

Contenido

Introducción	2
Procedimientos vs Triggers vs Funciones	2
Qué compone un “programa” en MySQL.....	2
Palabras reservadas.....	2
Operadores del lenguaje.	3
Funciones del lenguaje.	5
Variables.....	6
Instrucciones.	6
Instrucciones declarativas.	6
Instrucciones alternativas simples – Bloque IF	6
Instrucción alternativa múltiple – CASE.....	7
Instrucciones repetitivas.	8
Procedimientos almacenados	9
Parámetros de entrada y salida.....	9
Definiendo procedimientos almacenados.	11
Ejemplo: Procedimiento que inserta una fila en una tabla.....	11
Ejemplo: Procedimiento con una variable de salida.	12
Ejemplo: Procedimiento con argumentos de entrada y salida.	13
Usos de los procedimientos almacenados.	13
Definiendo funciones.	13
Creación y borrado de funciones.....	14
Llamada a la función.....	14
Definiendo Triggers.	15
Ejemplos de Triggers.	16
Ejemplo de BEFORE en INSERT.	16
Ejemplo de BEFORE en UPDATE.	16
AFTER en UPDATE.	17
BEFORE en DELETE.	17
Cursores	18
Declaración del cursor.....	18
Apertura del cursor.	18
Lectura del cursor.....	18
Cierre del cursor.	19
Consideraciones sobre cursores.	19
Ejemplo de uso de un cursor.	19
Referencias.....	21



Introducción

Cuando se utiliza un SGBD es necesario en muchas ocasiones la programación de tareas que puedan servir tanto de soporte a la administración como a la gestión de los datos almacenados. Dependiendo del SGBD podemos encontrar:

- Procedimientos almacenados.
- Funciones.
- Disparadores o triggers.
- Eventos que funcionarán en virtud del [calendario de eventos del sistema](#)

Procedimientos vs Triggers vs Funciones

Trigger. Programa creado para ejecutarse automáticamente cuando ocurra un evento en nuestra base de datos

Procedimiento almacenado. Conjunto de instrucciones que se guardan en el servidor para utilizarlo cuando sea necesario. A diferencia de las funciones no devuelven ningún valor.

Función. Conjunto de instrucciones que se almacena en el servidor para realizar alguna tarea repetitiva. Podemos diferenciar:

- [Funciones del lenguaje](#)
- Funciones definidas por el usuario. Por ejemplo podemos crear una función que nos calcule la letra del DNI a partir de los datos numéricos de manera que podemos comprobar que los DNI que se insertan en una tabla son correctos antes de la inserción.

Qué compone un “programa” en MySQL

```
1 DELIMITER //
2 • CREATE PROCEDURE mostrarNumeros (IN n INT)
3 BEGIN
4     DECLARE i INT DEFAULT 1; -- CONTADOR
5     WHILE i <= n DO
6         SELECT i; -- mostramos el valor del contador
7         SET i:=i+1; -- lo incrementamos
8     END WHILE;
9 END//
10 DELIMITER ;
```

En un programa MySQL podemos encontrar palabras reservadas, variables, constantes, funciones e instrucciones SQL.

Palabras reservadas.

Palabras que forman parte del lenguaje. Pueden ser de SQL (SELECT, INSERT, DELETE, DROP,...) o aquellas que definen estructuras de programación. (IF, WHILE, FUNCTION, etc...)



Operadores del lenguaje.

Básicamente son similares a los de otros lenguajes con algunas ampliaciones debido al entorno en el que estamos trabajando, podemos clasificarlos en:

Aritméticos. Suma(+), resta (-), multiplicación (*), división (/) , división entera (DIV), resto (MOD), potencia (^).

Lógicos. Utilizando habitualmente para expresar condiciones compuestas, el resultado depende de la tabla de verdad del operador lógico utilizado.

Operador	Descripción
AND, &&	AND Lógico
NOT, !	Negación
OR,	OR lógico
XOR	OR exclusivo

De comparación. Permiten realizar comparaciones, dependiendo del tipo de las variables implicadas serán o no de aplicación.

Nombre del operador	Descripción
<u>!=, <></u>	Operador de distinto que
<u><</u>	Menor que
<u><=</u>	Menor o igual que
<u><=></u>	Igual a prueba de nulos, devuelve 1 si ambos operandos son nulos o 0 si uno de ellos es nulo.
<u>=</u>	Igual. Si algún operando es nulo devuelve NULL.
<u>></u>	Mayor que
<u>>=</u>	Mayor o igual que
<u>BETWEEN ... AND ...</u>	Comprueba si un valor está dentro del rango especificado
<u>COALESCE()</u>	Devuelve el primer argumento no nulo
<u>GREATEST()</u>	Devuelve el argumento más grande
<u>IN()</u>	Comprueba si el argumento está dentro de la lista de valores
<u>INTERVAL()</u>	Devuelve el índice del argumento que es menor que el primer argumento



Nombre del operador	Descripción
IS	Prueba un valor contra un booleano
IS NOT	Prueba si un valor contra un booleano
IS NOT NULL	Prueba si un valor no es nulo
IS NULL	Prueba si un valor es nulo
ISNULL()	Prueba si el argumento es nulo
LEAST()	Devuelve el menor de los argumentos
LIKE	Comparación de patrones
NOT BETWEEN ... AND ...	Comprueba si un valor no está dentro del rango especificado
NOT IN()	Comprueba si un valor no está dentro de la lista de valores pasada como argumento
NOT LIKE	Comparación de patrones. En este caso los patrones no deben coincidir para que devuelva cierto.
STRCMP()	Comparación de dos cadenas (string compare)

Asignación (:=)

El operador de asignación permite asignar el valor resultante de una expresión a una variable siempre que los tipos de datos sean compatibles. En MySQL la asignación va a compañada de la palabra reservada **SET**, por ejemplo **SET a:=b+c;**

Precedencia de operadores

Cuando en una expresión coinciden diferentes tipos de operadores estos se evalúan según su precedencia, en la siguiente imagen se pueden encontrar todos los operadores y la precedencia existente entre ellos.



```
1 INTERVAL
2 BINARY, COLLATE
3 !
4 - (menos unario, cambia el signo del argumento), ~ (inversión de bit unario)
5 ^
6 *, /, DIV, %, MOD
7 -, +
8 <<, >>
9 &
10 |
11 = (comparación), <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN
12 BETWEEN, CASE, WHEN, THEN, ELSE
13 NOT
14 AND, &&
15 XOR
16 OR, ||
17 = (asignación), :=
```

Cuanto más arriba más precedencia, es decir, el producto (*) tiene mayor precedencia, y por lo tanto se ejecuta antes, que el operador de negación (NOT). En el caso de que en una expresión coincidan operadores del mismo nivel de precedencia estos se evalúan de izquierda a derecha.

Funciones del lenguaje.

Se pueden dividir en diferentes tipos:

Funciones de cadena.

Funciones que permiten realizar operaciones con cadenas de caracteres. Puedes encontrar las función existentes en MySQL [en este enlace](#). Hay que tener en cuenta que aquellas funciones que trabajan con posiciones de caracteres identifican la primera posición por el número 1. En el caso de funciones que trabajan con argumentos de longitud los argumentos no enteros son redondeados al entero más próximo.

Funciones matemáticas.

Funciones que realizan operaciones matemáticas con los argumentos que reciben. Todas ellas devuelven NULL en caso de error. [En este enlace](#) encontrarás las funciones matemáticas de aplicación a MySQL.

Funciones de fecha y hora.

Conjunto de funciones que permiten realizar operaciones con fechas y horas. Al trabajar con ellas hay que tener en cuenta los rangos de valores en los que se mueve cada uno de los tipos de datos de fecha y de hora y los formatos en los que esos valores se especifican. Si haces clic [aquí](#) encontrarás las funciones de fecha y hora que se aplican en MySQL.



Variables.

Espacios de memoria que contienen un valor que:

- Recibe un nombre para identificarlo en un programa.
- Pertenece a un tipo de datos determinado lo que determina las operaciones que se pueden realizar sobre el valor.
- Cambia de valor durante el desarrollo del programa en función de las operaciones que se realizan con ella.
- Las variables en MySQL pueden pertenecer a tipos de datos que coinciden con los de [los campos permitidos en MySQL](#)

Instrucciones.

MySQL proporciona el mismo tipo de instrucciones que la mayoría de lenguajes, veamos brevemente la sintaxis de cada una de ellas.

Instrucciones declarativas.

Realizan la declaración de variables locales a programas almacenados.

- Una variable local es aquella que solo es accesible dentro del programa donde ha sido declarada.
- Para declarar variables locales se usa la sentencia DECLARE.

```
DECLARE var1 [, var2] ... tipo_de_datos [DEFAULT valor_por_defecto]
```

- La visibilidad de la variable quedará establecida entre los bloques BEGIN ... END entre los que esté declarada.
- para establecer su valor se utilizar la sentencia SET.

```
DECLARE i INT DEFAULT 1; -- CONTADOR
WHILE i <= n DO
    SELECT i; -- mostramos el valor del contador
    SET i:=i+1; -- lo incrementamos
END WHILE;
```

Bloques de instrucciones. BEGIN ... END

- Se utiliza para crear instrucciones compuestas
- Dentro de un bloque **BEGIN ... END** pueden existir un número indefinido de sentencias individuales separadas por el carácter de fin de instrucción (;)
- Los bloques **BEGIN ... END** pueden estar anidados

Instrucciones alternativas simples – Bloque IF

Definición:

```
IF condición THEN lista_de_sentencias
[ELSEIF condicion THEN lista_de_sentencias] ...
[ELSE lista_de_sentencias]
END IF
```



Ejemplos:

```

1  DELIMITER //
2  • CREATE FUNCTION devolverMayor (n INT, m INT) RETURNS INT
3  BEGIN
4      IF n>m THEN
5          RETURN n;
6      ELSE
7          RETURN m;
8      END IF;
9  END//
10 DELIMITER ;

```

```

1  DELIMITER //
2  • CREATE FUNCTION devolverMenor (n INT, m INT, i INT) RETURNS INT
3  BEGIN
4      IF n<m THEN
5          IF n<i THEN
6              return n;
7          ELSEIF i<m THEN
8              RETURN i;
9          ELSE
10             RETURN m;
11         END IF;
12     ELSEIF m<i THEN
13         RETURN m;
14     ELSE
15         RETURN i;
16     END IF;
17 END//
18 DELIMITER ;

```

La sentencia DELIMITER que se observa en los ejemplos resuelve un problema común en MySQL. Por defecto en MySQL el punto y coma se interpreta como un delimitador de sentencias por lo que es necesario redefinir el delimitador para que MySQL interprete el cuerpo de la función o procedimiento como un solo elemento. Antes de definir el procedimiento se establece un nuevo delimitador que es recuperado cuando finaliza el código del procedimiento. En este ejemplo el delimitador es //.

Instrucción alternativa múltiple – CASE**Definición:****Posibilidad 1:**

```

CASE valor_a_evaluar
WHEN valor THEN Lista_de_sentencias
[WHEN valor THEN Lista_de_sentencias] ...
[ELSE Lista_de_sentencias]
END CASE

```

Posibilidad 2:

```

CASE

```



```

    WHEN condición_de_búsqueda THEN Lista_de_sentencias
    [WHEN condición_de_búsqueda THEN Lista_de_sentencias]
    ...
    [ELSE Lista_de_sentencias]
END CASE

```

Ejemplo:

```

1  DELIMITER //
2  • CREATE FUNCTION convertir (n INT) RETURNS VARCHAR(4)
3  BEGIN
4      CASE n
5      WHEN 1 THEN RETURN "UNO";
6      WHEN 2 THEN RETURN "DOS";
7      WHEN 3 THEN RETURN "TRES";
8      ELSE
9          RETURN "CERO";
10     END CASE;
11 END//
12 DELIMITER ;

```

Instrucciones repetitivas.

Bucle LOOP

1. Ejecuta las instrucciones dispuestas en su interior hasta que:
 - a. Encuentra la instrucción ITERATE que provoca que vuelva a la primera instrucción del bucle.
 - b. Encuentra una instrucción LEAVE que provoca la salida del bucle.
2. En caso de que no se incluya la instrucción LEAVE generaremos un bucle infinito.
3. Para identificar el bucle este debe ser etiquetado.

```

1  DELIMITER //
2  • CREATE PROCEDURE mostrarNumerosLoop (IN n INT)
3  BEGIN
4      DECLARE i INT DEFAULT 1; -- CONTADOR
5      Bucle1: LOOP
6          SELECT i; -- mostramos el valor del contador
7          SET i:=i+1; -- lo incrementamos
8          IF i>n THEN
9              LEAVE Bucle1;
10         END IF;
11     END LOOP;
12 END//
13 DELIMITER ;

```

Bucle REPEAT

1. Ejecuta las instrucciones del bucle
2. Comprueba la condición
3. Si es CIERTA → Finaliza el bucle
4. Si es FALSA → Vuelve al punto 1




```

1  DELIMITER //
2  • CREATE PROCEDURE mostrarNumerosDorepeat (IN n INT)
3  BEGIN
4  DECLARE i INT DEFAULT 1; -- CONTADOR
5  REPEAT
6  SELECT i; -- mostramos el valor del contador
7  SET i:=i+1; -- lo incrementamos
8  UNTIL i>n END REPEAT;
9  END//
10 DELIMITER ;

```

Bucle WHILE

1º Comprueba la condición

2º Si es CIERTA

→ Ejecuta instrucciones del bucle

→ Itera

Si es FALSA → Finaliza el bucle

```

1  DELIMITER //
2  • CREATE PROCEDURE mostrarNumeros (IN n INT)
3  BEGIN
4  DECLARE i INT DEFAULT 1; -- CONTADOR
5  WHILE i <= n DO
6  SELECT i; -- mostramos el valor del contador
7  SET i:=i+1; -- lo incrementamos
8  END WHILE;
9  END//
10 DELIMITER ;

```

Procedimientos almacenados

Un procedimiento almacenado es un conjunto de instrucciones que se guardan en el servidor para utilizar cuando sean necesarias.

Se definen con la siguiente sentencia:

```

CREATE PROCEDURE nombreProcedimiento(arg1 tipo, arg2 tipo, ....) BEGIN

END PROCEDURE

```

Entre la sentencia de inicio y final de procedimiento encontraremos el cuerpo del programa formado por instrucciones del lenguaje.

Cada sentencia se separa de la siguiente mediante el símbolo de punto y coma (;) por lo tanto si usamos mysql cliente desde línea de comandos tendremos que cambiar el delimitador final de línea, para ello se utiliza la sentencia **delimiter**, por ejemplo **delimiter //** que cambia el delimitador a //

Parámetros de entrada y salida

Un parámetro es un dato necesario para el funcionamiento del procedimiento.



Pueden ser de entrada (IN), salida (OUT), entrada/salida (INOUT)

Deben ser de un tipo de datos definido.

Los parámetros de salida son variables que recogerán el resultado del procedimiento una vez que haya finalizado.

Los parámetros de entrada/salida sirven tanto para introducir datos al procedimiento como para recogerlos una vez que el procedimiento ha finalizado.

Si no se indica el tipo de parámetro por defecto será del tipo IN.

```
[{IN|OUT|INOUT} ] nombre TipoDeDato
```



Definiendo procedimientos almacenados.

Veamos un ejemplo, la declaración simple de un procedimiento con un parámetro de entrada.

Declaración.

```

1 DELIMITER //
2 • CREATE PROCEDURE mostrarNumeros (IN n INT)
3 BEGIN
4   DECLARE i INT DEFAULT 1; -- CONTADOR
5   WHILE i <= n DO
6     SELECT i; -- mostramos el valor del contador
7     SET i:=i+1; -- lo incrementamos
8   END WHILE;
9   END//
10 DELIMITER ;

```

Llamada.

```
CALL mostrarNumeros(8);
```

Borrado.

```
DROP PROCEDURE [IF EXISTS] nombre procedimiento;
```

Por ejemplo: **DROP PROCEDURE mostrarNumeros;**

IF EXISTS es opcional y sirve para indicar que se borre en caso de que el procedimiento exista y no indique error en caso de que el procedimiento no esté definido en el sistema.

Ejemplo: Procedimiento que inserta una fila en una tabla.

Vamos a crear un procedimiento que inserta datos en una tabla denominada Equipo.

1 • DESC equipo;

#	Field	Type	Null	Key	Default	Extra
1	nombreEquipo	varchar(20)	NO	PRI	<input type="text" value="NULL"/>	
2	directorEquipo	varchar(30)	NO	UNI	<input type="text" value="NULL"/>	
3	numeroCorredores	int(11)	NO		<input type="text" value="NULL"/>	

El procedimiento sería:



```

1 DELIMITER //
2 • CREATE PROCEDURE insertar(nuevoNombre VARCHAR(20), nombreDirector VARCHAR(30), numCorredores INT)
3   COMMENT 'Procedimiento que inserta un nuevo equipo en la tabla de ciclistas'
4 BEGIN
5   IF NOT EXISTS (SELECT nombreEquipo FROM equipo AS e WHERE e.nombreEquipo = nuevoNombre) THEN
6     INSERT INTO equipo VALUES (nuevoNombre, nombreDirector, numCorredores);
7   ELSE
8     SELECT 'Este cliente ya existe en la base de datos!';
9   END IF;
10 END//
11
12 DELIMITER ;

```

Un ejemplo de ejecución:

Query 1 SQL File 3*

Limit to 1000 rows

```

1 • CALL insertar('Bicivoladores', 'Perico Delgado', 10);
2 • SELECT * FROM equipo;

```

Result Grid Filter Rows: Edit: Export/Import:

#	nombreEquipo	directorEquipo	numeroCorredores
1	Bicivoladores	Perico Delgado	10
*	NULL	NULL	NULL

Ejemplo: Procedimiento con una variable de salida.

```

1 DELIMITER //
2
3 • CREATE PROCEDURE contarEquipos (OUT num INT)
4   COMMENT 'Procedimiento que nos devuelve el número de filas de la tabla de equipos'
5 BEGIN
6   SELECT count(*) FROM equipos INTO num;
7 END//
8
9 DELIMITER ;

```

Ejemplo de ejecución:

Result Grid Filter Rows: Export: Wrap Cell Content:

#	@miVariable
1	30

Result 1

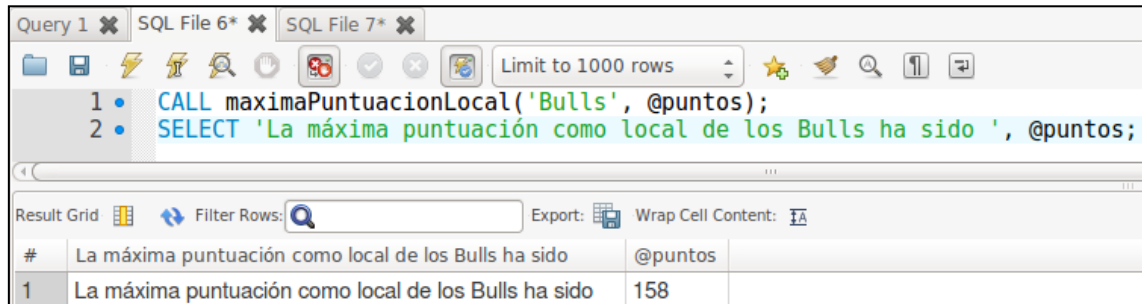
Ejemplo: Procedimiento con argumentos de entrada y salida.

```

1 DELIMITER //
2
3 • CREATE PROCEDURE maximaPuntuacionLocal (nombreEquipo VARCHAR(20), OUT puntuacion INT)
4 COMMENT 'Procedimiento que nos devuelve el número de filas de la tabla de equipos'
5 BEGIN
6 # primero comprobamos que el equipo exista
7 IF EXISTS (select equipo_local FROM partidos WHERE equipo_local like nombreEquipo) THEN
8 SELECT MAX(puntos_local) FROM partidos WHERE equipo_local like nombreEquipo INTO puntuacion;
9 ELSE
10 Set puntuacion:=0;
11 END IF;
12 END//
13
14 DELIMITER ;

```

Ejemplo de ejecución:



Query 1 SQL File 6* SQL File 7*

Limit to 1000 rows

```

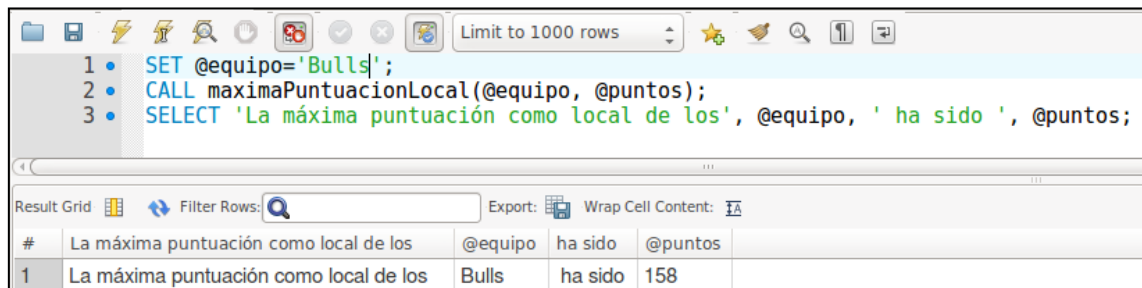
1 • CALL maximaPuntuacionLocal('Bulls', @puntos);
2 • SELECT 'La máxima puntuación como local de los Bulls ha sido ', @puntos;

```

Result Grid Filter Rows: Export: Wrap Cell Content:

#	La máxima puntuación como local de los Bulls ha sido	@puntos
1	La máxima puntuación como local de los Bulls ha sido	158

Otra posible llamada al procedimiento:



Limit to 1000 rows

```

1 • SET @equipo='Bulls';
2 • CALL maximaPuntuacionLocal(@equipo, @puntos);
3 • SELECT 'La máxima puntuación como local de los', @equipo, ' ha sido ', @puntos;

```

Result Grid Filter Rows: Export: Wrap Cell Content:

#	La máxima puntuación como local de los	@equipo	ha sido	@puntos
1	La máxima puntuación como local de los	Bulls	ha sido	158

Usos de los procedimientos almacenados.

- Tareas repetitivas que debemos realizar cada cierto tiempo. Por ejemplo dar de baja a clientes con fecha de finalización del contrato vencida.
- Comprobar integridad de una base de datos. P.e: Revisamos que no existan datos que no tengan coherencia.
- Gestión de transacciones.
- Automatización de tareas de gestión. P.e: Paso de datos vigentes de una base de datos a tablas históricas

Definiendo funciones.

Para definir funciones en MySQL debemos tener en cuenta las siguientes reglas:

- Una función es un conjunto de instrucciones que tomando unos valores de entrada los procesan y nos devuelven un valor de salida.
- Los parámetros de entrada deben ser de tipo IN.
- Deben devolver un único valor perteneciente a algún tipo de dato permitido en el lenguaje.
- Pueden usarse dentro de sentencias SQL.



Creación y borrado de funciones.

Las funciones se crean mediante la instrucción CREATE FUNCTION:

```
CREATE FUNCTION nombre_función ([parámetros[,...]])
    RETURNS Tipo_de_dato_del_valor_de_retorno
    [atributos de la rutina ...]
BEGIN
    cuerpo_de_la_función
END
```

Y Se eliminan con DROP FUNCTION.

```
DROP FUNCTION [IF EXISTS] nombre_función;
```

Podemos ver un ejemplo de creación de una función que calcula el factorial de un número entero.

```
1  DELIMITER //
2  • CREATE FUNCTION factorial (n INT) RETURNS INT
3  BEGIN
4  DECLARE factorial, multiplicador INT;
5  SET factorial := n; #establecemos factorial a valor del número recibido
6  IF factorial > 0 THEN
7      SET multiplicador = n-1;
8      WHILE multiplicador > 1 DO
9          SET factorial := factorial * multiplicador;
10         SET multiplicador:=multiplicador-1;
11     END WHILE;
12     RETURN factorial;
13 ELSE
14     RETURN 1;
15 END IF;
16 END//
17
18 DELIMITER ;
```

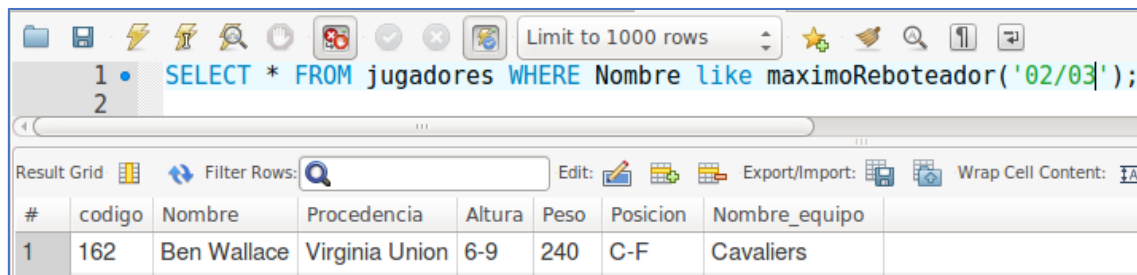
Llamada a la función.

Se puede llamar a la función desde procedimientos, triggers, otras funciones e incluso desde sentencias SQL. Por ejemplo supongamos la siguiente función:

```
1  DELIMITER //
2  • CREATE FUNCTION maximoReboteador (temp VARCHAR(5)) RETURNS VARCHAR(30)
3  BEGIN
4  DECLARE maximoRebotes FLOAT;
5  DECLARE codigoReboteador INT;
6  DECLARE nombreReboteador VARCHAR(30);
7  SELECT MAX(Rebotes_por_partido) FROM estadisticas WHERE temporada like temp INTO maximoRebotes;
8  SELECT jugador FROM estadisticas
9  WHERE ROUND(rebotes_por_partido,1) = ROUND(maximoRebotes,1)
10 INTO codigoReboteador;
11 SELECT Nombre FROM jugadores WHERE codigo = codigoReboteador INTO nombreReboteador;
12 RETURN nombreReboteador;
13 END//
14
15 DELIMITER ;
```

Por ejemplo podemos llamarla desde una sentencia SQL:





Definiendo Triggers.

Un **Trigger** es un tipo de programa almacenado creado para utilizarse de manera automática cuando ocurre un evento en nuestra base de datos. Los eventos se generarán mediante los comandos:

- INSERT
- UPDATE
- DELETE

El objetivo principal es proteger la integridad de nuestras tablas validando información, calcular datos derivados, seguir movimientos de bases de datos, etc...

Es importante tener en cuenta que para ejecutarse requieren el privilegio SUPER y TRIGGER.

Utiliza una sintaxis similar a la creación de procedimientos y funciones. Su prototipo es:

```
CREATE
[DEFINER = { usuario | CURRENT_USER }]
TRIGGER nombre_del_trigger { BEFORE | AFTER } { INSERT | UPDATE | DELETE }
ON nombre_de_la_tabla FOR EACH ROW
BEGIN
cuerpo_del_trigger
END
```

En el que:

DEFINER. Usuario que tiene privilegios para ejecutar el trigger. Por defecto será el usuario que crea el trigger.

BEFORE/AFTER. Indica si el trigger se ejecutará antes o después de la operación que provoca su ejecución.

UPDATE|INSERT|DELETE. Operación que provoca la ejecución.

Para ver la información sobre un trigger podemos ejecutar:

```
SHOW CREATE TRIGGER nombre_del_trigger;
```

También podemos ver los TRIGGERS definidos en el sistema mediante:

```
SHOW TRIGGERS;
```

Y borrar un trigger con:

```
DROP TRIGGER [IF EXISTS] nombre_trigger;
```



Significado de los identificadores NEW y OLD en Triggers:

Cuando usamos una sentencia **UPDATE** en el trigger podemos referirnos a los valores anteriores a la actualización anteponiendo la palabra OLD seguida del nombre del campo y a los nuevos usando la palabra NEW después del nombre del campo

OLD.nombreColumna

NEW.nombreColumna

En el caso de sentencias **INSERT** solo se puede usar NEW.

En el caso de sentencias **DELETE** solo usaremos OLD.

Ejemplos de Triggers.

En los ejemplos se trabajará sobre una tabla con la siguiente definición:

```
1 CREATE TABLE datosEmpleados (
2   codigoEmpleado INT(4) NOT NULL PRIMARY KEY,
3   nombreEmpleado VARCHAR(12) NOT NULL,
4   Apellido1Empleado VARCHAR(15) NOT NULL,
5   Apellido2Empleado VARCHAR(15),
6   sueldoEmpleado FLOAT(6,2),
7   DNIEmpleado VARCHAR(9));|
```

Y que tiene como datos:

#	codigoEmpleado	nombreEmpleado	Apellido1Empleado	Apellido2Empleado	sueldoEmpleado	DNIEmpleado
1	1	Cristiano	Romualdo	Santos	1800.00	12345678Z
2	2	Leopardo	Messi		1950.00	00234567J

Ejemplo de BEFORE en INSERT.

Comprueba que no se insertan valores negativos de sueldo en la tabla.

```
1 DELIMITER //
2 CREATE TRIGGER datosEmpleados_BI
3 BEFORE INSERT ON datosEmpleados FOR EACH ROW
4 BEGIN
5   IF NEW.sueldoEmpleado < 0 THEN
6     signal sqlstate '45000' SET message_text='El sueldo no puede ser negativo';
7   END IF;
8 END//
9 DELIMITER ;
```

Signal – Permite devolver un error dentro de un proceso. El código de error '45000' está asignado para "Excepciones definidas por el usuario".

Ejemplo de ejecución:

```
mysql> INSERT INTO datosEmpleados VALUES(3,'Yale', 'Bale', null, -3000, '3213123T');
ERROR 1644 (45000): El sueldo no puede ser negativo
```

Ejemplo de BEFORE en UPDATE.

Comprueba que no se actualiza el sueldo a valores negativos.




```

1 DELIMITER //
2 • CREATE TRIGGER datosEmpleados_BU
3 BEFORE UPDATE ON datosEmpleados FOR EACH ROW
4 BEGIN
5     IF NEW.sueldoEmpleado < 0 THEN
6         signal sqlstate '45000' SET message_text='El sueldo no puede ser negativo';
7     END IF;
8 END//
9 DELIMITER ;

```

Ejemplo de ejecución:

```

mysql> UPDATE datosEmpleados SET sueldoEmpleado='-3000' WHERE codigoEmpleado=1;
ERROR 1644 (45000): El sueldo no puede ser negativo

```

AFTER en UPDATE.

Guardamos un log de la operación en otra tabla llamada historicoDatosEmpleados cuya definición es:

```

1 • CREATE TABLE historicoDatosEmpleados (
2     codigoEmpleado INT(4) NOT NULL PRIMARY KEY,
3     nombreEmpleado VARCHAR(12) NOT NULL,
4     Apellido1Empleado VARCHAR(15) NOT NULL,
5     Apellido2Empleado VARCHAR(15),
6     sueldoEmpleado FLOAT(6,2),
7     DNIEmpleado VARCHAR(9),
8     accionRealizada VARCHAR(6),
9     fecha DATETIME);

```

El código sería:

```

1 DELIMITER //
2 • CREATE TRIGGER datosEmpleados_AU
3 AFTER UPDATE ON datosEmpleados FOR EACH ROW
4 BEGIN
5     INSERT INTO historicoDatosEmpleados
6     VALUES(OLD.codigoEmpleado, OLD.nombreEmpleado,
7             OLD.apellido1Empleado, OLD.apellido2Empleado,
8             OLD.sueldoEmpleado, OLD.DNIEmpleado, 'UPDATE', NOW());
9 END//
10 DELIMITER ;

```

Un ejemplo de ejecución podría ser:

```

1 • UPDATE datosEmpleados SET sueldoEmpleado='3000' WHERE codigoEmpleado=1;
2 • SELECT * FROM historicoDatosEmpleados;
3

```

#	codigoEmpleado	nombreEmpleado	Apellido1Empleado	Apellido2Empleado	sueldoEmpleado	DNIEmpleado	accionRealizada	fecha
1	1	Cristiano	Romualdo	Santos	2000.00	12345678Z	UPDATE	2017-02-02 17:21:18

BEFORE en DELETE.

Hace copia de la fila borrada en una tabla histórica.

El código sería:



```

DELIMITER //
CREATE TRIGGER datosEmpleados_BD
BEFORE DELETE ON datosEmpleados FOR EACH ROW
BEGIN
INSERT INTO historicoDatosEmpleados
VALUES(OLD.codigoEmpleado, OLD.nombreEmpleado,
      OLD.apellido1Empleado, OLD.apellido2Empleado,
      OLD.sueldoEmpleado, OLD.DNIEmpleado, 'DELETE', NOW());
END//
DELIMITER ;

```

En ejecución:

1	DELETE FROM datosEmpleados WHERE codigoEmpleado=1;
2	SELECT * FROM historicoDatosEmpleados;

#	codigoEmpleado	nombreEmpleado	Apellido1Empleado	Apellido2Empleado	sueldoEmpleado	DNIEmpleado	accionRealizada	fecha
1	1	Cristiano	Romualdo	Santos	3000.00	12345678Z	DELETE	2017-02-02 17:34:53
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Cursores

Puntero a las filas que devuelve una consulta, permite modificar los datos de cada fila de manera individual. Funcionan mediante cuatro estados.

- Declaración
- Apertura
- Lectura
- Cierre

Declaración del cursor.

La declaración del cursor asigna un puntero a la dirección en memoria del primer resultado de dicha consulta

```

DECLARE cursor_empleados CURSOR FOR
SELECT codigoEmpleado, nombreEmpleado, SueldoEmpleado
FROM empleados;

```

Apertura del cursor.

Para poder utilizar el cursor no basta con declararlo sino que en el momento en el que vamos a usarlo debemos abrirlo

```

OPEN cursor_empleados;

```

Lectura del cursor.

Nos permite pasar por las filas de la consulta una a una asignando su contenido a variables que habremos definido para tal fin.

Se utiliza el comando FETCH.



```

DECLARE codigo INT(4);
DECLARE nombre VARCHAR(12);
DECLARE sueldo FLOAT(6,0);

FETCH cursor_empleado INTO codigo, nombre, sueldo;

```

Cada vez que utilizamos el comando FETCH el cursor apunta a la siguiente fila de resultados.

Es necesario controlar cuándo hemos superado la última fila apuntada por el cursor. Para ello utilizaremos un manejador de error (HANDLER) que nos indicará cuándo el cursor ya no tiene más filas donde apuntar.

```

DECLARE CONTINUE HANDLER FOR NOT FOUND SET final=TRUE;

```

En la anterior declaración indicamos al sistema que si el cursor no encuentra fila a la que apuntar (NOT FOUND) establezca la variable finCursor a CIERTO. El contenido de esta variable nos indicará entonces:

- Ya no hay filas que leer en el cursor → TRUE
- Quedan filas en el cursor → FALSE

Cierre del cursor.

Una vez finalizada la lectura es preciso liberar el cursor mediante **CLOSE**.

Consideraciones sobre cursores.

Debemos tener en cuenta que:

- Los cursores solamente se leen desde el primer elemento hacia el último. En caso de desear otro tipo de ordenación es preciso utilizar ORDER BY en la cláusula SELECT del cursor
- No es posible ir directamente a una de las filas devueltas por el cursor sin leer las anteriores. Siempre podemos no tratar determinadas filas

Ejemplo de uso de un cursor.

La tabla **jugadores** de la base de datos NBA tiene el campo altura en pies. Queremos convertir dicho campo a centímetros para un equipo determinado. La definición y datos contenidos en la tabla se muestran a continuación:

Definición de la tabla.

#	Field	Type	Null	Key	Default	Extra
1	codigo	int(11)	NO	PRI	NULL	
2	Nombre	varchar(30)	YES		NULL	
3	Procedencia	varchar(20)	YES		NULL	
4	Altura	varchar(4)	YES		NULL	
5	Peso	int(11)	YES		NULL	
6	Posicion	varchar(5)	YES		NULL	
7	Nombre_equipo	varchar(20)	YES	MUL	NULL	



Datos almacenados:

codigo	Nombre	Procedencia	Altura	Peso	Posicion	Nombre_equipo
1	Corey Brever	Florida	6-9	185	F-G	Timberwolves
2	Greg Buckner	Clemson	6-4	210	G-F	Timberwolves
3	Michael Doleac	Utah	6-11	262	C	Timberwolves
4	Randy Foye	Villanova	6-4	213	G	Timberwolves
5	Ryan Gomes	Providence	6-7	250	F	Timberwolves
6	Marko Jaric	Serbia	6-7	224	G	Timberwolves
7	Al Jefferson	Prentiss Hs	6-10	265	C-F	Timberwolves

En primer lugar vamos a crear un función que convierte de pies a centímetros. La función actúa recibiendo como argumento de entrada un VARCHAR(4) que contiene la altura en pies y devolviendo otro VARCHAR(4) pero con la altura en centímetros.

```

1  DELIMITER //
2  • CREATE FUNCTION convertirAltura (alturaOriginal VARCHAR(4)) RETURNS VARCHAR(4)
3  BEGIN
4      DECLARE pies, pulgadas, posicionGuion, centimetros INT;
5      # DESCOMONEMOS LA ALTURA EN PIES Y PULGADAS
6      SET posicionGuion:=LOCATE('-', alturaOriginal);
7      SET pies:=CAST(SUBSTRING(alturaOriginal,1,1) AS UNSIGNED);
8      SET pulgadas:=CAST(SUBSTRING(alturaOriginal, posicionGuion+1, 2) AS UNSIGNED);
9      # pasamos el cálculo a centímetros
10     SET centimetros := pies *30.48 + pulgadas * 2.54;
11     RETURN centimetros;
12 END//
13 DELIMITER ;

```

Una vez creada la función definimos un procedimiento almacenado que va recorriendo las filas de la tabla y actualizando valores.

```

1  DELIMITER //
2  • CREATE PROCEDURE cambiarAltura (equipo VARCHAR(20))
3  BEGIN
4      #declaramos variables
5      DECLARE final BOOLEAN DEFAULT FALSE; # para el manejador del cursor
6      DECLARE codJugador INT; # recoge código de jugador para el cursor
7      DECLARE altJugador VARCHAR(4); # recoge altura de jugador para el cursor
8      #declaramos el cursor
9      DECLARE cursor_consulta CURSOR FOR
10     SELECT codigo, Altura FROM jugadores WHERE Nombre_equipo like equipo;
11     # declaramos el manejador de error
12     DECLARE CONTINUE HANDLER FOR NOT FOUND SET final=TRUE;
13     OPEN cursor_consulta; #abrimos cursor
14     WHILE NOT final DO #mientras haya registros en el cursor
15         FETCH cursor_consulta INTO codJugador, altJugador;
16         SET altJugador:=convertirAltura(altJugador); #convertimos la altura a centímetros
17         UPDATE jugadores SET Altura = altJugador WHERE codJugador=codigo; #actualizamos la tabla
18     END WHILE;
19     CLOSE cursor_consulta;
20 END//
21 DELIMITER ;

```

Una vez implementado solo nos queda llamar al procedimiento y ver cómo se ha producido los cambios:



```

1 • CALL cambiarAltura('Timberwolves');
2 • SELECT * FROM jugadores WHERE Nombre_Equipo like 'Timberwolves';
3

```

#	codigo	Nombre	Procedencia	Altura	Peso	Posicion	Nombre_equipo
1	1	Corey Brever	Florida	206	185	F-G	Timberwolves
2	2	Greg Buckner	Clemson	193	210	G-F	Timberwolves
3	3	Michael Doleac	Utah	211	262	C	Timberwolves
4	4	Randy Foye	Villanova	193	213	G	Timberwolves
5	5	Ryan Gomes	Providence	201	250	F	Timberwolves
6	6	Marko Jaric	Serbia	201	224	G	Timberwolves
7	7	Al Jefferson	Prentiss Hs	208	265	C-F	Timberwolves
8	8	Mark Madsen	Stanford	206	250	C-F	Timberwolves
9	9	Rashard McCants	North Carolina	193	21	G	Timberwolves
10	10	Chris Richard	Florida	206	270	F	Timberwolves
11	11	Craig Smith	Boston Coll...	203	250	F-C	Timberwolves
12	12	Kirk Snyder	Nevada-Reno	198	225	G	Timberwolves

Referencias

MySQL Server Reference Manual - <https://dev.mysql.com/doc/refman/5.7/en/>

