CENTRE ESPECÍFIC
D´EDUCACIÓ A DISTÀNCIA DE
LA COMUNITAT VALENCIANA

UNIDAD 7

USUARIOS Y EXTENSIONES (MYSQL)

BASES DE DATOS 22/23
CFGS DAW

PARTE 2 DE 2. EXTENSIONES DE MYSQL

Revisado y ampliado por:
Abelardo Martínez y Pau Miñana
Autor:
Sergio Badal

Licencia Creative Commons

Reconocimiento – NoComercial – CompartirIgual (by-nc-sa): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

ÍNDICE DE CONTENIDO

1. Extensiones MySql (Programas)....3
1.1 Mi primer programa....4
1.2 Modificar un programa ya creado....6
1.3 Ventajas e inconvenientes de su uso....6
1.4 Delimitadores....7
1.5 Variables y parámetros....9
1.6 Estructuras de control....10
1.6.1 Alternativas....10
1.6.2 Repetitivas....10
1.7 Operadores....11
1.8 Permisos....11
2. Procedimientos almacenados....12
2.1 Sintaxis....12
2.2 Parámetros....13
2.3 Casos prácticos de procedimientos....14
2.3.1 Procedimientos con parámetros de entrada IN....14
2.3.2 Procedimientos con parámetros de salida OUT....15
2.3.3 Procedimientos con parámetros de entrada/salida IN/OUT....16
2.3.4 Procedimientos que modifican datos....17
3. Funciones....18
4. Ejemplos de funciones y procedimientos....20
4.1 Ejemplo 1: función SUMA....20
4.2 Ejemplo 2: procedimiento PAR_IMPAR....21
4.3 Ejemplo 3: función NOTA ENTERA....22
4.4 Ejemplo 4: función NOTA REAL....23
4.5 Ejemplo 5: procedimiento del 1 al N....24
4.6 Ejemplo 6: procedimiento IMPARES....25
5. Triggers....26
5.1 Cómo crear un trigger....27
5.2 Cómo utilizar un trigger....28
5.3 Casos prácticos de triggers....28
5.3.1 Validación de datos de entrada....29
5.3.2 Registro de cambios....30
5.3.3 Participaciones mínimas 1:N en relaciones 1 a N....31
5.3.4 Participaciones mínimas 1:N en relaciones N a N....35
5.4 Cómo ver los triggers existentes....40
5.5 Ventajas e inconvenientes de usar triggers....41
6. Cursores....41

UD7.2. EXTENSIONES EN MYSQL

1. EXTENSIONES MYSQL (PROGRAMAS)

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:

- **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.

Usaremos esta base de datos:

```
DROP DATABASE IF EXISTS bd_productos;
CREATE DATABASE bd_productos;
USE bd_productos;
CREATE TABLE productos (
   id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   nombre VARCHAR(20) NOT NULL,
   estado VARCHAR(25) 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);
```

1.1 Mi primer programa

Veamos con un ejemplo muy sencillo la forma de un **PROGRAMA** -en este caso un procedimiento almacenado- para intentar deducir de manera intuitiva qué pretendemos conseguir.

```
USE bd_productos;
DROP PROCEDURE IF EXISTS obtenerProductosPorEstado;
CREATE PROCEDURE obtenerProductosPorEstado(
    IN nombre_estado VARCHAR(25)
)
BEGIN
  SELECT *
  FROM productos
  WHERE estado = nombre_estado;
END

CALL obtenerProductosPorEstado ('agotado');
```

En el ejemplo anterior tenemos un PROGRAMA (procedimiento en este caso) que recibe un parámetro de entrada, el nombre de estado , y construye una consulta en función de ese dato.

No lo intentes ejecutar porque no funcionará (por eso en rojo). Luego veremos por qué.

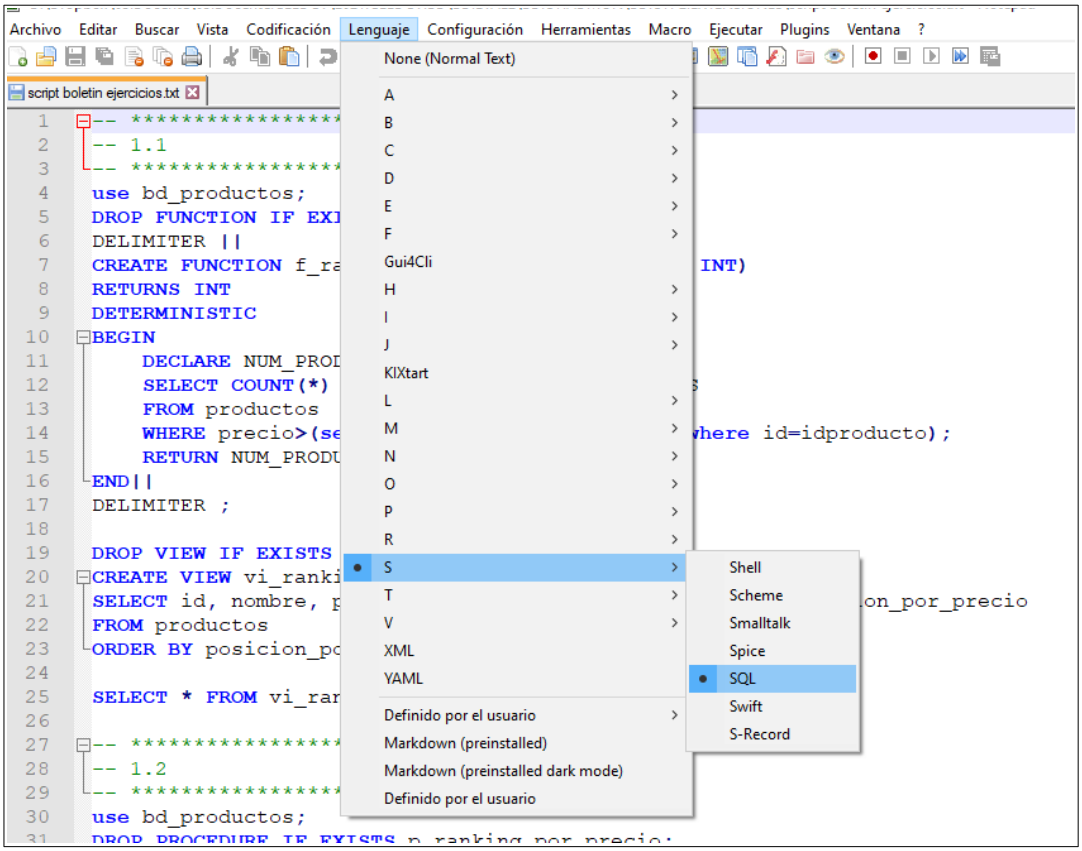
Te recomendamos que uses un EDITOR GRÁFICO que te ayudará a colorear el código.

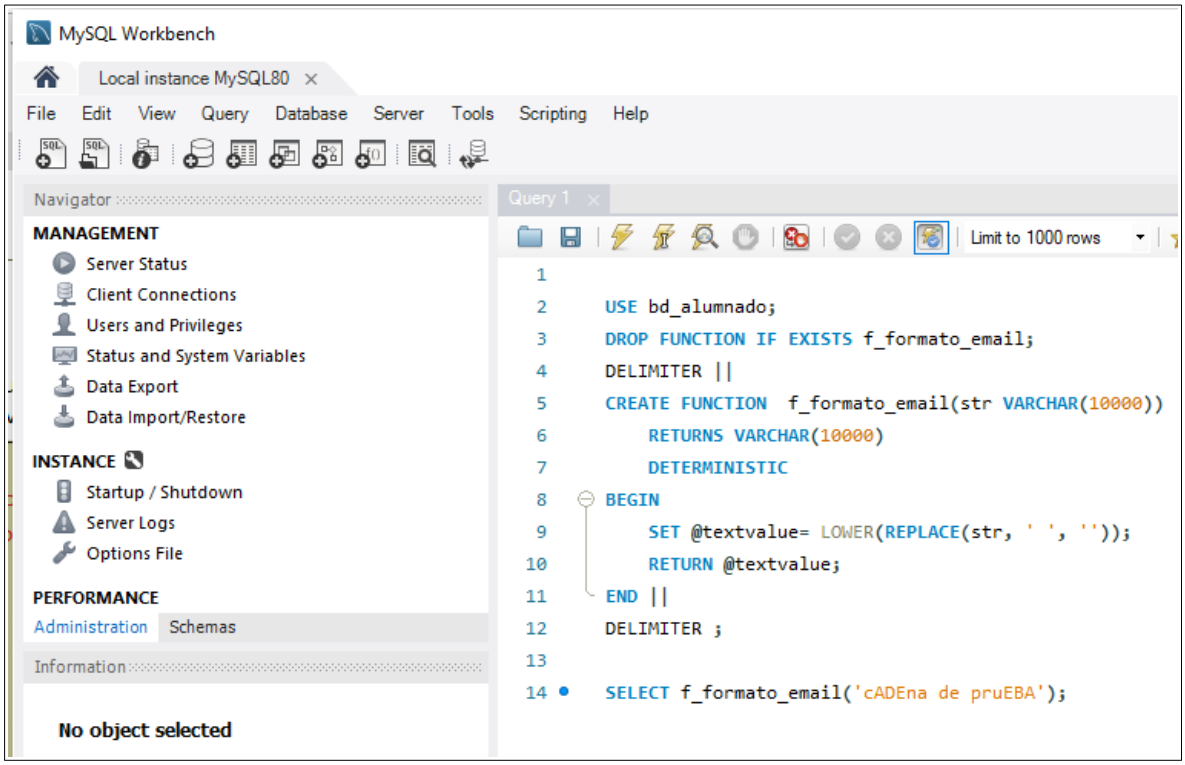
Puedes usar un **editor de código genérico** como Notepad++, indicando en el menú superior LANGUAGE → SQL o bien usar un **gestor de bases de datos MySQL** como WorkBench, Navicat, Heidi, etc. En Linux también disponemos de la alternativa Notepadqq, que es un clon de Notepad++.

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: '

Tienes una captura de **Notepad++** y de **WorkBench** en la siguiente página.


Archivo Editar Buscar Vista Codificación Lenguaje Configuración Herramientas Macro Ejecutar Plugins Ventana ?
script boletin ejercicios.txt
None (Normal Text)
A
B
C
D
E
F
Gui4Cli
H
I
J
KIXtart
L
M
N
O
P
R
S
T
V
XML
YAML
Definido por el usuario
Markdown (preinstalled)
Markdown (preinstalled dark mode)
Definido por el usuario
Shell
Scheme
Smalltalk
Spice
SQL
Swift
S-Record
use bd_productos;
DROP FUNCTION IF EXI
DELIMITER ||
CREATE FUNCTION f_ra
RETURNS INT
DETERMINISTIC
BEGIN
DECLARE NUM_PROD
SELECT COUNT(*)
FROM productos
WHERE precio>(se
RETURN NUM_PRODU
END||
DELIMITER ;
DROP VIEW IF EXISTS
CREATE VIEW vi_ranki
SELECT id, nombre, p
FROM productos
ORDER BY posicion_pc
SELECT * FROM vi_ran
use bd_productos;
INT)
where id=idproducto);
on_por_precio


MySQL Workbench
Local instance MySQL80
File Edit View Query Database Server Tools Scripting Help
Navigator
MANAGEMENT
Server Status
Client Connections
Users and Privileges
Status and System Variables
Data Export
Data Import/Restore
INSTANCE
Startup / Shutdown
Server Logs
Options File
PERFORMANCE
Administration Schemas
Information
No object selected
Query 1
Limit to 1000 rows
USE bd_alumnado;
DROP FUNCTION IF EXISTS f_formato_email;
DELIMITER ||
CREATE FUNCTION f_formato_email(str VARCHAR(10000))
RETURNS VARCHAR(10000)
DETERMINISTIC
BEGIN
SET @textvalue= LOWER(REPLACE(str, ' ', ''));
RETURN @textvalue;
END ||
DELIMITER ;
SELECT f_formato_email('cADEna de pruEBA');

1.2 Modificar un programa ya creado

En MySQL no está contemplada la opción de modificar un **PROGRAMA** por lo que, si quieres hacer cambios sobre un **PROGRAMA** tendrás que eliminarlo y crearlo de nuevo o bien, incluir la sentencia "DROP xxx IF EXISTS" de la que te hablamos al empezar este documento.

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

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

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

1.3 Ventajas e inconvenientes de su uso

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

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.

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.

1.4 Delimitadores

Al definir los programas en MySQL tendremos que usar delimitadores para indicar al SGBD que se trata de un bloque independiente. En los siguientes ejemplos, **DELIMITER $$** frena la ejecución de MySQL, que se retomará de nuevo en la sentencia **DELIMITER ;** del final.

El delimitador puede ser también //, aunque el más común es $$.

 IMPORTANTE

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 ($$).

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/

Aquí tienes el ejemplo anterior con delimitadores:

```
\! clear; -- LIMPIAR LA CONSOLA (Linux)
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_productos;
DROP PROCEDURE IF EXISTS  obtenerProductosPorEstado;
DELIMITER $$
CREATE PROCEDURE obtenerProductosPorEstado(
   IN nombre_estado VARCHAR(25)
)
BEGIN
   SELECT *
   FROM productos
   WHERE estado = nombre_estado;
END$$
DELIMITER ;

CALL obtenerProductosPorEstado ('agotado');
```

Prueba a ejecutar ese código en la consola o en el WorkBench.

¿No te funciona? Asegúrate de que usas la comilla SIMPLE RECTA. ÉSTA NO: ‘ ÉSTA SÍ: '

```
mysql> USE bd_productos;
Database changed
mysql> DROP PROCEDURE IF EXISTS  obtenerProductosPorEstado;
Query OK, 0 rows affected (0.01 sec)

mysql> DELIMITER $$
mysql> CREATE PROCEDURE obtenerProductosPorEstado(
    ->     IN nombre_estado VARCHAR(25)
    -> )
    -> BEGIN
    ->     SELECT *
    ->     FROM productos
    ->     WHERE estado = nombre_estado;
    -> END$$
Query OK, 0 rows affected (0.00 sec)

mysql> DELIMITER ;
mysql>
mysql> CALL obtenerProductosPorEstado ('agotado');
+----+-------------+---------+-------+--------+
| id | nombre      | estado  | coste | precio |
+----+-------------+---------+-------+--------+
|  3 | Melocotones | agotado |     5 |      8 |
|  4 | Kiwis       | agotado |     2 |      6 |
|  7 | Moras       | agotado |    10 |     20 |
+----+-------------+---------+-------+--------+
3 rows in set (0.01 sec)

Query OK, 0 rows affected, 1 warning (0.03 sec)
```

Query 1

Limit to 1000 rows

```
1  USE bd_productos;
2  DROP PROCEDURE IF EXISTS  obtenerProductosPorEstado;
3  DELIMITER $$
4  CREATE PROCEDURE obtenerProductosPorEstado(
5      IN nombre_estado VARCHAR(25)
6  )
7  BEGIN
8      SELECT *
9      FROM productos
10     WHERE estado = nombre_estado;
11 END$$
12 DELIMITER ;
13
14 CALL obtenerProductosPorEstado ('agotado');
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	nombre	estado	coste	precio
3	Melocotones	agotado	5	8
4	Kiwis	agotado	2	6
7	Moras	agotado	10	20

1.5 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 los entenderemos mejor con los ejemplos que vienen a continuación, pero como adelanto pueden ser:

1. **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 **@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;

2. **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;

3. **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;

	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		

1.6 Estructuras de control

1.6.1 Alternativas

Estas instrucciones representan qué hacer en función de una expresión o una condición:

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

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

```
CASE [expresión]
WHEN {valor1}
      THEN bloque_1
WHEN {valor2}
      THEN bloque_2
…
[ELSE
      bloque_defecto]
END CASE;
```

```
CASE
WHEN {condicion1}
      THEN bloque_1
WHEN {condicion2}
      THEN bloque_2
…
[ELSE
      bloque_por_defecto]
END CASE;
```

1.6.2 Repetitivas

El bucle *WHILE* realizará las instrucciones que contiene, mientras la condición sea verdadera y el bucle *REPEAT* realizará las instrucciones que contiene hasta que la condición sea verdadera.

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

```
REPEAT
instrucciones …
UNTIL condición
END REPEAT;
```

```
WHILE condición DO
instrucciones …
END WHILE;
```

1.7 Operadores

Operadores matemáticos MySQL	
+	Suma
-	Resta
*	Producto
/	División
DIV	Division entera
MOD(Dividendo,Divisor)	Resto de la división entera.
POW(base,exponente)	Elevado a (base elevado a exponente)

Operadores de relación MySQL.	
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
=	Igual a
<>	Distinto de (también se puede emplear ¡=)

Operador de concatenación MySQL.	
CONCAT(VALOR1,VALOR2,VALOR3,...)	Concatena los valores en una cadena.

Operadores lógicos MySQL.	
AND	Y lógico
OR	O lógico
NOT	Negación.

1.8 Permisos

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.

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.

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

 CONSEJO

Como acabas de ver más arriba, 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.

En caso del **CREATE PROCEDURE** que veremos en este tema no podemos hacer un **CREATE PROCEDURE IF NOT EXISTS** (da error de sintaxis), por lo que haremos un **DROP PROCEDURE IF EXISTS** antes de cada **CREATE PROCEDURE**. Lo mismo sucederá para las funciones con **CREATE FUNCTION IF NOT EXISTS**

2.1 Sintaxis

La sintaxis de un procedimiento almacenado es la siguiente:

```
CREATE PROCEDURE nombre
(param1,
...
 paramN)
BEGIN
        sentencia1;
        ...
        sentenciaN;
END
```

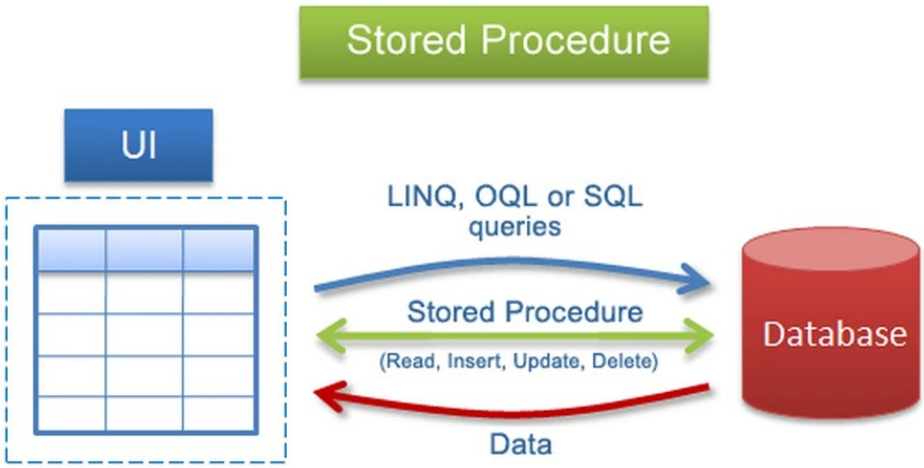
Para ejecutar un procedimiento almacenado lo invocamos así:

```
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 parámetros puede ser cambiado en el procedimiento, y además su valor modificado será enviado de vuelta al código o programa que invoca el procedimiento.
- **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 ojo al ejemplo que verás más adelante.



2.3 Casos prácticos de procedimientos

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.

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_productos;
DROP PROCEDURE IF EXISTS obtenerProductosPorEstado;
DELIMITER $$
CREATE PROCEDURE obtenerProductosPorEstado(
        IN nombre_estado VARCHAR(25)
)
BEGIN
   SELECT *
   FROM productos
   WHERE estado = nombre_estado;
END$$
DELIMITER ;
```

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');
+----+------------+---------+--------+
| id | nombre     | estado  | precio |
+----+------------+---------+--------+
|  3 | Producto C | agotado |     80 |
+----+------------+---------+--------+
1 row in set (0,00 sec)
```

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

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

¿No te funciona? Asegúrate de que usas la comilla SIMPLE RECTA. ÉSTA NO: ‘ ÉSTA SÍ: '

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.

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_productos;
DROP PROCEDURE IF EXISTS  contarProductosPorEstado;
DELIMITER $$
CREATE PROCEDURE contarProductosPorEstado(
   IN nombre_estado VARCHAR(25),
   OUT numero INT)
BEGIN
   SELECT count(id) INTO  numero
   FROM productos
   WHERE estado = nombre_estado;
END$$
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 (@numero1) para almacenar la salida. No es necesario declararla antes.
CALL contarProductosPorEstado('disponible', @numero1);
-- Para mostrar en pantalla la variable @numero1 hacemos un SELECT de esa variable
SELECT @numero1 AS disponibles;
```

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

mysql> SELECT @numero AS disponibles;
+-------------+
| disponibles |
+-------------+
|           2 |
+-------------+
```

La variable de usuario @numero1 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 (@numero1)

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

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.

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_productos;
DROP PROCEDURE IF EXISTS  venderProducto;
DELIMITER $$
CREATE PROCEDURE venderProducto(
   INOUT total_ventas DECIMAL(10,2),
   IN id_producto INT)
BEGIN
   DECLARE PVP DECIMAL(10,2);
   SELECT precio INTO PVP
   FROM productos
   WHERE id = id_producto;
  -- los parámetros no llevan @ y total_ventas es un parámetro
   SET total_ventas = total_ventas + PVP;
END$$
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;
```

Vemos el resultado en la siguiente página.

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)

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

¿No te funciona? Asegúrate de que usas la comilla SIMPLE RECTA. ÉSTA NO: ‘ ÉSTA SÍ: '

```
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 @total_ventas;
+---------------+
| @total_ventas |
+---------------+
|            12 |
+---------------+
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.

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_productos;
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, tendrías que invocar al procedimiento de este modo:

```
CALL aplicarOfertasyMostrarlas();
```

¿No te funciona? Asegúrate de que usas la comilla SIMPLE RECTA. ÉSTA NO: ‘ ÉSTA SÍ: '

```
mysql> CALL aplicarOfertasyMostrarlas();
+----+-------------+-----------+-------+--------+
| id | nombre      | estado    | coste | precio |
+----+-------------+-----------+-------+--------+
|  3 | Melocotones | en oferta |     5 |      5 |
|  4 | Kiwis       | en oferta |     2 |      2 |
|  7 | Moras       | en oferta |    10 |     10 |
+----+-------------+-----------+-------+--------+
3 rows in set (0.01 sec)

Query OK, 0 rows affected (0.03 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 guardar el resultado en una variable mediante SET o 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 y 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 NON DETERMINISTIC.

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.

Fíjate que incluimos las palabras reservadas RETURNS y RETURN:

- **RETURNS INT**: Declaración indicando que va a devolver un entero
- **RETURN 5:** Orden para devolver un valor o variable

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_productos;
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 base_ejemplo:

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

```
mysql> SELECT *, calcularBeneficio(coste, precio) AS beneficio FROM productos;
+----+------------+------------+-------+--------+-----------+
| id | nombre     | estado     | coste | precio | beneficio |
+----+------------+------------+-------+--------+-----------+
|  1 | Producto A | disponible |     4 |      8 |      4.00 |
|  2 | Producto B | disponible |     1 |    1.5 |      0.50 |
|  3 | Producto C | agotado    |    50 |     80 |     30.00 |
+----+------------+------------+-------+--------+-----------+
3 rows in set (0.00 sec)
```

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

Más info: https://www.neoguias.com/funciones-almacenadas-mysql/

 CONSEJO

Para ver el resultado de una función directamente por pantalla se puede llamar también a un SELECT sin FROM. Por ejemplo **SELECT** calcularBeneficio(20, 30) **AS** beneficio;

4. EJEMPLOS DE FUNCIONES Y PROCEDIMIENTOS

4.1 Ejemplo 1: función SUMA

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.

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
DROP FUNCTION IF EXISTS SUMA_DOS_NUMEROS;
DELIMITER $$
CREATE FUNCTION SUMA_DOS_NUMEROS (N1 DECIMAL(10,2), N2 DECIMAL(10,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    DECLARE SUMA DECIMAL(10,2) DEFAULT 0;
    SET SUMA = N1 + N2;
    RETURN SUMA;
END$$
DELIMITER ;
```

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

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

```
mysql> select suma_dos_numeros(3,5);
+-----------------------+
| suma_dos_numeros(3,5) |
+-----------------------+
|                     8 |
+-----------------------+
1 row in set (0.00 sec)
```

Con VARIABLES DE USUARIO(*):

```
DECLARE a INT;
DECLARE b INT;
SET a=3;
SET b=5;
SELECT SUMA_DOS_NUMEROS(a, b);
```

```
SET @a=3;
SET @b=5;
SELECT SUMA_DOS_NUMEROS(@a, @b);
```

(*) Recuerda que las LOCALES solo sirven dentro de un programa

Prácticamente cualquier procedimiento se puede reescribir como una función y viceversa. ¿Te atreves a conseguir el procedimiento equivalente y cómo probar que funciona?

¿Cómo lo harías?

4.2 Ejemplo 2: procedimiento PAR_IMPAR

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.

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
DROP PROCEDURE IF EXISTS PAR_IMPAR;
DELIMITER $$
CREATE PROCEDURE PAR_IMPAR (IN NUM INT)
BEGIN
        IF MOD(NUM,2) = 0 THEN
                SET @COMOES='PAR';
        ELSE
                SET @COMOES='IMPAR';
        END IF;
        SELECT @COMOES AS 'PAR O IMPAR';
END$$
DELIMITER ;
```

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

```
CALL PAR_IMPAR(3);
```

```
mysql> CALL PAR_IMPAR(3);
+-------------+
| PAR O IMPAR |
+-------------+
| IMPAR       |
+-------------+
1 row in set (0.00 sec)
```

Con VARIABLES DE USUARIO(*):

```
DECLARE a INT;
SET a=3;
CALL PAR_IMPAR(a);
```

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

Prácticamente cualquier procedimiento se puede reescribir como una función y viceversa. ¿Te atreves a conseguir la función equivalente y cómo probar que funciona?

¿Cómo lo harías?

4.3 Ejemplo 3: función NOTA ENTERA

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.

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
DROP FUNCTION IF EXISTS NOTA;
DELIMITER $$
CREATE FUNCTION NOTA(NOTA_NUM INT)
RETURNS VARCHAR(15)
DETERMINISTIC
BEGIN
	DECLARE NOTA_TEXTO VARCHAR(15);
	CASE
		WHEN NOTA_NUM BETWEEN 0 AND 4 THEN SET NOTA_TEXTO = 'Insuficiente';
		WHEN NOTA_NUM = 5 THEN SET NOTA_TEXTO = 'Suficiente';
		WHEN NOTA_NUM = 6 THEN SET NOTA_TEXTO = 'Bien';
		WHEN NOTA_NUM BETWEEN 7 AND 8 THEN SET NOTA_TEXTO = 'Notable';
		WHEN NOTA_NUM BETWEEN 9 AND 10 THEN SET NOTA_TEXTO = 'Sobresaliente';
		ELSE SET NOTA_TEXTO = 'Nota incorrecta!';
	END CASE;
	RETURN NOTA_TEXTO;
END$$
DELIMITER ;
```

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

```
SELECT NOTA(4) AS NOTA;
```

```
mysql> SELECT NOTA(4) AS NOTA;
+--------------+
| NOTA         |
+--------------+
| Insuficiente |
+--------------+
1 row in set (0.00 sec)
```

Con VARIABLES DE USUARIO:

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

Todo procedimiento se puede reescribir como una función y viceversa. ¿Te atreves a sacar el procedimiento equivalente y probarlo? **¿Cómo lo harías?**

4.4 Ejemplo 4: función NOTA REAL

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 un *CASE WHEN* y ||como delimitador.

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
DROP FUNCTION IF EXISTS NOTA;
DELIMITER ||
CREATE FUNCTION NOTA(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 NOTA(4.9) AS NOTA;
```

```
mysql> SELECT NOTA(4) AS NOTA;
+--------------+
| NOTA         |
+--------------+
| Insuficiente |
+--------------+
1 row in set (0.00 sec)
```

Con VARIABLES DE USUARIO:

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

Todo procedimiento se puede reescribir como una función y viceversa. ¿Te atreves a sacar el procedimiento equivalente y probarlo? **¿Cómo lo harías?**

4.5 Ejemplo 5: procedimiento del 1 al 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.

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
DROP PROCEDURE IF EXISTS DEL_1_AL_N;
DELIMITER $$
CREATE PROCEDURE DEL_1_AL_N (IN N INT)
BEGIN
    SET @temp=0;
    WHILE @temp < N DO
        SET @temp = @temp+1;
        SELECT @temp AS VALOR;
    END WHILE;
END$$
DELIMITER ;
```

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

```
CALL DEL_1_AL_N(10);
```

Con VARIABLES DE USUARIO:

```
SET @a=10;
CALL DEL_1_AL_N(@a);
```

```
mysql> CALL DEL_1_AL_N(3);
+------+
| I    |
+------+
|    1 |
+------+
1 row in set (0.01 sec)

+------+
| I    |
+------+
|    2 |
+------+
1 row in set (0.01 sec)

+------+
| I    |
+------+
|    3 |
+------+
1 row in set (0.01 sec)

+------+
| I    |
+------+
|    4 |
+------+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)
```

4.6 Ejemplo 6: procedimiento IMPARES

Ahora vamos a realizar un procedimiento que recibe un parámetro de tipo entero y muestra 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.

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
DROP PROCEDURE IF EXISTS IMPARES_DEL_1_AL_N;
DELIMITER #
CREATE PROCEDURE IMPARES_DEL_1_AL_N (IN N INT)
BEGIN
	DECLARE iter INT DEFAULT 1;
	IF N >= 1 THEN
	REPEAT
		SELECT iter;
		SET iter = iter+2;
	UNTIL iter > N
	END REPEAT;
END IF;
END#
DELIMITER ;
```

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

```
CALL IMPARES_DEL_1_AL_N(6);
```

Con VARIABLES DE USUARIO:

```
SET @a=6;
CALL IMPARES_DEL_1_AL_N(@a);
```

```
mysql> CALL IMPARES_DEL_1_AL_N(6);
+------+
| I    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)

+------+
| I    |
+------+
|    3 |
+------+
1 row in set (0.00 sec)

+------+
| I    |
+------+
|    5 |
+------+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)
```

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. En Oracle, existen importantes diferencias de sintaxis.

Fuente1: https://manuales.guebs.com/MySql-5.0/triggers.html

Fuente2: https://www.neoguias.com/como-crear-y-utilizar-triggers-en-mysql/

Fuente3: https://programmerclick.com/article/8726910429/

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
```

 CONSEJO

A diferencia de lo que pasaba con los procedimientos y las funciones, para los triggers sí que podemos hacer un **CREATE TRIGGER IF NOT EXISTS**, por lo que el DROP previo es opcional.

5.1 Cómo crear un trigger

Las siguientes sentencias crean una tabla y un trigger para sentencias INSERT dentro de la tabla. El trigger suma los valores insertados en una de las columnas de la tabla.

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_productos;
CREATE TABLE tabla_cuenta (num_cuenta INT, cantidad INT);
CREATE TRIGGER insertar_suma
BEFORE INSERT ON tabla_cuenta
FOR EACH ROW SET @sum = @sum + NEW.cantidad;
```

 IMPORTANTE

En este caso, como el cuerpo del programa solo tiene una sentencia (FOR EACH...) no son necesarios el BEGIN/END ni los delimitadores.

En este otro ejemplo, vamos a crear un trigger que actualice automáticamente el precio de los productos de la tabla creada en la PRIMERA PÁGINA DE ESTE DOCUMENTO 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. Para crear el trigger, ejecutamos esto:

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_productos;
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 ;
```

Esta vez hemos usado otro DELIMITER, para variar un poco. Ese nuevo delimitador puede ser casi cualquiera, siempre y cuando no sea algo que aparezca en una orden habitual.

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. Un negocio redondo :-)

5.2 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 esta:

```
UPDATE productos SET coste = 5 WHERE id = 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. Este será el resultado:

```
mysql> SELECT * FROM productos;
+----+------------+------------+-------+--------+
| id | nombre     | estado     | coste | precio |
+----+------------+------------+-------+--------+
|  1 | Producto A | disponible |     5 |     10 |
|  2 | Producto B | disponible |     1 |    1.5 |
|  3 | Producto C | agotado    |    50 |     80 |
+----+------------+------------+-------+--------+
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 de triggers

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.

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_productos;
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 ;
```

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.

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 (0.01 sec)

mysql> SELECT nombre, precio FROM productos WHERE nombre LIKE '%CESTA%';
+--------------------+--------+
| nombre             | precio |
+--------------------+--------+
| PINZAS DE LA CESTA |      0 |
+--------------------+--------+
```

Se puede validar y transformar cualquier campo de una base de datos. ¿Te atreves a crear un nuevo trigger para no permitir aumentos de precios y modificar éste para no permitir precios con decimales? **¿Cómo lo harías?**

5.3.2 Registro de cambios

Otra utilidad de los triggers, esta 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.

Creamos esta tabla para tales efectos y el siguiente trigger:

```
CREATE TABLE IF NOT EXISTS productosCambiosPrecio (
  idCambio INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  idProducto INT NOT NULL,
  precioAnterior DECIMAL(10,2) NOT NULL,
  precioNuevo DECIMAL(10,2) NOT NULL,
  fecha DATETIME NOT NULL);
```

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_productos;
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 (idProducto, precioAnterior, precioNuevo, fecha)
        VALUES (NEW.id, OLD.precio, NEW.precio, NOW());
    END IF;
END||
DELIMITER ;
```

En este ejemplo, hemos creado 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;

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:

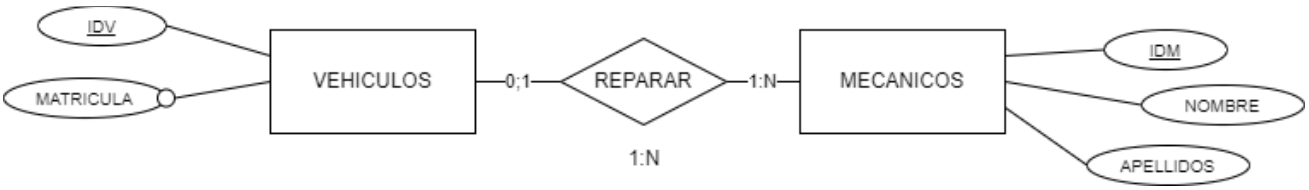
```
mysql> SELECT * FROM productosCambiosPrecio;
+----------+------------+----------------+-------------+---------------------+
| idCambio | idProducto | precioAnterior | precioNuevo | fecha               |
+----------+------------+----------------+-------------+---------------------+
|        1 |          2 |              8 |           9 | 2022-03-11 19:15:57 |
+----------+------------+----------------+-------------+---------------------+
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.

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

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.

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**:

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
DROP DATABASE IF EXISTS bd_taller_mecanico;
CREATE DATABASE bd_taller_mecanico;
USE bd_taller_mecanico;
CREATE TABLE vehiculos(
    idv INT PRIMARY KEY,
    matricula VARCHAR(20) NOT NULL);
CREATE TABLE mecanicos(
    idm INT PRIMARY KEY,
    nombre VARCHAR(20),
    apellidos VARCHAR(20),
    idv INT,
    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);
```

```
mysql> select * from mecanicos;
+-----+--------+-----------+------+
| idm | nombre | apellidos | idv  |
+-----+--------+-----------+------+
|  20 | Juan   | García    |   10 |
|  21 | Luisa  | Marín     |   11 |
|  22 | Eva    | Zahora    |   12 |
+-----+--------+-----------+------+
3 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)
```

Pues bien, ¡mediante un par de triggers SÍ que podemos conseguirlo!

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

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 este 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 vehiculos con mecánico sin ese mécánico)>0]
THEN
    Mostrar un error (que cancele el borrado).
```

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_taller_mecanico;
DROP TRIGGER  IF EXISTS borradoDeMecanicos;
DELIMITER ||
CREATE TRIGGER borradoDeMecanicos
BEFORE DELETE ON mecanicos
FOR EACH ROW
BEGIN
IF ((SELECT count(*) FROM vehiculos
    WHERE idv NOT IN (SELECT idv FROM mecanicos WHERE idm<>OLD.idm))>0)
  THEN
      SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'No se puede borrar este mecánico. Todo
vehículo debe tener, como mínimo, un mecánico asociado.';
  END IF;
END||
DELIMITER ;
```

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:

```
mysql> DELETE FROM mecanicos WHERE idm=22;
ERROR 1644 (45000): No se puede borrar este mecánico. Todo vehículo debe tener, como
 mínimo, un mecánico asociado.
```

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.

Piensa en cómo lo harías antes de continuar leyendo...

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)

Éste podría ser un trigger válido:

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_taller_mecanico;
DROP TRIGGER  IF EXISTS actualizacionDeMecanicos;
DELIMITER ||
CREATE TRIGGER actualizacionDeMecanicos
BEFORE UPDATE ON mecanicos
FOR EACH ROW
BEGIN
IF (
        (OLD.idv<> NEW.idv)
AND
        (       SELECT count(*)
                FROM vehiculos
                WHERE idv NOT IN (SELECT idv FROM mecanicos WHERE idm<>OLD.idm)
        )>0
)
  THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'No se puede actualizar este mecánico.
Todo vehículo debe tener, como mínimo, un mecánico asociado.';
  END IF;
END||
DELIMITER ;
```

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:

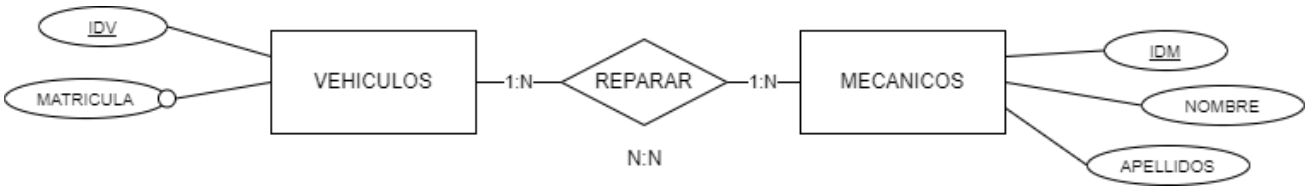
```
mysql> UPDATE mecanicos SET idv=11 WHERE idm=22;
ERROR 1644 (45000): No se puede actualizar este mecánico. Todo vehículo debe tener, como
mínimo, un mecánico asociado.
```

Como acabamos de ver, las participaciones 1:N no se podían representar hasta ahora que hemos aprendido a usar los triggers. ¿Crees que podríamos representar las participaciones 1:N si estuvieran a ambos lados de la relación anterior? Es decir, que todo vehículo necesite un mecánico y viceversa.

¿Cómo lo harías?

5.3.4 Participaciones mínimas 1:N en relaciones N a N

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**.

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**:

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
DROP DATABASE IF EXISTS bd_taller_mecanico;
CREATE DATABASE bd_taller_mecanico;
USE bd_taller_mecanico;
CREATE TABLE vehiculos(
   idv INT PRIMARY KEY,
   matricula VARCHAR(20) NOT NULL);
CREATE TABLE mecanicos(
   idm INT PRIMARY KEY,
   nombre VARCHAR(20),
   apellidos VARCHAR(20)
);
CREATE TABLE reparar(
   idv INT,
   idm INT,
   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.

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 vehículos se quedarán sin mecánico 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 vehiculos en tabla de cruce sin esa fila)>0]

OR

[contador(mecanicos) NOT IN (lista de mecanicos en tabla de cruce sin esa fila)>0]

THEN

Mostrar un error (que cancele el borrado).

Haciendo necesario crear este trigger:

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_taller_mecanico;
DROP TRIGGER  IF EXISTS borradoDeReparar;
DELIMITER ||
CREATE TRIGGER  borradoDeReparar
BEFORE DELETE ON reparar
FOR EACH ROW
BEGIN
IF (
        (
                SELECT count(*)
                FROM vehiculos
WHERE idv NOT IN (SELECT idv FROM reparar WHERE NOT (idm=OLD.idm AND idv=OLD.idv))>0
        )
        OR
        (
                SELECT count(*)
                FROM mecanicos
WHERE idm NOT IN (SELECT idm FROM reparar WHERE NOT (idm=OLD.idm AND idv=OLD .idv))>0
        )
)
  THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'No se puede borrar ESTA fila de la tabla
reparar por restricciones 1:N en ambos extremos';
  END IF;
END||
DELIMITER ;
```

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.

Lo vemos en la página siguiente:

Haciendo necesario crear este trigger:

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_taller_mecanico;
DROP TRIGGER  IF EXISTS borradoDeReparar;
DELIMITER ||
CREATE TRIGGER  borradoDeReparar
BEFORE DELETE ON reparar
FOR EACH ROW
BEGIN
IF
(SELECT COUNT(*) FROM reparar WHERE idv = OLD.idv) = 1
OR
(SELECT COUNT(*) FROM reparar WHERE idm = OLD.idm) = 1
  THEN
      SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'No se puede borrar ESTA fila de la tabla reparar por restricciones 1:N en ambos extremos';
  END IF;
END||
DELIMITER ;
```

En ambas soluciones, si intentamos borrar una fila de la tabla reparar que asocia a un vehículo con un mecánico, de manera que si la borramos quedan mecánicos o vehículos huérfanos… pasará esto:

```
DELETE FROM reparar WHERE idv=12 AND idm=22; -- debe fallar

DELETE FROM reparar WHERE idv=10 AND idm=21; --  debe funcionar
```

La primera sentencia, nos dará este mensaje de error y abortará la operación:

```
mysql> DELETE FROM reparar WHERE idv=12 AND idm=22;
ERROR 1644 (45000): No se puede borrar ESTA fila de la tabla reparar por
 restricciones 1:N en ambos extremos
```

PASO 2: TRIGGER DE ANTES DE LA ACTUALIZACIÓN

Este trigger perseguirá proteger las actualizaciones no deseadas, es decir, que una fila de la tabla de cruce reparar (que relaciona un mecánico con un vehículo) haga cambios que dejen a algún vehículo o a algún mecánico huérfano.

Como vimos antes, debido a la complejidad del trigger necesario para conseguir ese objetivo, decidimos (en relaciones N:N) simplemente impedir cualquier UPDATE en la tabla de cruce.

¿Cómo lo haremos? ANTES de cada actualización de una fila de la tabla de cruce (reparar) mostraremos un mensaje de error que detendrá la operación.

Haciendo necesario crear este trigger:

```
\! cls; -- LIMPIAR LA CONSOLA (Windows)
USE bd_taller_mecanico;
DROP TRIGGER  IF EXISTS actualizacionDeReparar;
DELIMITER ||
CREATE TRIGGER   actualizacionDeReparar
BEFORE UPDATE ON reparar
FOR EACH ROW
BEGIN
IF (NEW.idv <> OLD.idv)
THEN
        IF (      (SELECT COUNT(*) FROM reparar WHERE idv = OLD.idv)=1
        )
        THEN
                SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'No se puede actualizar esta tabla
al no poder garantizar las restricciones 1:N en ambos extremos';
        END IF;
END IF;

IF (NEW.idm <> OLD.idm)
THEN
        IF (      (SELECT COUNT(*) FROM reparar WHERE idm = OLD.idm)=1
        )
        THEN
                SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'No se puede actualizar esta tabla
al no poder garantizar las restricciones 1:N en ambos extremos';
        END IF;
END IF;
END||
DELIMITER ;
```

Si intentamos borrar una fila de la tabla reparar que asocia a un vehículo con un mecánico, de manera que si la borramos quedan mecánicos o vehículos huérfanos, pasará esto:

```
UPDATE reparar SET idm=22  WHERE idv=12; -- debe fallar
```

Cualquier UPDATE sobre la tabla de cruce nos dará este mensaje de error y abortará la operación:

```
mysql> UPDATE reparar SET idm=22  WHERE idv=12;
ERROR 1644 (45000): No se puede actualizar esta tabla al no poder garantizar
las restricciones 1:N en ambos extremos
mysql>
```

5.4 Cómo ver los triggers existentes

Para ver información básica de los disparadores que ya existen en la base de datos usaremos:

```
SHOW TRIGGERS;
```

En MySQL, todas las definiciones de disparadores se almacenan en la tabla de disparadores de la base de datos. La declaración de consulta para verlos es la siguiente.

```
SELECT trigger_name FROM information_schema.triggers;
```

```
mysql> select trigger_name
    -> from information_schema.triggers;
+----------------------------+
| TRIGGER_NAME               |
+----------------------------+
| sys_config_insert_set_user |
| sys_config_update_set_user |
| media                      |
| actualizarPrecioProducto   |
| validacionProducto         |
| borradoDeMecanicos         |
| actualizacionDeMecanicos   |
+----------------------------+
7 rows in set (0.00 sec)
```

5.5 Ventajas e inconvenientes de usar triggers

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 trigger nos permitirán llevar un control de los cambios realizados en una tabla. Para ello nos debemos de apoyar de una segunda tabla (Comúnmente una tabla log).

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.

6. CURSORES

En base de datos, un Cursor es un mecanismo que nos permite procesar fila por fila el resultado de una consulta.

Realmente son bucles que sirven para recorrer el resultado de una consulta (*query*) y guardar dicho resultado en variables o hacer operaciones con otras tablas. Para poder crear cursores hay que saber primero cómo crear procedimientos y funciones almacenados en MySql, ya que en MySql se crean dentro de las funciones y procedimientos almacenados.

No veremos cursores en este curso, pero te invitamos a que conozcas más sobre ellos en estos enlaces:

Más info: https://blogprog.gonzalolopez.es/articulos/crear-cursores-en-mysql.html

Video: https://www.youtube.com/watch?v=KQg4WSB57f8