



# MySQL

## ▼ Tipos de datos

### ▼ Numéricos:

**En las explicaciones que se dan a continuación la N indica el número total de dígitos y la D, el número de dígitos después de la coma.**

#### ▼ Enteros:

- TINYINT[(N)]: entre -128 y 127 con signo y entre 0 y 255 sin signo.
- SMALLINT[(N)]: entre -32.768 y 32.767 con signo y entre 0 y 65535 sin signo.
- MEDIUMINT[(N)]: entre -8.388.608 y 8.388.607 con signo y entre 0 y 16.777.215 sin signo.
- INT[(N)]: entre -2.147.483.648 y 2.147.483.647 con signo y entre 0 y 4.294.967.295 sin signo.
- BIGINT[(N)]: entre -9.223.372.036.854.775.808 y 9.223.372.036.854.775.807 con signo y entre 0 y 18.446.744.073.709.551.615 sin signo.

#### ▼ Reales:

- DECIMAL(M, D):
- FLOAT[(N,D)]: desde -3.402823466E+38 hasta -1.175494351E-38 y sin signo desde 1.175494351E-38 hasta 3.402823466E+38
- DOUBLE[(N,D)]: desde -1.7976931348623157E+308 hasta -2.2250738585072014E-308 y sin signo desde 2.2250738585072014E-308 hasta 1.7976931348623157E+308

▼ Booleanos:

BOOLEAN / BOOL: Un valor 0 se considera falso y un valor diferente de 0, verdadero.

▼ Binarios:

**En las explicaciones que se dan a continuación la M indica el número total de caracteres**

- BIT(M): Sirve para almacenar cadenas de bits.

El valor se le da de la siguiente manera:

SET variable = b'valor' → El valor debe ser en bits(es decir solo ceros y unos)

ejemplo:

```
CREATE TABLE t (b BIT(8));
```

```
INSERT INTO t SET b = b'101001';
```

- BLOB[(M)]: Permite almacenar objetos binarios de longitud variable, siendo la longitud máxima permitida en bytes la especificada en M. El rango de M es de 1 a 65.535 bytes. Para los atributos con tipo BLOB no es posible especificar valor por defecto.
- TINYBLOB[(M)]: Permite almacenar objetos binarios pequeños de longitud variable, siendo la longitud máxima permitida en bytes la especificada en M. El rango de M es de 1 a 255 bytes.
- MEDIUMBLOB[(M)]: Permite almacenar objetos binarios de tamaño medio y longitud variable, siendo la longitud máxima permitida en bytes la especificada en M. El rango de M es de 1 a 16.777.215 bytes

- LONGBLOB[(M)]: Permite almacenar objetos binarios grandes de longitud variable, siendo la longitud máxima permitida en bytes la especificada en M. El rango de M es de 4 GB (4.294.967.295 bytes).

▼ Alfanuméricos:

**En las explicaciones que se dan a continuación la M indica el número total de caracteres**

▼ Texto libre:

- CHAR[(M)]: Sirve para almacenar cadenas de caracteres de longitud fija, esto es, cadenas que siempre ocupan el número de caracteres especificado en M.. Si no se especifica M, la longitud por defecto es 1. Por ello, el tipo CHAR es sinónimo de CHAR(1). Si la cadena que se asigna a un dato con tipo de dato CHAR(M) tiene una longitud menor que M, se llenará con espacios en blanco a la derecha hasta alcanzar la longitud M. El rango de M es de 0 a 255 caracteres.
- VARCHAR(M): Permite almacenar cadenas de caracteres de longitud variable. El rango de M va desde 0 hasta 65.535.
- TEXT[(M)]: Permite almacenar cadenas de caracteres con una longitud máxima de 65.535 caracteres. Para los atributos con tipo TEXT no es posible especificar valor por defecto.
- TINYTEXT[(M)]: Permite almacenar cadenas de caracteres con una longitud máxima de 255 caracteres.
- MEDIUMTEXT[(M)]: Permite almacenar cadenas de caracteres con una longitud máxima de 16.777.215 caracteres.
- LONGTEXT[(M)]: Permite almacenar cadenas de caracteres con una longitud máxima de 4.294.967.295 caracteres

▼ Texto:

- ENUM('valor1', 'valor2', . . .): Permite almacenar solo uno de los valores especificados como 'valor1', 'valor2', etc. El número máximo de valores que se pueden especificar es 65535.

- SET('valor1', 'valor2', . . .): Permite almacenar cero, uno o más valores de los especificados como 'valor1', 'valor2', etc. El número máximo de valores que se pueden especificar es 64

▼ Fecha/Hora:

- DATE: Permite almacenar una fecha en el rango de '1001-01-01' a '9999-12-31'. Como se puede observar, las fechas se almacenan en el formato 'AAAA-MM-DD'
- TIME: Permite almacenar una hora en el rango de '-838:59:59' a '838:59:59'.
- DATETIME: Permite almacenar una fecha y una hora. Los valores permitidos oscilan entre '1001-01-01 00:00:00' a '9999-12-31 23:59:59'. Como se puede observar, estos datos se almacenan siguiendo el formato 'AAAA-MM-DD HH:MM:SS'.
- TIMESTAMP: Permite almacenar una fecha y una hora. Los valores permitidos oscilan entre '1970-01-01 00:00:01' a '2038-01-19 03:14:07'. Se puede asignar como valor por defecto a un campo de este tipo el valor CURRENT\_TIMESTAMP, que hace referencia a la fecha y hora actual del sistema.
- YEAR[(2)]: Sirve para almacenar años. Admite valores entre 1901 y 2155 y también el 0000. Si le pone el parámetro (2) solo mostrará los últimos 2 dígitos.

▼ Espaciales:

GEOMETRY / POINT / LINESTRING / POLYGON / GEOMETRYCOLLECTION / MULTILINESTRING / MULTIPOINT / MULTIPOLYGON

- JSON

▼ Creación db/table

## Creación de un DB:

`CREATE DATABASE my_db;`

SHOW CREATE DATABASE my\_db2; → Mostrará el código completo de la creación de la db el cual incluirá un CHARACTER SET y un COLLATE, el character especifica el tipo de unicode usado y el collate especifica unos ajustes específicos de ese unicode, por ejemplo si diferencia mayúsculas y minúsculas o no.

Crear una db con un COLLATE concreto:

```
CREATE DATABASE my_db DEFAULT CHARACTER SET utf8 COLLATE  
utf8_unicode_ci ;
```

Crear una tabla con un COLLATE concreto:

```
CREATE TABLE PERSONA (  
    DNI CHAR(9) PRIMARY KEY,  
    Nombre VARCHAR(20) NOT NULL,  
    Apellido VARCHAR(25) NOT NULL,  
) CHARACTER SET utf8 COLLATE utf8_unicode_ci ;
```

Crear una columnas con un COLLATE concreto:

```
CREATE TABLE PERSONA (  
    DNI CHAR(9) PRIMARY KEY CHARACTER SET utf8 COLLATE  
    utf8_unicode_ci ,  
    Nombre VARCHAR(20) NOT NULL CHARACTER SET latin1 COLLATE  
    latin1_german1_ci ,  
    Apellido VARCHAR(25) NOT NULL CHARACTER SET utf8 COLLATE  
    utf8_polish_ci,  
);
```

Modificar el COLLATE de una db:

```
ALTER DATABASE my_db CHARACTER SET utf8mb4 COLLATE  
utf8mb4_unicode_ci;
```

Modificar el COLLATE de una tabla:

```
ALTER TABLE PERSONA CONVERT TO CHARACTER SET utf8mb4 COLLATE  
utf8mb4_unicode_ci;
```

Modificar el COLLATE de una columna:

```
ALTER TABLE PERSONA MODIFY Nombre VARCHAR(20) CHARACTER SET  
utf8mb4 COLLATE utf8mb4_unicode_ci;
```

## Creación de Tablas:

```
CREATE TABLE my_table;  
  
CREATE TABLE my_table2 (  
    id_cliente INT NOT NULL AUTO_INCREMENT,  
    PRIMARY KEY (id_cliente),  
    nombre VARCHAR (100),  
    apellido VARCHAR (100),  
    edad INT,  
    telefono INT,  
    email VARCHAR(100)  
);  
  
CREATE TABLE pedidos (  
    id_pedido INT NOT NULL AUTO_INCREMENT,  
    PRIMARY KEY (id_pedido),  
    fecha DATE,  
    cantidad INT,  
    id_cliente INT,  
    id_producto INT,  
    FOREIGN KEY (id_cliente) REFERENCES clientes(id_cliente), → Los Foreign  
    keys siempre van abajo del todo  
    FOREIGN KEY (id_producto) REFERENCES productos(id_producto)  
);
```

### ▼ Crear usuarios

```
CREATE USER 'Nadir' @'localhost' IDENTIFIED BY 'Julio369';
```

# Privilegios

Para ver los privilegios de un usuario:

```
SHOW GRANTS FOR 'nadir' @'localhost';
```

Para agregar privilegios a un usuario:

```
GRANT privilegio ON my_db.* TO 'nadir' @'localhost';
```

```
GRANT privilegio ON my_db.my_table TO 'nadir' @'localhost';
```

Lista de todos los [privilegios](#):

MySQL :: MySQL 8.0 Reference Manual :: 6.2.2 Privileges Provided by MySQL

The privileges granted to a MySQL account determine which operations the account can perform. MySQL privileges differ in the contexts in which they apply and at different levels of operation: Administrative privileges enable users to manage operation of the MySQL

 <https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html>

Para eliminar privilegios a un usuario:

```
REVOKE privilegio ON my_db.my_table FROM 'nadir' @'localhost';
```

## ▼ Manipulando db/table/column/data

# DB

Eliminación de una base de datos:

```
DROP DATABASE my_db;
```

# TABLE

Creación de una tabla a partir de otra u otras:

```
CREATE TABLE nuevatabla AS SELECT * FROM otratabla;
```

```
CREATE TABLE nuevatabla AS SELECT column1, column2, column3  
FROM otratabla1 AS t1, otratabla2 AS t2 WHERE t1.id = t2.id ;
```

Eliminación de una tabla:

```
DROP TABLE my_table;
```

Renombrar una tabla:

```
ALTER TABLE my_table RENAME coches;
```

## COLUMN

Creación de una columna:

```
ALTER TABLE my_table ADD COLUMN dirección VARCHAR(30);
```

Eliminación de una columna:

```
ALTER TABLE my_table DROP COLUMN dirección;
```

Renombrar una columna:

```
ALTER TABLE my_table CHANGE telefono celular INT(11); → Cambia el nombre de la columna telefono por celular y el tipo de dato es el mismo pero aún así por sintaxis hay que volver a especificarlo
```

Cambiar el tipo de dato de una columna:

```
ALTER TABLE my_table MODIFY celular INT(20); → Cuando solo se quiere cambiar el tipo de dato de la columna pero no el nombre
```

Convertir una columna en un Primary key:

```
ALTER TABLE my_table ADD PRIMARY KEY (id);
```

Convertir una columna en un Foreign Key:

```
ALTER TABLE my_table ADD CONSTRAINT FOREIGN KEY (id_fk)  
REFERENCES my_table2(id);
```

## DATA

Insertar un registro:

```
INSERT INTO clientes (nombre, telefono)VALUES('Nadir', '632170581');
```

```
INSERT INTO clientes SET nombre = 2 , telefono = '632170581';
```

```
INSERT INTO clientes SELECT 2 , '632170581';
```

Eliminar un registro (toda una fila):

```
DELETE FROM clientes WHERE id=2;
```

Modificar un dato de una columna:

```
UPDATE clientes SET telefono='632170587' WHERE telefono='632170581';
```

## ▼ SELECT

# Claúsula SELECT

Guardar el resultado de una operación aritmética en una nueva columna o en una ya existente:

```
SELECT 2+2*2-2 AS calculo;
```

## WHERE

```
SELECT * FROM clientes;
```

```
SELECT id_cliente,edad FROM clientes WHERE telefono='632170587';
```

```
SELECT id_cliente,edad FROM clientes WHERE upper(apellido) =  
upper('boughlala');
```

```
SELECT * FROM pedidos WHERE cantidad >= 15;
```

```
SELECT * FROM pedidos WHERE id_producto <> 4;
```

## LIKE

```
SELECT * FROM clientes WHERE upper(nombre) LIKE upper('Nad%');
```

```
SELECT * FROM clientes WHERE email LIKE '%gmail.com';
```

```
SELECT * FROM clientes WHERE nombre LIKE '%el%';
```

```
SELECT nombre FROM clientes WHERE nombre LIKE '_a_i_';
```

## BETWEEN

```
SELECT * FROM pedidos WHERE cantidad BETWEEN 10 AND 15; //Siempre  
poner la cantidad más pequeña primero
```

```
SELECT * FROM pedidos WHERE cantidad NOT BETWEEN 10 AND 15;
```

## IN

```
SELECT * FROM clientes WHERE upper(nombre) IN  
(upper('Nadir'),upper('Luis'),upper('Urko'));
```

```
SELECT nombre FROM clientes WHERE id_cliente IN (SELECT id_cliente FROM pedidos WHERE cantidad>=50);
```

```
SELECT * FROM clientes WHERE upper(nombre) NOT IN (upper('Nadir'),upper('Luis'),upper('Urko'));
```

## **NULL**

```
SELECT * FROM pedidos WHERE cantidad IS NULL;
```

```
SELECT * FROM pedidos WHERE cantidad IS NOT NULL;
```

## **EXIST**

```
SELECT * FROM clientes WHERE EXIST (SELECT cantidad FROM pedidos WHERE cantidad>=100);
```

## **INTO**

```
SELECT c.nombre, c.apellido, pr.nombre, pr.precio, pe.cantidad, pr.precio*pe.cantidad AS precio_total INTO nuevatabla1 FROM clientes AS c, productos AS pr, pedidos AS pe WHERE pe.id_producto=pr.id_producto;
```

## **DISTINCT**

```
SELECT DISTINCT edad FROM clientes;
```

## **GROUP BY (MIN,MAX,SUM,COUNT,AVG)**

```
SELECT pr.nombre, SUM(pe.cantidad) AS suma FROM pedidos AS pe, productos AS pr WHERE pe.id_producto=pr.id_producto GROUP BY pr.nombre;
```

## **ORDER BY**

```
SELECT nombre, precio FROM productos ORDER BY precio ASC;  
SELECT nombre, precio FROM productos ORDER BY precio DESC;
```

## **HAVING**

```
SELECT pr.nombre, SUM(pe.cantidad) AS suma FROM pedidos AS pe, productos AS pr WHERE pe.id_producto=pr.id_producto GROUP BY pr.nombre HAVING SUM(pe.cantidad)>9;
```

```
SELECT id_cliente, COUNT(id_cliente) AS numduplicados FROM clientes  
GROUP BY id_cliente HAVING COUNT(id_cliente)>1;
```

## LIMIT/OFFSET

```
SELECT * FROM productos ORDER BY precio DESC LIMIT 5;  
SELECT * FROM productos ORDER BY precio DESC LIMIT 5 OFFSET 2;
```

## CASE

```
SELECT *,  
CASE  
    WHEN cantidad<5 THEN '10.0€'  
    WHEN cantidad>5 AND cantidad<10 THEN '3.0€'  
    ELSE 'Gratis'  
END AS CosteEnvio  
FROM pedidos;
```

```
SELECT nombre, apellido, telefono FROM clientes ORDER BY  
(CASE  
    WHEN apellido IS NULL THEN NOMBRE  
    ELSE apellido  
END) ;
```

### ▼ Variables

Una variable sirve para almacenar información cuyo valor puede variar a lo largo de la ejecución del programa. Toda variable tiene asociado un identificador y un tipo de dato y se almacena en memoria principal.

## Sintaxis

```
DECLARE Nombre Tipo [NOT NULL] [DEFAULT Valor];
```

# Asignar un valor a una variable

```
SET Nombre_variable = Valor;
```

## ▼ Procedimientos almacenados

## Qué son los Procedimientos almacenados

Es un conjunto de instrucciones que se usan para ejecutar determinada acción, pueden ser ejecutadas en cualquier momento según sea el caso y están almacenadas en la base de datos.

MySQL guarda en memoria cada vez que un procedimiento almacenado se ejecuta, esto con el fin de hacer muchísimo más rápido la ejecución.

Dentro de los procedimientos almacenados también se pueden usar sentencias de control como: **IF, CASE, LOOP**.

## Creación de un procedimientos almacenado

Un procedimiento almacenado se crea de la siguiente manera:

```
DELIMITER //  
  
CREATE PROCEDURE Nombre ([parametros])  
  
BEGIN  
  
    [sentencia sql];  
  
END //  
  
DELIMITER ;
```

La instrucción DELIMITER, se utiliza para cambiar el delimitador por defecto que es el punto y coma para que se tome como parte de la creación del

procedimiento almacenado, al terminar de crear el procedimiento se debe restablecer el delimitador al punto y coma.

## Ejecución de un procedimiento

Para realizar la ejecución de un procedimiento almacenado, se utilizará la instrucción **CALL**.

```
CALL Nombre([parametros]);
```

## Eliminar un procedimiento

Para eliminar por completo un procedimiento almacenado de la base de datos se utilizará la instrucción **DROP**.

```
DROP PROCEDURE [IF EXISTS] Nombre;
```

## Modificar un procedimiento

```
ALTER PROCEDURE Nombre [características];
```

Las características que se pueden modificar son las siguientes:

```
{CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA}  
SQL SECURITY {DEFINER | INVOKER}  
COMMENT 'comentarios'
```

## Mostrar los procedimientos

```
SHOW PROCEDURE STATUS WHERE DEFINER = 'root@localhost'\G;
```

## Sentencia IF

```
DELIMITER //  
  
CREATE PROCEDURE Nombre ([parametros])  
  
BEGIN  
  
IF condición THEN  
    [sentencia sql];  
ELSE  
    [sentencia sql];  
END IF;  
  
END //  
  
DELIMITER ;
```

## Sentencia While

```
DELIMITER //  
  
CREATE PROCEDURE Nombre ([parametros])  
  
BEGIN  
  
WHILE condición DO  
    [sentencia sql];  
END WHILE;  
  
END //  
  
DELIMITER ;
```

### ▼ Funciones

Una función es un subprograma que devuelve algún dato.

## Creación de una Función

```
DELIMITER //
```

```
CREATE FUNCTION Nombre ([Lista_Parámetros]) RETURNS Tipo  
BEGIN  
  
[DECLARE declaración1;  
DECLARE declaración2;...]  
  
Instrucciones  
  
END //  
  
DELIMITER ;
```

En el conjunto de **instrucciones** del cuerpo de la función se debe incluir una instrucción RETURN (expresión);o RETURN expresión; para devolver un valor.

Ejemplo:

```
DELIMITER //  
  
CREATE FUNCTION LeerDescriArti (cod char(5)) RETURNS varchar(30)  
BEGIN  
  
DECLARE descri varchar(30);  
  
SELECT DesArt INTO descri FROM Articulo WHERE CodArt=cod;  
  
RETURN descri;  
  
END //  
  
DELIMITER ;
```

## Ejecución de una función

```
SELECT Nombre();
```

Ejemplo:

```
SELECT LeerDescriArti ('A0043') Descripción;
```

```
+-----+
| Descripción      |
+-----+
| Bolígrafo azul |
+-----+
1 row in set (0.00 sec)
```

## Eliminar una Función

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

## Modificar una Función

```
ALTER FUNCTION Nombre [características];
```

Las características que se pueden modificar son las siguientes:

```
{CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA}
SQL SECURITY {DEFINER | INVOKER}
COMMENT 'comentarios'
```

## Funciones Predefinidas más útiles

- length (cadena)
- substring (cadena, posición, longitud)
- min(valores)
- max(valores)
- count(valores)
- sum(valores)
- avg(valores) → Devuelve el promedio

### ▼ Estructuras de control

Como en todos los lenguajes de programación, al escribir procedimientos y funciones en MySQL se pueden incluir diversas estructuras de control para controlar el flujo de ejecución de los programas.

## Estructura alternativa

La estructura alternativa se construye mediante la sentencia IF y la sentencia CASE y permite decidir qué secuencia de código se va a ejecutar a continuación en función del cumplimiento o no de una determinada condición.

### IF

```
If condición then
    instrucción1;
    ...
    instrucciónn;
end if;
```

```
If condición then
    instrucción1;
    ...
    instrucciónn;
else
    instrucciónn+1;
    ...
    instrucciónn+m;
end if;
```

```
if condición1 then
    instrucciones1;
elseif condición2 then
    instrucciones2;
elseif condición3 then
    instrucciones3;
...
[else
    instruccionesn+1;]
end if;
```

## CASE

```
case variable
when valor1 then
    instrucciones1;
when valor2 then
    instrucciones2;
when valor3 then
    instrucciones3;
...
[else
    instruccionesn+1;]
end case;
```

```
case
when condición1 then
    instrucciones1;
when condición2 then
    instrucciones2;
when condición3 then
    instrucciones3;
...
[else
    instruccionesn+1;]
end case;
```

## Estructura repetitiva

Una estructura repetitiva permite repetir una secuencia de instrucciones un número determinado de veces

## WHILE

```
while condición do
    instrucción1;
    instrucción2;
    ...
    instrucciónn;
end while;
```

## REPEAT

Se trata de una estructura repetitiva de 1 a n, lo que quiere decir que el conjunto de instrucciones que forman parte del bucle como mínimo se va a ejecutar una vez. Cómo el DO WHILE vamos, pero **ATENTO** a diferencia del WHILE y del DO WHILE la instrucción en REPEAT se repite **HASTA que se cumpla la condición y NO mientras.**

```
repeat
    instrucción1;
    instrucción2;
    ...
    instrucciónn;
until condicion
end repeat;
```

## ▼ Parámetros

El paso de información a los subprogramas se realiza por medio de los parámetros. Hemos de distinguir entre parámetros formales y parámetros reales o actuales:

- Los **parámetros formales** son los que aparecen declarados en la cabecera del procedimiento o función. Por cada uno de ellos recordemos que se debe especificar el nombre y el tipo de dato asociado.
- Los **parámetros actuales o reales** son los que aparecen en la llamada al procedimiento o función.

Los tipos de los parámetros formales y actuales deben ser compatibles.

Los parámetros que se pasan a los procedimientos y funciones pueden ser de entrada, de salida o de bien de entrada/salida

- Los parámetros de entrada (**IN**) hacen referencia a los datos que se pasan a un subprograma para que este efectúe operaciones con ellos. En el subprograma llamado no se le puede asignar ningún valor al parámetro formal, sino que solamente se puede utilizar el valor que tiene dicho parámetro.
- Los parámetros de salida (**OUT**) se usan para devolver datos del subprograma llamado al que realizó la llamada. El parámetro actual debe

ser obligatoriamente una variable a la que se asignará valor en el subprograma llamado para devolverlo al programa llamante.

- Los parámetros de entrada/salida (**IN OUT**) sirven para pasar un dato del programa llamante al subprograma llamado, modificar dicho valor en el subprograma y devolver el valor modificado al programa llamante.

El tipo de parámetro se especifica delante del nombre del mismo. De esta manera una lista de parámetros tendrá la siguiente forma:

```
([tipo1] Nombre tipo_dato1, [tipo2] Nombre tipo_dato2, ...)
```

El tipo por defecto es de entrada, por lo que no poner nada equivale a escribir **IN** como tipo de parámetro.

### ▼ Excepciones

Se considera una excepción cualquier error que pueda ocurrir a lo largo de la ejecución de un programa. Si una excepción no es tratada, provocará la terminación anormal del programa en el que se produzca. Sin embargo, si lo que se desea es que se informe del error o excepción al usuario y que se pueda continuar con la ejecución del programa, entonces será necesario tratar la excepción correspondiente mediante un manejador de errores o handler.

## Sintaxis

```
DECLARE handler_action HANDLER
    FOR condition_value [,condition_value] ...
        statement

    handler_action: {
        CONTINUE
        | EXIT
        | UNDO
    }

    condition_value: {
        mysql_error_code
        | SQLSTATE ['VALUE'] sqlstate_value
        | condition_name
        | SQLWARNING
```

```
    | NOT FOUND
    | SQLEXCEPTION
}
```

### Ejemplo1:

```
declare continue handler for 1062 set duplicado = 1;
```

### Ejemplo2:

```
DELIMITER //

CREATE PROCEDURE MostrarDescri (codar char(5))

BEGIN

    DECLARE encontrado bool DEFAULT 1;
    DECLARE descri varchar(30);

    DECLARE continue HANDLER FOR sqlstate '02000' SET encontrado = 0;

    SELECT DesArt INTO descri FROM Articulo WHERE codart = codar;
    IF encontrado = 0 THEN
        SELECT concat ('No existe ningún artículo con el código ', codar) ERROR;
    ELSE
        SELECT descri Descripción;
    END IF;

END //

DELIMITER ;
```

## handler\_action

después de la palabra DECLARE se debe indicar si se trata de un manejador de tipo CONTINUE o EXIT:

- Con un manejador del tipo **CONTINUE**, tras el levantamiento de la excepción, la ejecución del programa continúa en la siguiente instrucción.
- Con un manejador del tipo **EXIT**, después de producirse la excepción, la ejecución del bloque en el que se ha producido la excepción finaliza,

pasando la ejecución al bloque externo dentro del mismo programa, o bien, si es el bloque principal, se devuelve la ejecución al programa externo que invocó al procedimiento o función.

## condition\_value

condición por la que se produce la excepción, que puede tomar varios formatos, entre ellos:

- SQLSTATE 'valor\_estado\_SQL': Es un código alfanumérico de 5 caracteres que lleva asociado cada posible error en MySQL.
- código\_error\_MySQL: Es un código numérico de 4 cifras que lleva asociado cada posible error en MySQL.
- NOT FOUND: Hace referencia a estados SQL que comienzan por '02'.

Se muestran en la siguiente tabla algunas de las excepciones que se pueden producir en MySQL, indicando por cada una de ellas su código de error, su valor de estado y una descripción. Están resaltadas en amarillo las excepciones más relevantes.

<b>Código de error</b>	<b>Valor de estado</b>	<b>Descripción</b>
1005	HY000	No se puede crear la tabla indicada
1006	HY000	No se puede crear la base de datos indicada
1007	HY000	No se puede crear la base de datos indicada. La base de datos ya existe.
1008	HY000	No se puede borrar la base de datos indicada. La base de datos no existe.
1009	HY000	Error eliminando base de datos.
1040	08004	Demasiadas conexiones
1044	42000	Acceso denegado para el usuario indicado sobre la base de datos indicada.
1045	28000	Acceso denegado para el usuario indicado con la contraseña indicada.
1046	3D000	No seleccionada base de datos.
1047	08S01	Comando desconocido
1048	23000	La columna indicada no puede tomar valor nulo
1049	42000	Base de datos desconocida
1050	42S01	La tabla indicada ya existe
1051	42S02	Tabla desconocida
1052	23000	Columna ambigua

1054	42S22	Columna desconocida en la tabla indicada
1055	42000	La columna indicada no está en GROUP BY
1056	42000	No se puede agrupar sobre el campo indicado
1057	42000	La misma sentencia contiene funciones de resumen y atributos
1062	23000	Entrada duplicada para el atributo clave indicado
1068	42000	Definida clave primaria duplicada
1090	42000	No se pueden eliminar todas las columnas con ALTER TABLE; use DROP TABLE
1102	42000	Nombre de base datos incorrecta
1103	42000	Nombre de tabla incorrecta
1106	42000	Nombre de procedimiento desconocido
1107	42000	Incorrecto número de parámetros para el procedimiento indicado.
1108	42000	Parámetros incorrectos para el procedimiento indicado.
1215	HY000	No se puede añadir una restricción de clave ajena.
1216	HY000	No se puede añadir o eliminar una fila hija porque falla una restricción de clave ajena
1217	HY000	No se puede añadir o eliminar una fila padre porque falla una restricción de clave ajena
1231	42000	A la variable indicada no se le puede asignar el valor indicado
1242	21000	La subconsulta devuelve más de una fila.
1280	42000	Nombre de índice indicado incorrecto.
1329	02000	Cero filas (ningún dato) recibido, seleccionado o procesado
1348	HY000	La columna indicada no es modificable
1451	23000	No se puede eliminar o modificar una fila padre porque falla una restricción de clave ajena
1452	23000	No se puede insertar o modificar una fila hija porque falla una restricción de clave ajena

## ▼ Cursos

Hasta ahora todas las consultas que hemos creado en nuestros procedimientos y funciones nos devolvían una sola fila y esto no era casualidad. Esto se debe a que hemos empleado lo que se llaman cursos implícitos. Y es que estos cursos permiten solo manejar una fila. De hecho, si

en los programas que hemos creado hasta ahora una consulta nos devolviera varias filas, se produciría una excepción que, al no ser tratada, provocaría la terminación anormal del programa.

Por todo esto, si queremos ejecutar consultas que devuelvan varias filas, debemos usar cursores explícitos, que vamos a tratar a continuación.

## Declaración/Creación de un cursor

Para poder utilizar un cursor explícito, al igual que para poder utilizar una variable, es necesario declararlo. Un cursor se declara de acuerdo con la siguiente sintaxis:

```
DECLARE Nombre CURSOR FOR sentencia_SELECT;
```

## Uso de un cursor

Por su parte, para utilizar un cursor, será necesario, en primer lugar, abrirlo. Para ello, se utilizará la siguiente instrucción:

```
OPEN Nombre;
```

Al abrir un cursor, se ejecuta la sentencia select que se asignó al mismo en la declaración y se almacenan los resultados de la ejecución en estructuras internas de memoria.

Para acceder a la información almacenada en el cursor, es decir, a los datos resultantes de la ejecución de la sentencia select, es necesario usar la instrucción siguiente:

```
FETCH Nombre INTO [variable1, variable2, ...];
```

Habrá que escribir tantas variables separadas por comas como elementos aparezcan en la cláusula select. Esta/s variable/s tendrá/n que estar previamente declarada/s.

La instrucción fetch recupera una de las filas y pasa automáticamente a la siguiente fila de las resultantes de la ejecución de la sentencia select. Si al ejecutar una orden fetch, esta no devuelve datos, o lo que es lo mismo, si una vez leídas todas las filas del cursor, se pretende leer otra fila, se produce el error o excepción NOT FOUND (no encontrado). Si este error o excepción no es tratado, se producirá la terminación anormal del subprograma ejecutado.

Una vez empleado el cursor, es necesario cerrarlo con la siguiente instrucción:

```
CLOSE Nombre;
```

Excepción para evitar el Not Found en el fetch:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET NombreVariable = valor;
```

Ejemplo:

```
DELIMITER //

Create procedure VerPedido()
Begin
Declare refe char(5);
Declare fecha date;
Declare fin bool default 0;

Declare c cursor for select refped, fecped from pedido;

Declare continue handler for not found set fin = 1;

Open c;
Fetch c into refe, fecha;
while fin = 0 do
    Select concat ('Referencia: ', refe, ' Fecha: ', fecha) "Datos pedidos";
    Fetch c into refe, fecha;
end while;
Close c;
End //

DELIMITER ;
```

## ▼ Triggers

Un trigger es un tipo especial de rutina almacenada asociada a una tabla que se ejecuta o dispara automáticamente cuando ocurre una inserción (INSERT), borrado (DELETE) o modificación (UPDATE) sobre una tabla. Los disparadores se pueden emplear para diferentes propósitos, entre los que se encuentran:

- Implementar restricciones complejas de seguridad o de integridad.
- Impedir transacciones erróneas.
- Generar automáticamente valores derivados.
- Auditarse actualizaciones, incluso enviando alertas.

## Creación de un Trigger

```
CREATE TRIGGER nombre_disparador momento_disparo evento_disparo
ON nombre_tabla FOR EACH ROW
[orden_disparador]
sentencia_disparador
```

El momento\_disparo puede ser BEFORE (antes) o AFTER (después).

El evento\_disparo puede ser r INSERT, UPDATE o DELETE.

Al final, se indicará la sentencia del disparador, esto es, la sentencia que queremos que se ejecute cuando se active el disparador. Si se desean ejecutar varias sentencias, deben colocarse entre las palabras BEGIN y END, que permiten construir sentencias complejas.

A las columnas de la tabla asociada al disparador nos podemos referir con los alias OLD y NEW. Con OLD.nombre\_atributo nos referimos al valor de un atributo de una fila existente antes de ser borrada o modificada. Con NEW.nombre\_atributo nos referimos al valor de un atributo en una nueva fila que va a ser insertada o después de ser modificada una fila existente.

## Eliminar un Trigger

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

## Mostrar Triggers

```
SHOW TRIGGERS [[FROM | IN] nombre_BD];
```