

Procedimientos - Funciones - Cursores - Triggers.

Introducción.....	1
Variables de usuario.....	2
Procedimientos.....	4
Variables.....	7
Variables locales.....	7
Variables globales.....	8
Parámetros.....	9
Instrucciones condicionales.....	11
IF-THEN-ELSE.....	11
CASE.....	12
Instrucciones repetitivas o bucles.....	13
LOOP.....	13
REPEAT.....	14
WHILE.....	14
Bloques de instrucciones.....	15
Funciones.....	16
Cursores.....	17
Gestión de errores.....	18
Triggers.....	19
Crear Trigger.....	20
Con workbench.....	21
Ejercicios.....	24

Introducción.

Podemos definir un procedimiento como un conjunto de ordenes MySQL agrupadas bajo el nombre del procedimiento y cuando se llama a este desde cualquier lugar del servidos se ejecutan esas instrucciones.

Para poder trabajar con procedimientos almacenados se ha de tener privilegios de: CREATE PROCEDURE, ALTER PROCEDURE (si fue el autor, lo tiene por defecto) y EXECUTE.

El uso de procedimientos puede justificarse por distintos motivos:

- Labores administrativas. Trabajos de comprobación, seguimientos o registros de información que sean rutinarios y relevante para el administrador.
- Control de acceso a la información. El acceso a los datos a través de procedimientos ofrece las siguientes ventajas:
 - Seguridad: El usuario no tiene acceso directo a las tablas. No tiene por qué conocer la estructura: ni tipos de datos, ni claves foráneas, nombre de los atributos,.. Solo puede acceder a la información a través de los procedimientos establecidos por el administrador.
 - Rendimiento: Las ordenes incluidas en un procedimiento almacenado, están compiladas por lo que la ejecución es más rápida que si se realizan desde el interprete de comandos.
 - Comodidad: El procedimiento contendrá una serie de instrucciones que resuelven algún problema o realizan alguna tarea, que habrá que realizar en los momentos que sean oportunos. Al tenerlas almacenadas solo llamando al procedimiento se ejecutarán todas de forma rápida y precisa.

En MySQL, podemos identificar como rutinas a procedimientos y a funciones. Ambos tienen en común la realización de un conjunto de ordenes al ser invocados. La diferencia entre ellos:

- Una función: FUNCTION, devuelve un valor y se puede invocar desde dentro de un comando. Ejemplo:
`mysql>SELECT nombre_funcion();`
- Un procedimiento: PROCEDURE, no devuelve ningún valor y se invoca con la instrucción CALL. Ejemplo:
`mysql>CALL nombre_procedimiento();`

Variables de usuario.

En el trabajo diario puede ser interesante almacenar algún resultado parcial en una variable, de forma que se utilice en otra expresión y nos facilite la sintaxis de los comandos.

Ejemplo: Basándonos en la base de datos: concesionariodb, supongamos que queremos obtener un listado de todos los vehículos que tienen un precio hora superior a la media:

```
mysql>SET @media = (SELECT AVG(preciodia) FROM coche);
mysql>SELECT * FROM coche WHERE preciodia>=@media;
mysql>SELECT @media;
```

Frente al comando SELECT que requiere una subconsulta, y ambas estrategias nos ofrecen el mismo resultado:

```
mysql> SELECT * FROM coche WHERE preciodia>=
      (SELECT AVG(preciodia) FROM coche);
```

```
mysql> select * from coche;
+-----+-----+-----+-----+-----+
| matricula | modelo | color | marca | preciodia |
+-----+-----+-----+-----+-----+
| 1111-AA   | Zafira | Azul  | Opel  | 25         |
| 2222ABB   | Corolla | Blanco | Toyota | 30         |
| 2233ABK   | Corsa  | Blanco | Opel  | 20         |
| 2763AFF   | Corsa  | Beig  | Opel  | 20         |
| 3333GTA   | Picasso | Blanco | Citroen | 35         |
| 4114GTA   | Picasso | Rojo  | Citroen | 35         |
| 962DSF    | Yaris  | Verde | Toyota | 25         |
| 972GDA    | Rav-4  | Verde | Toyota | 40         |
| 9963KFF   | Vectra | Beig  | Opel  | 40         |
+-----+-----+-----+-----+-----+
9 rows in set (0.00 sec)

mysql> SET @media = (SELECT AVG(preciodia) FROM coche);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM coche WHERE preciodia>=@media;
+-----+-----+-----+-----+-----+
| matricula | modelo | color | marca | preciodia |
+-----+-----+-----+-----+-----+
| 2222ABB   | Corolla | Blanco | Toyota | 30         |
| 3333GTA   | Picasso | Blanco | Citroen | 35         |
| 4114GTA   | Picasso | Rojo  | Citroen | 35         |
| 972GDA    | Rav-4  | Verde | Toyota | 40         |
| 9963KFF   | Vectra | Beig  | Opel  | 40         |
+-----+-----+-----+-----+-----+
5 rows in set (0.01 sec)

mysql> select @media;
+-----+
| @media |
+-----+
| 30.000000000 |
+-----+
1 row in set (0.00 sec)
```

Las variables de usuario se pueden incluir en ordenes como: UPDATE, INSERT, DELETE,..

Para nombrar correctamente una variable debemos seguir las siguientes reglas:

- Empezar por @.
- Se pueden utilizar letras y números.
- Se pueden usar símbolos incluido el espacio en blanco: “.”, “_”, “\$”.
- Otros símbolos incluido el espacio en blanco, pero requiere entrecomillar a la variable: @'mi vble', @'mi-vble',..
- No distingue entre mayúsculas y minúsculas.
- Longitud máxima del nombre 64 caracteres.

Para asignar un valor a una variable se usa el comando SET:

```
mysql>SET @variable=valor;
```

valor: Puede ser número, carácter, resultado de una función o de un SELECT.

Se pueden asignar el valor a más de una variable en el mismo comando SET:

```
mysql>SET @vbl1=val1, @vbl2=val2, @vbl3=val3;
```

Las variables se pueden usar como valores de entrada de un registro en las sentencias INSERT:

```
mysql>INSERT INTO nombre_tabla VALUES(@vbl1, @vbl2, @vbl3);
```

Para mostrar el contenido de una variable se usa el comando SELECT:

```
mysql>SELECT @variable;
```

Las variables de usuario no se declaran ni se define el tipo de dato que va a contener. Según se le asigne un valor adquiere el tipo de dato de ese valor. Si posteriormente se le asigna otro tipo de dato adquiere ese nuevo tipo de dato.

Si no se le asigna ningún valor estaríamos hablando de una variable nueva y por defecto su contenido es NULL. Por tanto, si por error al nombrar una variable, es decir, que se omita, repita o permuten caracteres, MySQL lo interpretará como otra variable y utilizará el valor NULL en el lugar donde se esté utilizando.

También podemos asignar un valor o varios valores a una o varias variables en una misma operación SELECT utilizando la cláusula INTO. Para ello ha de coincidir el número de variables a asignar con el número de columnas que muestre SELECT y se asignarán en el mismo orden.

Ejemplo:

```
mysql> SELECT marca, modelo, matricula INTO @marca,@modelo,@matricula
      FROM coche WHERE matricula LIKE '%GDA';
```

```
mysql> SELECT marca, modelo, matricula INTO @marca, @modelo, @matricula
      FROM coche WHERE matricula LIKE '%GDA';
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @marca, @modelo, @matricula;
+-----+-----+-----+
| @marca | @modelo | @matricula |
+-----+-----+-----+
| Toyota | Rav-4   | 972GDA     |
+-----+-----+-----+
1 row in set (0.00 sec)
```

Si la orden SELECT ofrece más de una línea o registro nos dará un error, pues aparecerían más de un valor para cada variable. Una posibilidad de limitar a una línea la respuesta de SELECT es utilizar la cláusula “LIMIT 1”. (Recordamos que LIMIT permite limitar el número de líneas que devuelve SELECT. En su sintaxis también permite indicar a partir de que línea mostraría. Ejemplo: LIMIT 5 OFFSET 20: indica que muestre sólo 5 líneas siendo la primera la número 20).

Si por otra parte la sentencia SELECT no devuelve ningún valor, las variables quedan asignadas con NULL.

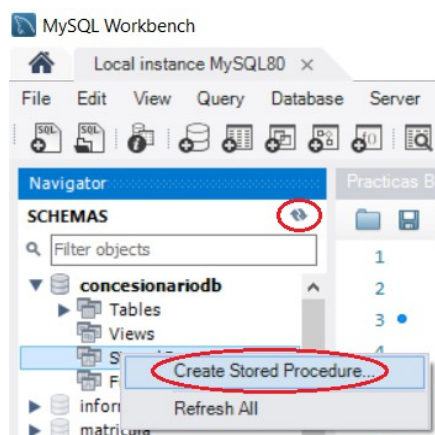
Las variables se pueden usar dentro de expresiones aritméticas (+, -, *, /, mod), de carácter (CONCAT_WS, TRIM, LEFT,..), de relación (=, <=, >=, <, >, !=). Así mismo, los resultados de esas expresiones se pueden almacenar en otras variables.

Procedimientos.

Los procedimientos están asociados a una base de datos. En concreto a la base de datos que esté activa en el momento de crearlo. Para crear un procedimiento podemos proceder de la siguiente forma:

- Escribirlo en un script con un editor de texto plano. El fichero en el que se almacene ha de tener extensión .sql
- Entrar en el servidor MySQL activar una base de datos en la que vamos a asociar el procedimiento y ejecutar el script con “source”.

Desde workbench, activamos una base de datos (que aparezca en negrita):



Una vez creado es necesario refrescar workbench.

El nombre de un procedimiento puede ser cualquier cadena de caracteres incluidos los números. No hay regla al respecto, pero si es bueno seguir algún método o estilo a la hora de nombrarlos. Por ejemplo: Objeto_Acción, siendo “Objeto” el objeto sobre el que va a actuar el procedimiento y “Acción” que es lo que hará.

La sintaxis, tanto para procedimiento como para función es similar (versión 5.7, véase página oficial):

```

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

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

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

func_parameter:
    param_name type

type:
    Any valid MySQL data type

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

routine_body:
    Valid SQL routine statement

```

El cuerpo de procedimiento ha de tener como mínimo: BEGIN y END.

Ejemplo:

```

CREATE PROCEDURE ejemplo1()
BEGIN
    SELECT "Hola mundo.";
    SELECT "Mi primer procedimiento del curso";
END

```

El procedimiento ha de ejecutarse como una única instrucción, por lo que al encontrarse el primer “;” considera fin de instrucción y empieza a compilar, esto nos daría error. La sintaxis correcta de un procedimiento requiere terminar con END.

Para evitar este problemas hay que definir de alguna forma el indicador de fin de comando para que se compile o traduzca completo (hasta el END). Esto se hace con DELIMITER. Una vez terminado el procedimiento debemos de volver a poner el delimitador de comando a “;”.

Ejemplo:

```

DELIMITER $$
DROP PROCEDURE IF EXISTS holamundo $$
CREATE PROCEDURE holamundo()
BEGIN
    SELECT "Hola mundo.";
    SELECT "Mi primer procedimiento del curso";
END $$

DELIMITER ;

```

Al igual que los comandos de crear base de datos o una tabla tenemos disponible la cláusula IF NOT EXISTS.

Para ver el listado de los procedimientos y funciones disponibles, se pueden consultar:

En versión mysql 5.7.

```
mysql>select name, type y db from mysql.proc;
```

```
mysql> select name, type, db from proc;
```

name	type	db
infreco	PROCEDURE	botanicadb
recolectado	FUNCTION	botanicadb
holamundo	PROCEDURE	dallasdb
calificacion	PROCEDURE	empleadodb
cambio	FUNCTION	empleadodb
dia	PROCEDURE	empleadodb
dpto	PROCEDURE	empleadodb

En mysql 8.0.

```
mysql>select routine_name, routine_type, routine_schema  
from routines.information_schema;
```

```
mysql> select routine_name, routine_type, routine_schema from routines order by routine_schema;
```

ROUTINE_NAME	ROUTINE_TYPE	ROUTINE_SCHEMA
prueba4	PROCEDURE	matricula
hola	PROCEDURE	matricula
prueba2	PROCEDURE	matricula
prueba3	PROCEDURE	matricula
prueba1	PROCEDURE	matricula
film_not_in_stock	PROCEDURE	sakila

Si queremos ver los procedimientos asociados a una base de datos concreta con la orden: SHOW PROCEDURE STATUS WHERE Db='nombre_db'.

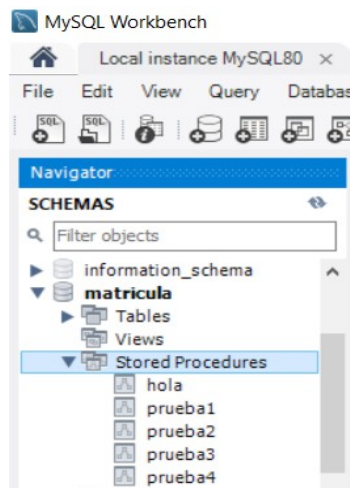
Ejemplo:

```
mysql>SHOW PROCEDURE STATUS WHERE Db='empleadodb';
```

```
mysql> show procedure status where db='empleadodb';
```

Db	Name	Type	Definer	Modified
empleadodb	calificacion	PROCEDURE	root@localhost	2020-08-26 10:00:00
empleadodb	dia	PROCEDURE	root@localhost	2020-08-26 10:00:00
empleadodb	dpto	PROCEDURE	root@localhost	2020-08-26 10:00:00
empleadodb	dpto_loop	PROCEDURE	root@localhost	2020-08-26 10:00:00
empleadodb	empleados	PROCEDURE	root@localhost	2020-08-26 10:00:00
empleadodb	esprimo	PROCEDURE	root@localhost	2020-08-26 10:00:00
empleadodb	geometria	PROCEDURE	root@localhost	2020-08-26 10:00:00
empleadodb	holamundo	PROCEDURE	root@localhost	2020-08-26 10:00:00

En el caso de workbench:



Para ejecutar un procedimiento se ha de realizar una llamada. Esto se hace mediante la orden: CALL.

Ejemplo: vamos a ejecutar un procedimiento llamado “calificacion” y realiza el cambio de una nota a su nombre de calificación: MD, Ins, Suf, B, Not o SB.

```
mysql> use empleadodb;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> call calificacion(8,@cal);
Query OK, 0 rows affected (0.00 sec)

mysql> select @cal;
+-----+
| @cal |
+-----+
| Not  |
+-----+
1 row in set (0.00 sec)
```

Es necesario poner en uso la base de datos a la que está asociada el procedimiento y llamarlo, en este caso el procedimiento requiere dos parámetros:

- Un valor numérico a traducir.
- Una variable donde dejará la calificación.

Para ver el código del procedimiento:

```
mysql> SHOW CREATE PROCEDURE holamundo;
```

Se puede asociar un comentario a un procedimiento. El comando sería: ALTER PROCEDURE.

```
mysql> ALTER PROCEDURE holamundo COMMENT 'Primer procedimiento';
```

Para borrar un procedimiento: DROP.

```
mysql> DROP PROCEDURE IF EXISTS holamundo;
```

Para modificar el contenido de un procedimiento, la rutina que describe, hay que eliminarlo y volver a crearlo. Es por ello que mantenerlo en un fichero independiente es buena opción. Se modifica el fichero y se vuelve a cargar (source).

Variables.

Variables locales.

Dentro de un procedimiento se pueden definir variables que solo se puedan ver dentro del

procedimiento.

Sintaxis:

```
DECLARE nombre_vble1 [,nombre_vble2 ...] tipo [DEFAULT valor];
```

Las variables locales se han de declarar antes del comienzo de las instrucciones (justo detrás de BEGIN). Se pueden declarar a la vez más de una variable del mismo tipo separadas por comas y se deben inicializar a un valor por defecto. Si no se inicializan quedan definidas a NULL.

Puede haber tantas líneas DECLARE como haga falta.

Los tipos válidos son todos los admitidos en los atributos al crear una tabla.

Para asignar un valor a una variable se utiliza comando: SET.

```
SET vble = expresión;
```

En una misma instrucción SET se pueden asignar valores a más de una variable:

```
SET vble1 = expresion1, vble2 = expresion2, ...;
```

En el procedimiento holamundo2, se utiliza una variable local.

Variables globales.

Las variables globales son las que definimos anteriormente como de usuario. Sus características son:

- El nombre empieza por @.
- No se definen y el tipo de dato que contiene es el del último que se le haya asignado.
- Su ámbito es la sesión del usuario.

Son accesibles dentro de los procedimientos y los valores que adquieran serán los valores que tendrán fuera del procedimiento.

En el procedimiento geometria(n), se utiliza una variable global.

Dentro de un procedimiento también se pueden hacer llamadas a otros procedimientos.

```
DELIMITER $$
DROP PROCEDURE IF EXISTS variables4 $$
CREATE PROCEDURE variables4 ()
BEGIN
  SET @v1='Raquel';
  CALL variables5(); ←
  SELECT @v1;
END $$
DELIMITER ;
```

```
DELIMITER $$
DROP PROCEDURE IF EXISTS variables5 $$
CREATE PROCEDURE variables5 ()
BEGIN
  SET @v1= CONCAT(@v1, ' y Mario Carrera');
END $$
DELIMITER ;
```

```
mysql> USE TEST
Database changed
mysql> CALL variables4(); ←
+-----+
| @v1 |
+-----+
| Raquel y Mario Carrera |
+-----+
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.02 sec)
```


Parámetros.

Los parámetros son variables que se envían a los programas y estos las reciben. En la definición del procedimiento, después del nombre se pueden especificar los parámetros que se le van a pasar entre paréntesis. Por cada parámetro se ha de especificar: nombre, tipo de parámetro y tipo de dato que contendrá.

```
CREATE PROCEDURE nombre_proc ([IN|OUT|INOUT] nombre_parametro tipo_dato ...)
```

Tipos de parámetros:

- IN. Opción por defecto. El procedimiento trabaja con una copia del parámetro que recibe. Por tanto, cuando termina y devuelve el control al punto donde se le llamó los parámetros que se le pasaron siguen manteniendo el valor que tenían antes de la llamada al procedimiento.
- OUT. En este caso los cambios que se hagan dentro del procedimiento quedan reflejados en el punto en el que se llamó. Pero, hasta que no se le asigne un valor dentro del procedimiento su valor será NULL.
- INOUT. En este caso el parámetro que se le pasa mantiene el valor hasta que se le asigne uno nuevo y cuando termine el procedimiento se mantendrá el último valor asignado en el punto donde se llamó el procedimiento.

Ejemplo de variable IN:

```
delimiter $$
drop procedure if exists prueba1$$
create procedure prueba1(in entrada int)
begin
    select entrada as "El valor de entrada es:";
    set entrada=entrada*2;
    select entrada as "Ahora es el doble";
end$$
delimiter ;
```

Al ejecutar el script con una variable global tenemos, observar que se mantiene el valor de la variable “@vble” al salir del procedimiento:

```
mysql> set @vble=12;
Query OK, 0 rows affected (0.00 sec)

mysql> call prueba1(@vble);
+-----+
| El valor de entrada es: |
+-----+
|           12 |
+-----+
1 row in set (0.00 sec)

+-----+
| Ahora es el doble |
+-----+
|           24 |
+-----+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)

mysql> select @vble;
+-----+
| @vble |
+-----+
|     12 |
+-----+
1 row in set (0.00 sec)
```

Ejemplo de variable OUT:

```
delimiter $$
drop procedure if exists prueba2$$
create procedure prueba2(out param1 int)
begin
    select param1 as "Valor al entrar en el procedimiento:";
    set param1=26;
    select param1 as "Valor una vez asignado 26:";
end$$
delimiter ;
```

Al ejecutar el script con una variable global tenemos, observar que se pierde el valor de la variable “@vble” nada mas entrar en el procedimiento pasando a ser NULL hasta que se le asigne un nuevo valor, que se mantendrá al salir del procedimiento:

```
mysql> set @vble=3;
Query OK, 0 rows affected (0.00 sec)

mysql> call prueba2(@vble);
+-----+
| Valor al entrar en el procedimiento: |
+-----+
|                                     NULL |
+-----+
1 row in set (0.00 sec)

+-----+
| Valor una vez asignado 26: |
+-----+
|                26 |
+-----+
1 row in set (0.02 sec)

Query OK, 0 rows affected (0.03 sec)

mysql> select @vble;
+-----+
| @vble |
+-----+
|    26 |
+-----+
1 row in set (0.00 sec)
```

Ejemplo de parámetro. INOUT:

```
delimiter $$
drop procedure if exists prueba3$$
create procedure prueba3(inout param1 int)
begin
    select param1 as "Valor al entrar en el procedimiento:";
    set param1=param1*2;
    select param1 as "Valor una vez doblado:";
end$$
delimiter ;
```

En esta ocasión al ejecutar el script con una variable global tenemos, se mantiene el valor al entrar en el procedimiento y los cambios que se le hagan al parámetro permanecen en la variable con que se llamó al procedimiento:

```
mysql> set @vble=9;
Query OK, 0 rows affected (0.00 sec)

mysql> call prueba3(@vble);
+-----+
| Valor al entrar en el procedimiento: |
+-----+
|                                     9 |
+-----+
1 row in set (0.02 sec)

+-----+
| Valor una vez doblado: |
+-----+
|                   18 |
+-----+
1 row in set (0.03 sec)

Query OK, 0 rows affected (0.03 sec)

mysql> select @vble;
+-----+
| @vble |
+-----+
|    18 |
+-----+
1 row in set (0.00 sec)
```

Instrucciones condicionales.

IF-THEN-ELSE.

Se analiza una condición y según el valor de verdad ejecutará un conjunto de instrucciones u otro.

Sintaxis:

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

search_condition es la condición a analizar. Si el resultado es Verdadero/True/1 ejecutará todas las instrucciones entre “THEN” y “ELSE”, en caso contrario ejecutará las ordenes entre “ELSE” y “END IF”.

En el siguiente ejemplo dada una nota, que se pasará a través del parámetro de entrada (IN), comprueba si es mayor o igual a 5, en ese caso asigna la cadena “Aprobado.” a “nota”, si no le asigna la cadena “Suspenso.”. Una vez hecho esto, muestra el valor de la variable local “nota”.

```

delimiter $$
drop procedure if exists prueba4$$
create procedure prueba4(in param1 int)
begin
    declare nota varchar(10) default 'Indefinido';
    select param1 as "Valor al entrar en el procedimiento:", nota as Nota;
    if param1 >= 5 then
        set nota = 'Aprobado.';
    else
        set nota = 'Suspenso.';
    end if;
    select nota as Nota;
end$$
delimiter ;

```

Como parámetros podemos pasar una constante o una variable global.

<pre> mysql> call prueba4(4); +-----+-----+ Valor al entrar en el procedimiento: Nota +-----+-----+ 4 Indefinido +-----+-----+ 1 row in set (0.00 sec) +-----+ Nota +-----+ Suspenso. +-----+ 1 row in set (0.01 sec) Query OK, 0 rows affected (0.01 sec) mysql> call prueba4(4); </pre>	<pre> mysql> set @vble=6; Query OK, 0 rows affected (0.00 sec) mysql> call prueba4(@vble); +-----+-----+ Valor al entrar en el procedimiento: Nota +-----+-----+ 6 Indefinido +-----+-----+ 1 row in set (0.00 sec) +-----+ Nota +-----+ Aprobado. +-----+ 1 row in set (0.01 sec) </pre>
--	---

Se pueden anidar de la forma:

```

IF condicion1 THEN
    acciones1;
ELSEIF condicion2 THEN
    acciones2;
ELSEIF condicion3 THEN
    acciones3;
ELSE
    acciones4
END IF

```

Se pueden poner tantos bloques ELSEIF como sean necesarios.

CASE.

Similar a la sentencia condicional anterior IF-THEN-ELSE. Todo lo que se puede expresar con CASE se puede expresar con IF-THEN-ELSE. CASE ofrece un código más legible cuando hay un número elevado de comparaciones a realizar.

Podemos utilizar dos sintaxis:

```

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

```

```

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

```

La diferencia entre una y otra es si se evalúa una expresión y según el resultado se ejecutan unas listas de instrucciones u otras; o una condición para cada lista de instrucciones.

Instrucciones repetitivas o bucles.

Las instrucciones repetitivas tienen en común repetir la ejecución de un conjunto de instrucciones denominadas cuerpo del bucle hasta que se cumpla una condición. Según cuando se realice la condición tendremos distintas estructuras:

- Loop: Entre las instrucciones del cuerpo del bucle.
- While: Al inicio del cuerpo del bucle.
- Repeat: Al final del cuerpo del bucle.

Es muy importante de comprobar que en todas las posibles entradas al bucle se cumplirá la condición de salida del mismo. De no ser así, entraríamos en un bucle sin fin y el sistema se bloquearía.

LOOP.

Todas las instrucciones comprendidas entre las palabras reservadas “LOOP” y “END LOOP” forman el cuerpo del bucle, se ejecutan un número de veces hasta que la ejecución del bucle se encuentra con la instrucción “LEAVE label”.

Sintaxis:

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

Ejemplo:

<pre>DELIMITER \$\$ DROP PROCEDURE IF EXISTS prueba7\$\$ CREATE PROCEDURE prueba7(limite INT) BEGIN DECLARE contador INT; SET contador=0; etiquetal: LOOP SET contador=contador+1; IF contador <= limite THEN SELECT contador; ELSE LEAVE etiquetal; END IF; END LOOP etiquetal; END\$\$ DELIMITER ;</pre>	<pre>DELIMITER \$\$ DROP PROCEDURE IF EXISTS prueba7\$\$ CREATE PROCEDURE prueba7(limite INT) BEGIN DECLARE contador INT; SET contador=0; etiquetal: LOOP SET contador=contador+1; IF contador <= limite THEN SELECT contador; ITERATE etiquetal; END IF; LEAVE etiquetal; END LOOP etiquetal; END\$\$ DELIMITER ;</pre>
---	---

LEAVE: Cuando llega a ejecutarse sale del bucle y se ejecutará la primera sentencia detrás de “END LOOP”.

ITERATE: Cuando se ejecute vuelve al principio del bucle, saltándose el resto de instrucciones del cuerpo del bucle. Es recomendable evitar esta instrucción, pues no sigue la metodología de programación estructurada.

REPEAT.

El cuerpo del bucle se ejecutará hasta que sea cierta la condición de final que se sitúa al final del cuerpo del bucle.

Sintaxis:

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

Dentro de la lista de instrucciones 'statement_list' se ha de modificar la/s variable/s que intervengan en la condición detrás de 'UNTIL' para evitar un bucle sin fin.

Ejemplo: el mismo ejemplo visto con LOOP, pero con REPEAT:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS prueba8$$
CREATE PROCEDURE prueba8(limite INT)
BEGIN
    DECLARE contador INT;
    SET contador=0;
    etiqueta1: REPEAT
        SET contador=contador+1;
        SELECT contador;
    UNTIL contador >= limite
    END REPEAT etiqueta1;
END$$
DELIMITER ;
```

WHILE.

En este caso la condición se evalúa antes de entrar en el bucle y se vuelve a comprobar antes de cada iteración. Al igual que en las anteriores en el cuerpo del bucle se ha de modificar la/s variables/s que intervienen en la condición de entrada al WHILE, de otra forma: bucle sin fin.

Sintaxis:

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

El mismo ejemplo anterior resuelto con WHILE:

```

DELIMITER $$
DROP PROCEDURE IF EXISTS prueba9$$
CREATE PROCEDURE prueba9(limite INT)
BEGIN
    DECLARE contador INT;
    SET contador=0;
    etiqueta1: WHILE contador<limite DO
        SET contador=contador+1;
        SELECT contador;
    END WHILE etiqueta1;
END$$
DELIMITER ;

```

Bloques de instrucciones.

Los procedimientos vistos hasta ahora solo contienen un bloque de instrucciones. Esto es todo lo comprendido entre BEGIN y END.

Se pueden poner más de un bloque de instrucciones dentro del mismo procedimiento, esto permite agrupar de forma lógica las instrucciones que forman el procedimiento para realizar una determinada función. Los bloques, al igual que las estructuras anteriores, pueden estar anidados. En todo anidamiento el bloque interno ha de terminar antes que el bloque externo

Respecto a las variables utilizadas, solo serán visibles dentro del bloque en el que se han declarado. Por tanto, son visibles en procedimientos internos al que han sido declaradas. En el caso que la variable externa e interna tuviesen el mismo nombre, dentro del bloque interno se referencia a la interna.

Dentro de un bloque podemos encontrar:

- Declaración de variables.
- Declaración de cursores
- Declaración de manejadores de error.
- Código de procedimiento.

Para evitar errores se ha de seguir ese orden.

Los bloques se pueden etiquetar, esto permite que sea más legible el código. La etiqueta se ha de poner antes de BEGIN y detrás de END.

Al usar las etiquetas se facilita el uso del comando LEAVE, este permite abandonar un bloque antes de llegar al final del mismo. Si se especifica la etiqueta se sale del bloque de esa etiqueta (independientemente sea interno o externo) y se ejecutará la primera instrucción detrás del END del bloque que ha abandonado. Si fuese el último saldría del procedimiento.

Estructura:

```

etiqueta1: BEGIN
    Declaración de variables.
    Declaración de cursores.
    Declaración de manejadores de error.
    Código del bloque.
END etiqueta1;

```

Ejemplos:

```

1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS bloque3 $$
3 CREATE PROCEDURE bloque3 ()
4 BEGIN
5     DECLARE v INT DEFAULT 500;
6     BEGIN
7         DECLARE v INT;
8         SET v = 200;
9         SELECT v; -- 200
10    END;
11    SELECT v; -- 500
12 END $$
13 DELIMITER ;

```

```

1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS bloque4 $$
3 CREATE PROCEDURE bloque4 ()
4 bloque_externo: BEGIN
5     DECLARE v INT DEFAULT 100;
6     bloque_interno: BEGIN
7         IF v < 500 THEN
8             LEAVE bloque_interno;
9         END IF;
10        SELECT 'No llega a ejecutarse esta instrucción';
11    END bloque_interno;
12    SELECT 'Fin del bloque externo';
13 END bloque_externo $$
14 DELIMITER ;

```

Funciones.

Las funciones a diferencia de los procedimientos devuelven un valor, esto se hace con la orden RETURN. Los parámetros de entrada OUT y INOUT no tienen sentido, pues la comunicación se hace a través del valor devuelto con RETURN. Lo expuesto para los procedimientos es aplicable a las funciones.

Las funciones pueden utilizarse dentro de otras instrucciones y funciones MySQL, el resultado que devuelven será con el que trabajen dichas instrucciones o funciones.

Consideraciones de las funciones:

La cláusula RETURNS es obligatoria y define el tipo de dato que devuelve la función.

No especifica el tipo de parámetro (IN, OUT, INOUT). Siempre se consideran IN.

Dentro del cuerpo de la función ha de haber al menos una instrucción RETURN que devuelve el valor indicado al punto donde se llamó y se da por finalizada la función. Puede haber más de un RETURN. Si se llega al final de la función y no hay RETURN provocaría un error.

Para ver las funciones que existen en una determinada base de datos:

```
mysql>SHOW FUNCTION STATUS WHERE Db='nombre_base_datos';
```

Para ver todas las funciones de todas las bases de datos:

```
mysql>SHOW FUNCTION STATUS;
```

Al crear una función, si tenemos activado el registro binario de log, debemos indicar que dicha función es **determinista**. Una función se denomina determinista si siempre va a devolver el mismo resultado al aplicar la misma entrada. Si no fuera así y realizamos un procedimiento de restauración de la base de datos empleando el fichero de log binario, podría darse el caso de que los datos 'recuperados' no fueran los mismos que antes de la recuperación.

Para ello, en la definición de la función se ha de especificar: DETERMINISTIC.

```
CREATE FUNCTION nombre_funcion() RETURNS tipo_dato DETERMINISTIC
```

Ejemplo de función que recibe como parámetro una cantidad de pesetas devuelve los euros a los que equivale.

Sin tener activado el registro bin:	Estando activado el registro bin:
<pre>DELIMITER \$\$ DROP FUNCTION IF EXISTS euros \$\$ CREATE FUNCTION euros (ptas DECIMAL(12,2)) RETURNS DECIMAL(10,3) BEGIN DECLARE n_euros DECIMAL(10,3); SET n_euros=ptas/166.386; RETURN (n_euros); END \$\$</pre>	<pre>DELIMITER \$\$ DROP FUNCTION IF EXISTS euros \$\$ CREATE FUNCTION euros (ptas DECIMAL(12,2)) RETURNS DECIMAL(10,3) DETERMINISTIC BEGIN DECLARE n_euros DECIMAL(10,3); SET n_euros=ptas/166.386; RETURN (n_euros); END \$\$</pre>

Cursores.

Es el instrumento que se utiliza cuando una sentencia SELECT de MySQL devuelve más de una fila y queremos almacenarlo en una o un conjunto de variables, según el número de columnas que devuelva.

El cursor es una zona de memoria que contendrá el conjunto de filas resultantes de la sentencia SELECT con la ventaja de poder recorrer cada fila una a una para visualizarlas y/o manipularlas. Para ello se ha de declarar el cursor, abrirlo, movernos por las distintas filas y cerrar el cursor.

Sintaxis para declarar el cursor:

```
DECLARE nombre_cursos CURSOR FOR sentencia_select;
```

Los cursores han de declararse en los procedimientos o funciones después de las variables, de no hacerlo así produciría una situación de error.

Para abrir el cursor declarado:

```
OPEN nombre_cursor;
```

Para pasar al siguiente registro del cursor:

```
FETCH nombre_cursor INTO vble1, vble2, ...;
```

Si se llega a la última fila o registro del cursor y se vuelve a ejecutar FETCH se provoca un error.

Para cerrar el cursor:

```
CLOSE nombre_cursor;
```

Cada vez que se ejecuta FETCH se pasa a la fila siguiente del cursor. Cuando se llega a la última fila y se intenta pasar a la siguiente nos devuelve un mensaje de error:

```
("ERROR 1329 (02000): No data - zero rows fetched, selected, or processed")
```

Para tratar el error y evitar ese mensaje con su correspondiente pérdida de control del sistema definimos un manejador de error. En esta situación el manejador asignará un valor a una variable que controle el final del proceso:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET vble=valor_de_fin;
```

Ejemplo: De las tablas empleado y departamento vamos a crear una nueva tabla que tendrá tres columnas:

- Código identificador de empleado. Número entero ascendente.
- Nombre del empleado. Con el formato: "apellido1 apellido2, nombre".

- Nombre del departamento al que pertenece.

El procedimiento creará la tabla y registro a registro de los empleados irá insertando su nombre y el del departamento en la nueva tabla.

```
DELIMITER $$
DROP PROCEDURE IF EXISTS empl_dpto $$
CREATE PROCEDURE empl_dpto()
BEGIN
    DECLARE n_empleado, n_dpto VARCHAR(50);
    DECLARE nombree,apellido1,apellido2,nombred VARCHAR(50);
    DECLARE contador INT DEFAULT 1;
    DECLARE fin_cursor VARCHAR(2) DEFAULT 'no';
    DECLARE cursor_ed CURSOR FOR
        SELECT e.nombre, e.apellido1, e.apellido2, d.nombre
        FROM empleado e JOIN departamento d
        ON e.codigo_departamento=d.codigo;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin_cursor='si';

    DROP TABLE IF EXISTS EMP_DPTO;
    CREATE TABLE IF NOT EXISTS EMP_DPTO
    (
        codigo int not null primary key,
        empleado varchar(50) not null,
        departamento varchar(50) not null
    );

    OPEN cursor_ed;
    WHILE fin_cursor='no' DO
        FETCH cursor_ed INTO nombree, apellido1, apellido2, nombred;
        SET n_empleado=CONCAT_WS(' ', CONCAT_WS(' ', apellido1, apellido2),
                                nombree);
        SET n_dpto=nombred;
        INSERT INTO EMP_DPTO VALUES (contador,n_empleado,n_dpto);
        SET contador=contador+1;
    END WHILE;
    CLOSE cursor_ed;
    SELECT 'Tabla Empleados - Departamentos creada.' as 'Mensaje' ;
END $$
DELIMITER ;
```

Gestión de errores.

Si una sentencia falla dentro de un programa almacenado provoca un error que interrumpe la ejecución del programa en ese punto y finaliza su ejecución, salvo que se gestione adecuadamente. En caso de error, no gestionado, el control pasa al punto donde se llamo al programa. Si el error ocurre dentro de un bloque de instrucciones sale del bloque (al punto justo detrás de su END).

Para no perder el control de la ejecución del sistema se definen los **controladores de error o handler**.

Un handler es un bloque de instrucciones MySQL que se ejecutan cuando se verifica una condición de excepción o error generada por el servidor.

Sintaxis:

```
DECLARE {CONTINUE|EXIT|UNDO} HANDLER FOR {SQLSTATE codigo_sqlstate|
Mysql codigo_error|nombre_condicion}
Instrucciones_del_manejador
```

La declaración de los manejadores de error se ha de realizar detrás de la declaración de variables y cursors.

Tipos de manejadores:

- CONTINUE: Se mantiene el control de la ejecución del programa.
- EXIT: Se pone fin al bloque de instrucciones o programa donde ocurre el error. El control vuelve al punto donde se llamó o al bloque externo si ocurre en un bloque de instrucciones interno.
- UNDO: No implementado en MySQL.

La condición del error se puede expresar de tres formas distintas:

- Código estándar de ANSI SQLSTATE.
- Código de error de MySQL.
- Una expresión.

En caso de producirse el error, que se está gestionando, se ejecutará el conjunto de ordenes indicado.

Ejemplo: al dar de alta un registro cuya clave primaria ya existe se genera un error MySQL número 1062 “entrada duplicada”. Podemos crear un procedimiento en el que se controle ese error:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS prueba $$
CREATE PROCEDURE prueba (codigo INT, nombre VARCHAR(30))
BEGIN
    DECLARE CONTINUE HANDLER FOR 1062
        SELECT CONCAT('Nº Alumno: ',codigo,' ya existe.')
            AS 'Aviso de error.';
    INSERT INTO tabla_alumno VALUES (codigo,nombre);
    SELECT 'Aún en el procedimiento, a pesar del error';
END $$
DELIMITER ;
```

Si llamamos a este procedimiento con un alumno que ya existe saltará el error y el mensaje programado se mostrará, de no ser así se dará de alta.

```
mysql>CALL(4,'José Luis');
```

Triggers.

Un trigger es como un procedimiento que se ejecuta automáticamente cuando sobre una tabla se realiza una operación que modifica algún dato: DELETE, INSERT o UPDATE.

El trigger ha de estar asociado a una tabla y a una operación en concreto. No tiene por qué realizar alguna operación sobre la tabla, sino que se activa cuando se realiza algún: DELETE, INSERT o UPDATE. Por cada fila modificada se ejecuta una vez el trigger y puede ejecutarse antes o después de la orden que modifica algún dato.

Usos que puede tener un trigger:

- Monitorizar operaciones sobre una tabla. Se puede guardar información de quien y cuando modificó una tabla.
- Verificación de datos. Antes de modificar alguna información el trigger puede comprobar que los datos son correctos.
- Mecanismo para implementar columnas calculadas. Ejemplo: Supongamos la gestión de un almacén. Si queremos mantener un stock óptimo de artículos, cada vez que vendamos una unidad de un artículo que esté por debajo de ese óptimo se puede incrementar en una tabla de pedidos a proveedores, de forma que cuando venga el próximo pedido se volverán a esos

niveles óptimos.

Los trigger no se van a ejecutar en caso de operaciones en cascada debido a las restricciones por integridad referencial.

Cuando un trigger se ejecuta **antes** (BEFORE) de la instrucción que lo activa suele ser para comprobar que los datos son correctos. En caso de que se active **después** (AFTER) de la instrucción se suele utilizar para columnas calculadas, realizar o registrar operaciones una vez modificada la tabla.

Crear Trigger.

Es necesario tener permiso TRIGGER sobre la tabla donde se va a crear.

Para crear un TRIGGER es necesario definir:

- Sobre que tabla se va a aplicar.
- Qué operación lo va a activar: INSERT, DELETE o UPDATE.
- Se ejecutará antes o después de la acción que lo activa.

El trigger se ejecutará por cada una de las filas que se vean afectadas con las ordenes: INSERT, DELETE o UPDATE.

Dentro del trigger se puede acceder a los valores de cada fila de la tabla antes y después de la ejecución de la orden que lo dispara. Es decir, si se realiza una operación UPDATE se puede acceder tanto a los valores de la tabla antes del UPDATE, con el alias OLD; como a los valores de la tabla después de ejecutar UPDATE, con el alias NEW.

Ejemplo: Supongamos que disponemos de un trigger que se activa al modificar la tabla cliente. Si queremos modificar el nombre de un cliente: 'Mariano' por 'José Pedro' que está en la clave primaria: 7. La orden a ejecutar sería:

```
mysql> UPDATE cliente SET nombre='José Pedro' WHERE id=7;
```

Esta instrucción activará el TRIGGER y dentro del cuerpo de instrucciones podemos disponer de:

- OLD.nombre, que tendría el valor 'Mariano'.
- NEW.nombre que tendría el valor 'José Pedro'

Según las instrucciones a ejecutar tendremos:

- UPDATE:
 - NEW: valores nuevos.
 - OLD: valores anteriores.
- INSERT:
 - NEW: valores nuevos.
 - OLD: no existen.
- DELETE:
 - NEW: no existen.
 - OLD: valores anteriores.

Sintaxis de creación de trigger, MySQL 8.0:

```

CREATE
    [DEFINER = user]
    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

```

Se ha de utilizar la orden DELIMITER, igual que con los procedimientos y funciones.

La orden LOAD DATA también provoca la ejecución del trigger ya que realiza operaciones de INSERT sobre las tablas.

Es posible disponer de más de un TRIGGER sobre la misma tabla y la misma operación. En ese caso el orden de ejecución será el orden de creación, salvo que se haya alterado con las cláusulas: FOLLOWS o PRECEDES.

Ejemplo de trigger que se active al añadir un registro en la tabla clientes. El objetivo es recoger la información de la modificación, en este caso de un alta, para posteriores comprobaciones. Se almacenara en una tabla: cliente_ctrol los datos del usuario que realiza el alta, la fecha y hora y los datos del cliente que se da de alta. Se puede hacer con el siguiente código:

```

DELIMITER $$
CREATE TRIGGER alta_cliente AFTER INSERT ON cliente FOR EACH ROW
BEGIN
    INSERT INTO cliente_ctrol (usuario, fecha, hora, accion, registro,
                             nombre, apellido1, apellido2, ciudad, categoria)
    VALUES (user(), date(now()), time(now()), 'Alta', new.id,
            new.nombre, new.apellido1, new.apellido2, new.ciudad,
            new.categoria);
END $$
DELIMITER ;

```

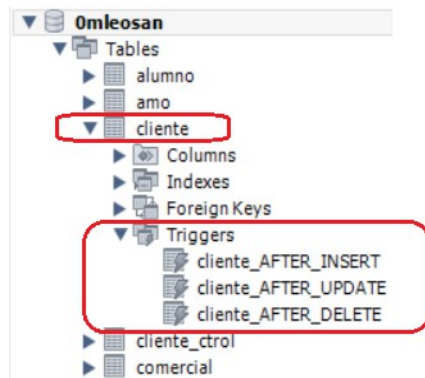
La cláusula DEFINER recoge al usuario que crea el TRIGGER. Dicho usuario ha de tener permiso otorgado con GRANT sobre la tabla para crear, borrar, mostrar y ejecutar un trigger. Cuando otro usuario realiza la operación MySQL sobre la tabla a la que está asociado el trigger, el usuario que creó el trigger ha de conservar los permisos antes mencionados para que se pueda ejecutar.

Como dentro del trigger se puede acceder a los valores anteriores y posteriores, entoces:

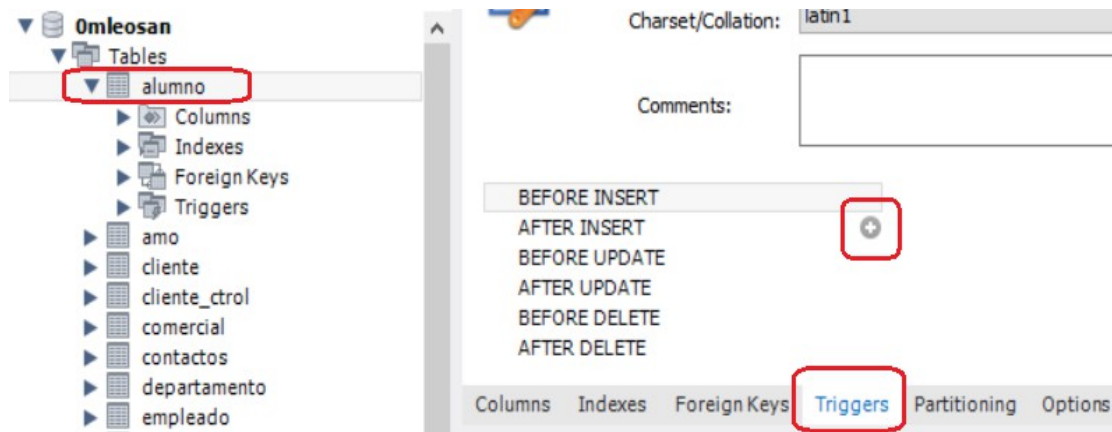
- Para poder modificar los valores nuevos (SET NEW.col=valor) el usuario que creo el trigger ha de tener permisos de UPDATE sobre la tabla.
- Para poder consultar un valor anterior o el nuevo (OLD.col o NEW.col) el usuario que creo el trigger ha de tener permisos de SELECT sobre la tabla.

Con workbench.

Los TRIGGER van asociados a cada tabla. La siguiente imagen muestra los triggers asociados a la tabla 'cliente':



Para crear un trigger se ha de ir a la estructura de la tabla: 'ALTER TABLE' y elegir la pestaña 'trigger'.



En la imagen se ha seleccionado la tabla 'alumno' para añadir un TRIGGER, después de insertar un registro.

Para ello, se accede a modificar su estructura (ALTER TABLE) y en la parte inferior derecha se ha hecho clic sobre 'trigger'. Se muestran las tres instrucciones que activan los triggers, cada una de ellas antes o después de ejecutarse. Según elijamos al pasar el ratón aparece el símbolo '+' que nos permitirá añadir un trigger en la acción que nos interese. En este caso: AFTER INSERT.

Visualizar triggers.

Con el comando:

```
mysql>SHOW TRIGGERS;
```

Se muestran los triggers asociados a la base de datos por defecto en uso. Para especificar una en particular:

```
mysql>SHOW TRIGGERS FROM nombre_BD;
```

Se pueden especificar las cláusulas LIKE y WHERE para especificar con más detalle los triggers asociados a una tabla en particular o que cumplan alguna condición concreta.

```
mysql>SHOW TRIGGERS WHERE Definer LIKE 'root%';
```

Modificación de triggers.

No se pueden modificar. Habría que eliminarlo y volverlo a crear.

Borrar triggers.

```
mysql>DROP TRIGGERS IF EXISTS nombre_trigger;
```

Consideraciones.

Algunas funciones que pueden ser útiles en un trigger:

USER(). Para obtener el usuario actual. El que ha ejecutado la orden que ha activado el trigger.

NOW(). Devuelve la fecha y la hora actual.

DATE(NOW()). Devuelve sólo la fecha actual.

TIME(NOW()). Devuelve sólo la hora actual.

SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Mensaje de aviso a mostrar'. Cancela la operación que activó el trigger y muestra ese mensaje. El valor 45000 identifica una excepción definida por el usuario.

DATE_FORMAT(fech, form) Devolverá la fecha dada “fech” con un formato específico “form”.

Los formatos disponibles son:

“%W”	Nombre del día.
“%D”	Cardinal del día.
“%d”	Día con dos dígitos.
“%e”	Día con uno o dos dígitos, según necesite.
“%Y”	Año con cuatro dígitos.
“%y”	Año con 2 dígitos.
“%M”	Nombre de mes completo.
“%m”	Número del mes con dos dígitos.
“%b”	Nombre del mes, pero solo las tres primeras letras.

Estos formatos se pueden incluir más de uno de ellos en la misma orden incluso usar otros caracteres. Ejemplo: supongamos que queremos mostrar la fecha de hoy con el formato: **lunes, 31 de enero de 2021**

```
mysql> select date_format(now(), "***%W, %d de %M de %Y***");
+-----+
| date_format(now(), "***%W, %d de %M de %Y***") |
+-----+
| **Monday, 31 de January de 2022**              |
+-----+
1 row in set (0.01 sec)
```

Observa que los nombre de día y mes aparecen en inglés. Nomenclatura por defecto.

Por defecto los nombre de días y meses aparecen en inglés, configuración por defecto. Esto afecta a los resultados de la función: DATE_FORMAT. Esta configuración se gestiona desde la variable global: lc_time_names, cuyo valor por defecto es: “en_US”. Para que aparezcan los nombres en castellano habría que cambiarla a: “es_ES”.

>SELECT @@lc_time_names; # para consultar el valor actual.

>SET lc_time_names = “es_ES”; # para asignarla un nuevo valor.

```

mysql> select date_format(now(), "**%W, %d de %M de %Y**");
+-----+
| date_format(now(), "**%W, %d de %M de %Y**") |
+-----+
| **Monday, 31 de January de 2022**          |
+-----+
1 row in set (0.00 sec)

mysql> select @@lc_time_names;
+-----+
| @@lc_time_names |
+-----+
| en_US           |
+-----+
1 row in set (0.00 sec)

mysql> set lc_time_names = "es_ES";
Query OK, 0 rows affected (0.00 sec)

mysql> select @@lc_time_names;
+-----+
| @@lc_time_names |
+-----+
| es_ES           |
+-----+
1 row in set (0.00 sec)

mysql> select date_format(now(), "**%W, %d de %M de %Y**");
+-----+
| date_format(now(), "**%W, %d de %M de %Y**") |
+-----+
| **lunes, 31 de enero de 2022**              |
+-----+
1 row in set (0.00 sec)

```

También puede ser necesario modificar las variables globales que gestionan la zona horaria, para ajustar los resultados que ofrecen las funciones relacionadas con la hora a los de nuestra zona horaria. Nuestra zona horaria es: “Europe/Madrid” y las variable que lo gestiona: time_zone.

```

mysql> SELECT @@GLOBAL.time_zone, @@SESSION.time_zone;
mysql> SET GLOBAL time_zone = “Europe/Madrid”;
mysql> SET time_zone = “Europe/Madrid”;

```

Por defecto tiene el valor “SYSTEM”, toma la zona horaria del sistema, por lo que no ha de haber problemas.

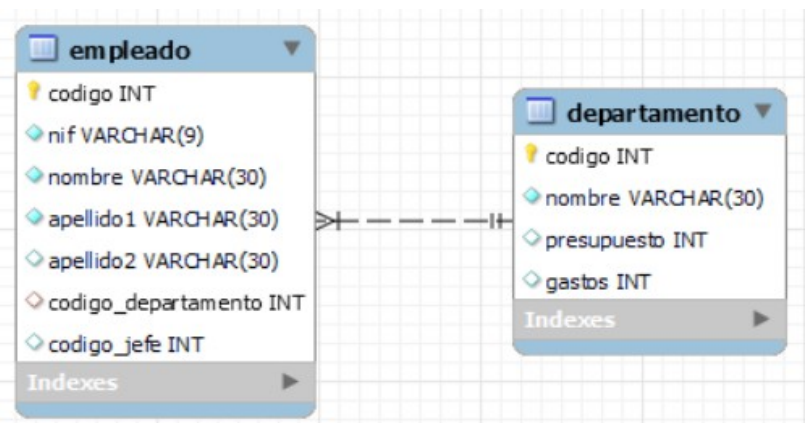
Ejercicios.

1. Realiza un procedimiento que muestre el mensaje “Hola Mundo”.
2. Realiza un procedimiento que almacene en una variable local el mensaje “Hola Mundo 2” y muestre el contenido de esa variable.
3. Geometría: geometria(IN lado FLOAT). Realiza un programa que recibe como parámetro un valor real (FLOAT) y calcula:
 - Área de un triángulo rectángulo isósceles en el que el valor dado representa el tamaño de los catetos.
 - Área de un cuadrado cuyo lado es el parámetro pasado.
 - Área de un círculo cuyo radio es el parámetro indicado.

Se debe modificar una variable global indicando que se han realizado los cálculos. A la vuelta de la llamada se verificará el valor de la misma.

4. Par-impar: parimpar(IN numero INTEGER). El procedimiento ha de mostrar un mensaje indicado si el número que se pasa como parámetro es par o es impar.
5. Calificaciones: calificacion(nota FLOAT, OUT cali VARCHAR(15)). El procedimiento ha de convertir un valor real entre 0 y 10 en una calificación según la siguiente tabla:

- Entre 0 y 3,5: MD.
 - Entre 3,5 y 4,5: Ins.
 - Entre 4,5 y 5,5: Suf.
 - Entre 5,5 y 6,5: B.
 - Entre 6,5 y 8,5: Not.
 - Entre 8,5 y 10: Sb.
 - Menor que 0 o mayor que 10: Calificación no válida.
6. Día de la semana: dia(num INTEGER, OUT dia VARCHAR(15)). Uso de CASE. Ha de devolver el nombre del día según el número que se pase como parámetro. Ej: para 3 devolver miércoles, para 4 jueves,..
 7. Nombre del mes: mes(fecha DATE, OUT mes VARCHAR(15)). Uso de CASE. Ha de devolver el nombre del mes según la fecha que se le pase. La fecha tendrá el formato: yyyy-mm-aa.
 8. Lista de números pares: listapar(tamano INTEGER). Bucle LOOP. Crea la tabla “pares” donde se almacenarán los primeros números pares, tantos como indique tamano. También se ha de acumular la suma de los mismos sobre una variable global: @suma.
 9. Calcula la suma de todos los números naturales hasta un valor dado. Ejemplo: para el 6 ha de calcular: $1 + 2 + 3 + 4 + 5 + 6 = 21$. Realiza el ejercicio con bucle REPEAT con un parámetro numérico de entrada. Comprueba que el valor de entrada esté entre 1 y 20.
 10. Comprobar si un número es primo o no: esprimo(numero INTEGER, OUT primo VARCHAR(1)). Bucle WHILE. Dado un número que recibe el procedimiento como parámetro se dejará en la variable de salida “primo” una “s” o una “n” según sea primo o no el número dado.
 11. Lista de números primos: listaprimo(tamano INTEGER). Bucle WHILE. Se ha de crear una tabla con los primeros números primos, tantos como indique tamano. Se ha de utilizar el procedimiento “esprimo()” del ejercicio anterior.
 12. Empleados de un departamento: empleados(departamento VARCHAR(25)). SELECT-JOIN. Dado un departamento se mostrarán los empleados del mismo.



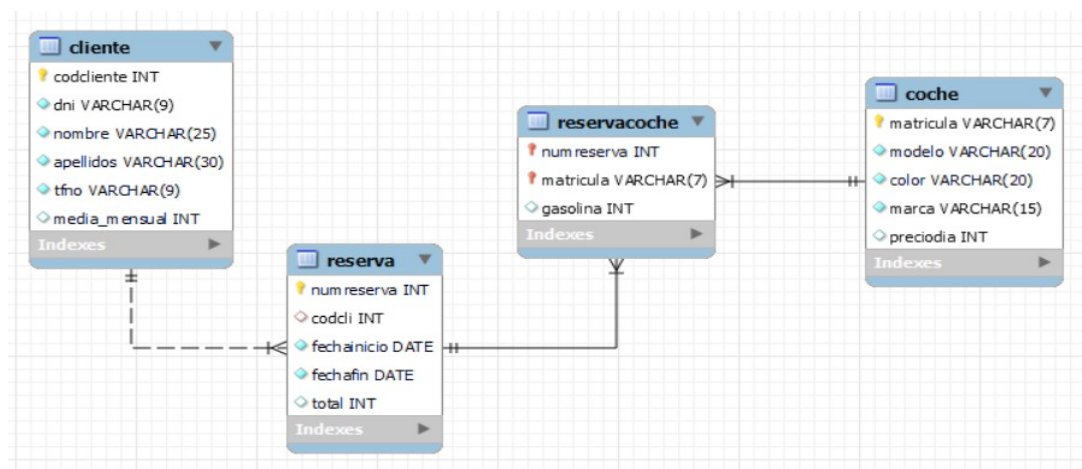
13. Información sobre un empleado en particular: infemp(codi INTEGER). A partir del código de un empleado se ha de mostrar:
 - Nombre del departamento al que pertenece.
 - Nombre del jefe del departamento (nombre y apellidos en una columna).
 - Listado de sus compañeros/as. En el listado se ha de evitar que aparezca el propio empleado.
14. Factura. Sobre la tabla concesionariodb. Se pide actualizar el valor del total de una reserva

dada. Para ello se ha de tener en cuenta los coches que se han alquilado en esa reserva, cada uno tiene su propio precio por día y los litros de gasolina con los que se entrega. El procedimiento tendrá como parámetros de entrada: N° de reserva (INT) y precio del litro de gasolina (FLOAT).

Además de actualizar: reserva.total el procedimiento ha de mostrar un detalle que contenga: N° Reserva, Matricula, marca, modelo, Fecha inicio y fin, Días alquilados, litros de gasolina, Precio de la gasolina y sub-total (por ese vehículo).

Ejemplo:

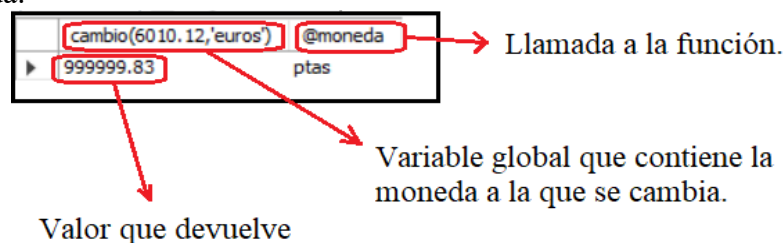
	Reserva	Matricula	marca	modelo	Fecha inicio	Fecha fin	Dias	Precio día	Litros gasolina	Precio gasolina	Sub-total
▶	4	2222ABB	Toyota	Corolla	2020-12-07	2020-12-18	11	30	30	1.81	384.29
	4	2233ABK	Opel	Corsa	2020-12-07	2020-12-18	11	20	30	1.81	274.29
	4	3333GTA	Citroen	Picasso	2020-12-07	2020-12-18	11	35	30	1.81	439.29



15. Función que devuelva el factorial de un número natural. El factorial de un número n se define como el producto de todos los números anteriores hasta 1: $n! = 1 * 2 * 3 * \dots * n$. Teniendo en cuenta que:

- El factorial de 0 es 1: $0! = 1$
- No están definidos los factoriales para los números negativos.

16. Realiza una función que realice el cambio entre euros y pesetas. Los parámetros de entrada serán: Cantidad de dinero y en qué moneda está esa cantidad. La función ha de devolver el cambio en la otra moneda. Puedes usar una variable global que devuelva en nombre de la otra moneda.



17. Modifica el procedimiento que calcula si un número es primo y conviértelo en una función que recibe un número Natural (0, 1, 2, 3, 4,...) y nos devuelve si es primo o no.

18. Del mismo modo modifica el procedimiento que muestra la lista de números primos de la siguiente forma:

- Utilice una función para saber si un número es primo o no.
- Se pasa como parámetros un número entre 3 y 6, que indica un número de cifras que ha

de tener el primer número primo a mostrar (100, 1000, 10000, 100000). Ej para el 3 el primer primo ha de ser a partir de 100.

- El procedimiento ha de calcular los 10 primeros primos que empiecen a partir del número de cifras indicado.

19. Realiza un procedimiento en el que se asigne un correo electrónico a cada uno de los empleados de nuestra empresa. Utiliza las tablas de la base de datos empleadodb: empleado y departamento. Se han de añadir dos columnas:

- correo: formado por: inicial de nombre + tres primeras letras del primer apellido + tres primeras letras del segundo apellido + @ + nombre departamento + .es
- contraseña: su nombre puesto al revés +tres últimos números del NIF. Ejemplo de Juan con NIF 23100233x sería: nauJ233.

Para poder volver a ejecutar has de tener en cuenta que se han creado las columnas: correo y contrasena.

Debes usar cursores.







Las funciones de carácter más apropiadas para utilizar son:

TRIM, CONCAT WS, LEFT, SUBSTR y REVERSE

20. Se necesita realizar un rastreo del mantenimiento de los clientes de la base de datos concesionariodb. Dicho rastreo consiste en dejar constancia en una nueva tabla cliente_ctrol de todas las modificaciones de datos sobre clientes: Altas, bajas y actualización. Recogiendo el usuario que ha realizado la actualización, fecha, hora y tipo de actualización así como los datos modificados. Debes realizar el trabajo definiendo los triggers que consideres oportunos.

- Has de disponer de más de un usuario con permisos para modificar los datos de los clientes (alta, bajas y actualización).
- Realiza los distintos cambios con cada uno de ellos. Estas pruebas han de quedar recogidas en: cliente_ctrol.
- Las descripciones de las tablas: cliente y cliente_ctrol:

Descripción tabla 'cliente'

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G
 id	INT(10)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
 nombre	VARCHAR(100)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 apellido1	VARCHAR(100)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 apellido2	VARCHAR(100)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 ciudad	VARCHAR(100)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 categoria	INT(10)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Descripción tabla 'cliente_ctrol'

[illegible]