------+------------+------------+-------+--------+
| prodID | nombre     | estado     | coste | precio |
+--------+------------+------------+-------+--------+
|      1 | Producto A | disponible |  5.00 |  10.00 |
|      2 | Producto B | disponible |  1.00 |   1.50 |
|      3 | Producto C | agotado    | 50.00 |  80.00 |
+--------+------------+------------+-------+--------+
3 rows in set (0,00 sec)
```

Se trata de un ejemplo muy básico del uso de un trigger, pero es bastante ilustrativo con fines introductorios. Los triggers pueden ser extremadamente complicados, pero en este módulo solo veremos los más sencillos.

5.3. Casos prácticos

Existen infinitas utilidades de los triggers, pero vamos a representar varios casos prácticos muy ilustrativos para demostrar su utilidad.

5.3.1. Validación de datos de entrada

> Aunque **la validación de los datos suele hacerse directamente en el aplicativo/formulario desde donde se leen los datos**, también podemos hacerla en la misma base de datos, mediante triggers. No es muy común, pero nos ayudará a entender su funcionamiento.

En este ejemplo, usamos SET para cambiar el valor de un campo. En concreto, antes de guardar cada producto, convertimos su nombre a mayúsculas (usando la función UPPER, que ya conocíamos), y guardamos 0 en vez del precio si el precio tiene un valor menor que cero.

Creamos el trigger:

```
DROP TRIGGER IF EXISTS validacionProducto;
DELIMITER $$
CREATE TRIGGER validacionProducto
BEFORE INSERT ON productos
FOR EACH ROW BEGIN
    SET NEW.nombre = UPPER(NEW.nombre);
    IF (NEW.precio < 0) THEN
        SET NEW.precio = 0;
    END IF;
END$$
DELIMITER ;
```

Si añadimos un producto que tenga un código en minúsculas y un precio menor que 0, y pedimos que se nos muestre el resultado, veremos ésto:

> **INSERT INTO** productos (nombre, precio) **VALUES** ('Pinzas de la cesta', -40);
>
> **SELECT** nombre, precio
> **FROM** productos
> **WHERE** nombre **LIKE** '%CESTA%';

```
mysql> INSERT INTO productos (nombre, precio) VALUES ('Pinzas de la cesta', -40);
Query OK, 1 row affected (0,00 sec)

mysql> SELECT nombre, precio
    -> FROM productos
    -> WHERE nombre LIKE '%CESTA%';
+--------------------+--------+
| nombre             | precio |
+--------------------+--------+
| PINZAS DE LA CESTA |   0.00 |
+--------------------+--------+
1 row in set (0,00 sec)
```

Se puede validar y transformar cualquier campo de una base de datos. Prueba a crear un nuevo trigger para no permitir aumentos de precios y modificar éste para no permitir precios con decimales.

5.3.2. Registro de cambios

Otra utilidad de los triggers, mucho más común que la anterior, consiste en hacer una copia de los datos sensibles a otra tabla cuando éstos son cambiados, a modo de registro. En este ejemplo, crearemos un disparador que se activa DESPUÉS de actualizar la tabla productos y, si el precio varía, inserta una tupla en una segunda tabla llamada productosCambiosPrecio.

Creamos la tabla para almacenar la copia:

```
CREATE TABLE IF NOT EXISTS productosCambiosPrecio (
cambioID          INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
productoID        INTEGER NOT NULL,
precioAnterior    DECIMAL(10,2) NOT NULL,
precioNuevo       DECIMAL(10,2) NOT NULL,
fecha             DATETIME NOT NULL
);
```

Creamos el trigger:

```
DROP TRIGGER IF EXISTS registrarCambioDePrecio;
DELIMITER $$
CREATE TRIGGER registrarCambioDePrecio
AFTER UPDATE ON productos
FOR EACH ROW
BEGIN
    IF (OLD.precio <> NEW.precio) THEN
        INSERT INTO productosCambiosPrecio (productoID, precioAnterior, precioNue
        VALUES (NEW.prodID, OLD.precio, NEW.precio, NOW());
    END IF;
END$$
DELIMITER ;
```

Si, mediante un UPDATE, cambiamos el precio del producto con idProducto = 1 de 8 a 9 y listamos la tabla de registro, obtendremos este resultado:

```
UPDATE productos
SET precio = 9
WHERE prodID = 1;

SELECT * FROM productosCambiosPrecio;
```

```
mysql> UPDATE productos
    -> SET precio = 9
    -> WHERE prodID = 1;
Query OK, 1 row affected (0,01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM productosCambiosPrecio;
+----------+------------+----------------+-------------+---------------------+
| cambioID | productoID | precioAnterior | precioNuevo | fecha               |
+----------+------------+----------------+-------------+---------------------+
|        1 |          1 |          10.00 |        9.00 | 2024-03-01 12:54:48 |
+----------+------------+----------------+-------------+---------------------+
1 row in set (0,00 sec)
```

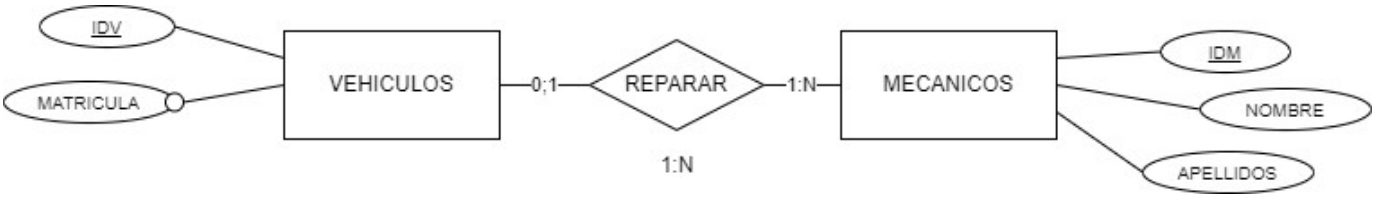
5.3.3. Participaciones mínimas 1:N en relaciones 1 a N

¿Recuerdas cuando vimos que no se podían representar las participaciones mínimas en una relación de cardinalidad 1:N? ¿Recuerdas que establecíamos una restricción de integridad? Pues bien, ya ha llegado el momento de implementar esa 1:N y solucionar el problema.

> **Este punto y el siguiente son de especial complejidad e importancia, y requieren que repases conceptos como cardinalidad, participación y pérdida semántica.**

Diagrama E-R

Usaremos este diagrama para crear una sencilla base de datos:



Este esquema de arriba (E-R) nos dice, entre otras cosas, que cada vehículo DEBE tener entre 1 y N mecánicos asociados y, cada mecánico puede tener o no un único vehículo asociado.

Modelo físico

Con lo que sabemos hasta hoy, esta sería la traducción más fiel posible, asumiendo que **hay una pérdida semántica en el lado derecho en el 1:N, es decir, que no se puede forzar a que todo vehículo tenga, como mínimo, un mecánico asociado**:

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

CREATE TABLE vehiculos (
idv         INTEGER PRIMARY KEY,
matricula   VARCHAR(20) NOT NULL
);
```

```
CREATE TABLE mecanicos (
idm         INTEGER PRIMARY KEY,
nombre      VARCHAR(20),
apellidos   VARCHAR(20),
idv         INTEGER,
FOREIGN KEY (idv) REFERENCES vehiculos(idv)
);

INSERT INTO vehiculos VALUES
(10, '3399BNJ'),
(11, '7879LOI'),
(12, '9985PIK');

INSERT INTO mecanicos VALUES
(20, 'Juan', 'García', 10),
(21, 'Luisa', 'Marín', 11),
(22, 'Eva', 'Zahora', 12),
(23, 'Leo', 'Lis', 10);
```

Comprobamos los datos:

```
mysql> SELECT * FROM mecanicos;
+-----+--------+-----------+------+
| idm | nombre | apellidos | idv  |
+-----+--------+-----------+------+
|  20 | Juan   | García    |   10 |
|  21 | Luisa  | Marín     |   11 |
|  22 | Eva    | Zahora    |   12 |
|  23 | Leo    | Lis       |   10 |
+-----+--------+-----------+------+
4 rows in set (0,00 sec)

mysql> SELECT * FROM vehiculos;
+-----+-----------+
| idv | matricula |
+-----+-----------+
|  10 | 3399BNJ   |
|  11 | 7879LOI   |
|  12 | 9985PIK   |
+-----+-----------+
3 rows in set (0,00 sec)
```

Podemos conseguir que -mediante un par de triggers- todo vehículo tenga uno o varios mecánicos controlando que cuando se ACTUALIZA o BORRA un mecánico que siempre haya uno como mínimo para cada vehículo.

Metodología a seguir

> **En relaciones 1:N seguiremos esta línea de actuación**:
>
> 1. No controlaremos las inserciones, ya que no podremos controlar que cuando se inserta un vehículo nuevo éste tenga ya un mecánico asociado porque no podemos incluir el id de vehículo como clave ajena en mecánicos si aún no hemos insertado el vehículo. Podríamos hacerlo, pero sería mediante transacciones y la cosa se complicaría MUCHO.
>
> 2. Crearemos DOS triggers que, antes del borrado y antes de la actualización de un mecánico, comprueben que se mantendrán las participaciones mínimas tras esas operaciones.

PASO 1: TRIGGER DE ANTES DEL BORRADO

Este trigger perseguirá proteger los borrados no deseados, es decir, que un mecánico sea borrado cuando es el único asociado a un determinado vehículo. ¿Cómo lo haremos? ANTES de cada borrado de un mecánico, **contaremos cuántos vehículos se quedarían sin mecánico si eliminamos ese mecánico**. Si es > 0 pararemos el borrado del mecánico.

En otras palabras:

> **IF** [contador(vehículos) **NOT IN** (lista de vehículos con mecánico sin ese mécánico) > 0]
>
> **THEN**
>
> Mostrar un error (que cancele el borrado).

Creamos el trigger:

Si intentamos borrar un mecánico asociado a un vehículo que no tiene otro mecánico:

DELETE FROM mecanicos WHERE idm = 22; -- debe fallar
DELETE FROM mecanicos WHERE idm = 23; -- debe funcionar

Nos dará este mensaje de error y abortará la operación:

***IMG

Necesitamos otro trigger para proteger las actualizaciones no deseadas, es decir, que un mecánico no pueda cambiar de vehículo si el vehículo que tiene asociado no tiene ya otro mecánico diferente asociado.

PASO 2: TRIGGER DE ANTES DE LA ACTUALIZACIÓN

Este trigger perseguirá proteger las actualizaciones no deseadas, es decir, que un mecánico que tiene asociado un vehículo no pueda cambiar por otro si el primer vehículo no tiene otro mecánico que pueda hacerse cargo de él. ¿Cómo lo haremos? ANTES de cada actualización de un mecánico, veremos si se ha modificado su vehículo asociado y, si ha sido el caso, **contaremos cuántos vehículos se quedarían sin mecánico (huérfanos) si actualizamos ese mecánico**.

SI

(se ha modificado el vehículo asociado) Y (vehículos quedarán huérfanos es > 0)

ENTONCES

pararemos la actualización del mecánico.

En otras palabras:

IF

[en el update estamos cambiando el vehículo asociado a ese mecánico]

AND

[contador(vehículos) NOT IN (lista de vehiculos con mecánico sin ese mécánico)>0]

THEN

Mostrar un error (que cancele la actualización)

Creamos el trigger:

Si intentamos cambiar el vehículo asociado a un mecánico que es el único para ese vehículo:

```
UPDATE mecanicos SET idv = 11 WHERE idm = 22; -- debe fallar
UPDATE mecanicos SET idv = 11 WHERE idm = 23; -- debe funcionar
```

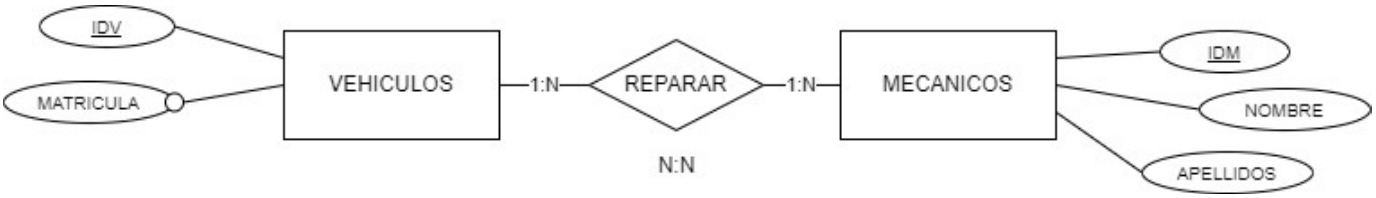
Nos dará este mensaje de error y abortará la operación:

Como acabamos de ver, las participaciones 1:N no se podían representar hasta ahora que hemos aprendido a usar los triggers. Del mismo modo, también podemos representar las participaciones 1:N si estuvieran a ambos lados de la relación anterior; es decir, que todo vehículo necesite un mecánico y viceversa.

5.3.4. Participaciones mínimas 1:N en relaciones N a N

Diagrama E-R

Usaremos este diagrama para crear una sencilla base de datos:



Este esquema de arriba (E-R) nos dice, entre otras cosas, que cada vehículo DEBE tener entre 1 y N mecánicos asociados y cada mecánico DEBE tener entre 1 y N vehículos asociados, **necesitando una tabla de cruce llamada REPARAR**.

Modelo físico

Con lo que sabemos hasta hoy, esta sería la traducción más fiel posible, asumiendo que **hay una pérdida semántica en ambos lados en el 1:N; es decir, que no se puede forzar a que todo vehículo tenga, como mínimo, un mecánico asociado y viceversa**:

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

CREATE TABLE vehiculos (
idv         INTEGER PRIMARY KEY,
matricula   VARCHAR(20) NOT NULL
);

CREATE TABLE mecanicos (
idm         INTEGER PRIMARY KEY,
nombre      VARCHAR(20),
apellidos   VARCHAR(20)
);

CREATE TABLE reparar(
idv INTEGER,
idm INTEGER,
```

```
PRIMARY KEY (idv, idm),
FOREIGN KEY (idv) REFERENCES vehiculos(idv),
FOREIGN KEY (idm) REFERENCES mecanicos(idm)
);

INSERT INTO vehiculos VALUES
(10, '3399BNJ'),
(11, '7879LOI'),
(12, '9985PIK');

INSERT INTO mecanicos VALUES
(20, 'Juan', 'García'),
(21, 'Luisa', 'Marín'),
(22, 'Eva', 'Zahora');

INSERT INTO reparar VALUES
(10, 20),
(11, 21),
(12, 22),
(10, 21);
```

En este caso, podemos conseguir que todo vehículo tenga uno o varios mecánicos y viceversa controlando cuando se ACTUALIZA o BORRA una fila de la tabla de cruce (reparar) para que siempre haya como mínimo un vehículo asociado a un mecánico y viceversa.

Metodología a seguir

> **En relaciones N:N seguiremos esta línea de actuación:**
>
> 1. No controlaremos las inserciones, ya que no pueden provocar problemas.
> 2. Crearemos UN trigger que, antes del borrado, comprueba que se mantienen las participaciones mínimas, de manera similar a como hemos hecho en el apartado anterior con las relaciones 1:N.
> 3. Crearemos UN trigger que, antes de la actualización, comprueba que se mantienen las participaciones mínimas, de manera similar a como hemos hecho en el apartado anterior con las relaciones 1:N.

PASO 1: TRIGGER DE ANTES DEL BORRADO

Este trigger perseguirá proteger los borrados no deseados, es decir, que una fila de la tabla de cruce "reparar" (que relaciona un mecánico con un vehículo) sea borrada cuando no hay más filas de ese vehículo o de ese mecánico. ¿Cómo lo haremos? ANTES de cada

borrado de una fila de la tabla de cruce (reparar) **contaremos cuántos vehículos se quedarían sin mecánico y cuántos mecánicos se quedarán sin vehículo si eliminamos esa fila**. Si ALGUNO de los dos contadores es > 0 pararemos el borrado de la fila.

> **SI**
> (vehículos quedarán huérfanos tras borrar esa fila es > 0)
> **O**
> (mecánicos quedarán huérfanos tras borrar esa fila es > 0)
> **ENTONCES**
> pararemos el borrado de la fila de la tabla de cruce (reparar)

En otras palabras:

> **IF**
> [contador(vehículos) **NOT IN** (lista de vehículos en tabla de cruce sin esa fila)>0]
> **OR**
> [contador(mecánicos) **NOT IN** (lista de mecánicos en tabla de cruce sin esa fila)>0]
> **THEN**
> Mostrar un error (que cancele el borrado).

Creamos el trigger:

Alternativamente, podemos contar las filas que hay en la tabla de cruce con esos mismos Ids que queremos eliminar. Si solo hay una fila que coincida con cada uno de los dos Ids, no permitiremos el borrado.

Si intentamos borrar un mecánico asociado a un vehículo que no tiene otro mecánico:

DELETE FROM mecanicos WHERE idm = 22; -- debe fallar
DELETE FROM mecanicos WHERE idm = 23; -- debe funcionar

Nos dará este mensaje de error y abortará la operación:

***IMG

Necesitamos otro trigger para proteger las actualizaciones no deseadas, es decir, que un mecánico no pueda cambiar de vehículo si el vehículo que tiene asociado no tiene ya otro mecánico diferente asociado.

PASO 2: TRIGGER DE ANTES DE LA ACTUALIZACIÓN

Este trigger perseguirá proteger las actualizaciones no deseadas, es decir, que un mecánico que tiene asociado un vehículo no pueda cambiar por otro si el primer vehículo no tiene otro mecánico que pueda hacerse cargo de él. ¿Cómo lo haremos? ANTES de cada actualización de un mecánico, veremos si se ha modificado su vehículo asociado y, si ha sido el caso, **contaremos cuántos vehículos se quedarían sin mecánico (huérfanos) si actualizamos ese mecánico**.

SI

(se ha modificado el vehículo asociado) Y (vehículos quedarán huérfanos es > 0)

ENTONCES

pararemos la actualización del mecánico.

En otras palabras:

IF

[en el update estamos cambiando el vehículo asociado a ese mecánico]

AND

[contador(vehículos) NOT IN (lista de vehiculos con mecánico sin ese mécánico)>0]

THEN

Mostrar un error (que cancele la actualización)

Creamos el trigger:

Si intentamos cambiar el vehículo asociado a un mecánico que es el único para ese vehículo:

```
UPDATE mecanicos SET idv = 11 WHERE idm = 22; -- debe fallar
UPDATE mecanicos SET idv = 11 WHERE idm = 23; -- debe funcionar
```

Nos dará este mensaje de error y abortará la operación:

Como acabamos de ver, las participaciones 1:N no se podían representar hasta ahora que hemos aprendido a usar los triggers. Del mismo modo, también podemos representar las participaciones 1:N si estuvieran a ambos lados de la relación anterior; es decir, que todo vehículo necesite un mecánico y viceversa.

5. Bibliografía

- MySQL 8.0 Reference Manual. Defining Stored Programs. https://dev.mysql.com/doc/refman/8.0/en/stored-programs-defining.html
- Tutorial de Triggers SQL con ejemplos sencillos. https://www.srcodigofuente.es/aprender-sql/triggers-sql?utm_content=cmp-true
- MySQL Tutorial. https://www.w3schools.com/mysql/
- Oracle Database Documentation. https://docs.oracle.com/en/database/oracle/oracle-database/index.html