

BASES DE DATOS

Desarrollo de Aplicaciones Web

GESTIÓN DE BASES DE DATOS

Administración de Sistemas Informáticos en Red

PROGRAMACIÓN DE BASES DE DATOS en MySQL

Apuntes

Luis Dorado Garcés
Basado en el trabajo de Alba Tortosa López

Contenido

1	INTRODUCCIÓN	
1.1	¿Qué son las rutinas MySQL	
1.2	Ventajas de las rutinas MySQL	
1.3	Desventajas de las rutinas MySQL	
2	PROGRAMACIÓN DE BBDD EN MYSQL I	
2.1	Procedimientos.....	
2.1.1	Borrado de procedimientos	
2.1.2	Creación y ejecución.....	
2.1.3	Ejemplo sin parámetros.....	
2.1.4	El delimitador: DELIMITER	
2.1.5	Parámetros de entrada IN	
2.1.6	Ejemplo con parámetros	
2.1.7	Ampliación: Parámetros de salida OUT	
2.2	Variables.....	
2.2.1	Introducción	
2.2.2	Tipos de variables.....	
2.2.3	Sentencia DECLARE	
2.2.4	Sentencia SET.....	
2.2.5	Sentencia SELECT ... INTO	
2.3	Mostrar por pantalla varias líneas de texto.....	
3	PROGRAMACIÓN DE BBDD EN MYSQL II	
3.1	Estructuras de control condicional IF.....	
3.1.1	Estructura IF.....	
3.1.2	Estructura IF ... ELSE	
3.1.3	Estructura IF ... ELSEIF ... ELSE.....	
3.1.4	Otras estructuras IF	
3.1.5	Estructuras IF EXISTS (consulta) y NOT IN (consulta) THEN.....	
3.2	Estructuras de control condicional CASE.....	
3.2.1	Comprobar la igualdad de una expresión con sucesivos valores.....	
3.2.2	Comprobar sucesivas condiciones.....	
3.3	Estructuras de control iterativas (bucles).....	
3.3.1	Bucle WHILE.....	
3.3.2	Ampliación: Otras estructuras de control iterativas	
3.4	Funciones	
3.4.1	Permisos para funciones (Error DETERMINISTIC)	
3.4.2	Creación y ejecución.....	
3.4.3	Ejemplos.....	
3.4.4	Borrado de funciones	
3.5	Cursores	
3.5.1	Declarar un cursor.....	
3.5.2	Abrir un cursor (OPEN)	
3.5.3	Leer un cursor (FETCH)	
3.5.4	Cerrar un cursor	
3.5.5	Recorrer un cursor	
3.5.6	Ejemplo 1: recorrido simple de un cursor	
3.5.7	Ejemplo 2: recorrido simple de un cursor	
3.5.8	Ejemplo 3: recorrido anidado de cursors	
4	PROGRAMACIÓN DE BBDD EN MYSQL III	
4.1	Control de errores.....	
4.1.1	Concepto de error.....	
4.1.2	Información de los errores.....	
4.1.3	Los manejadores de errores	
4.1.4	Tipos de declaración de manejadores	
4.1.5	Ejemplo de declaración de un manejador	
4.1.6	Obtención de información del error: SQLWARNING y SQLEXCEPTION	
4.1.7	Ejemplo de obtención de información del error por parte del manejador	

4.1.8 Lanzamiento de excepciones	
4.1.9 Ejemplo de lanzamiento y captura de excepciones de usuario	
4.1.10 Ampliación: Declaración de excepciones.....	
4.2 Transacciones I	
4.2.1 Introducción	
4.2.2 Propiedades ACID.....	
4.2.3 START TRANSACTION	
4.2.4 COMMIT	
4.2.5 ROLLBACK	
4.2.6 Estado de los datos durante la transacción.....	
4.2.7 Cómo revertir la instrucción/transacción que genera un error	
4.3 Transacciones II	
4.3.1 SAVEPOINT, ROLLBACK TO SAVEPOINT y RELEASE SAVEPOINT.....	
4.3.2 Acceso concurrente a los datos.....	
4.3.3 Niveles de aislamiento.....	
4.3.4 Políticas de bloqueo	
4.4 Disparadores.....	
4.4.1 Sintaxis básica de creación de un disparador	
4.4.2 Concepto.....	
4.4.3 Características y usos	
4.4.4 Acceso a los datos afectados por la instrucción que lanzó el trigger	
4.4.5 Ejemplo guiado	
4.4.6 Limitaciones.....	
4.4.7 Cancelar instrucciones o revertir transacciones.....	
4.5 Ampliación: Transacciones	
4.5.1 Desactivar AUTOCOMMIT para uso obligado transacciones.....	

1 INTRODUCCIÓN

1.1 ¿Qué son las rutinas MySQL

En esta unidad veremos cómo crear rutinas en MySQL.

Las rutinas son programas que se almacenan físicamente en una tabla dentro del sistema gestor de bases de datos. Estos programas están hechos con un lenguaje propio de cada Gestor de BD y están compilados, por lo que la velocidad de ejecución será muy rápida. Un programa es un conjunto de comandos SQL que pueden almacenarse en el servidor de forma que no es necesario relanzar los comandos individuales, sino que basta con referirse a su nombre para ejecutarlos en conjunto.

Las rutinas pueden ser de dos tipos: procedimientos o funciones. En esta unidad veremos sólo los procedimientos.

1.2 Ventajas de las rutinas MySQL

Seguridad: Cuando llamamos a una rutina, esta deberá realizar todas las comprobaciones pertinentes de seguridad y seleccionará la información lo más precisamente posible, para enviar de vuelta la información justa y necesaria y que por la red corra el mínimo de información, consiguiendo así un aumento del rendimiento de la red considerable.

Rendimiento: el SGBD, en este caso MySQL, es capaz de trabajar más rápido con los datos que cualquier otro lenguaje del lado del servidor, y llevará a cabo las tareas con más eficiencia. Solo realizamos una conexión al servidor y este ya es capaz de realizar todas las comprobaciones sin tener que volver a establecer una conexión.

Reutilización: la rutina podrá ser invocada desde cualquier parte del programa, y no tendremos que volver a preparar la consulta a la BD cada vez que queramos obtener unos datos.

1.3 Desventajas de las rutinas MySQL

El programa se guarda en la BD, por lo tanto, si se corrompe y perdemos la información también perderemos nuestras rutinas. Esto es fácilmente subsanable llevando a cabo una buena política de copias de seguridad de la BD.

2 PROGRAMACIÓN DE BBDD EN MYSQL I

2.1 Procedimientos

Los procedimientos almacenados se crean con el comando **CREATE PROCEDURE**. Un procedimiento se invoca usando un comando **CALL**.

Las rutinas se asocian con una base de datos. Esto tiene varias implicaciones:

- Cuando se invoca la rutina, se realiza implícitamente **USE db_name** (y se deshace cuando acaba la rutina). Los comandos **USE** dentro de rutinas no se permiten.
- Cuando se borra una base de datos, todos los procedimientos almacenados asociados con ella también se borran.

2.1.1 Borrado de procedimientos

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

Este comando se usa para borrar un procedimiento almacenado. La cláusula **IF EXISTS** evita que ocurra un error si el procedimiento no existe.

Recomendamos que cada vez que crees un procedimiento primero lo borres si es que existe, para poder actualizar el código del procedimiento sin complicaciones.

2.1.2 Creación y ejecución

Creación

Estos comandos crean un procedimiento. Por defecto, el procedimiento se asocia con la base de datos actual.

```
CREATE PROCEDURE nombre_proc ([nombreParámetro tipo[,...]])  
BEGIN  
  Cuerpo del procedimiento  
END
```

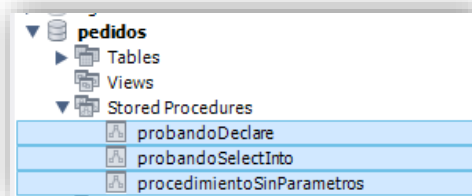
tipo: Cualquier tipo válido de MySQL

La lista de parámetros entre paréntesis debe estar siempre presente. Si no hay parámetros, se debe usar una lista de parámetros vacía **()**.

Cada parámetro es un valor que el usuario puede introducir al ejecutar el procedimiento. Al nivel que trabajaremos nosotros, estos valores deben ser estáticos (ej. 20, 'hola', '2018-12-03').

La sintaxis **BEGIN ... END** se utiliza para escribir sentencias compuestas que pueden aparecer en el interior de rutinas. Una sentencia compuesta puede contener múltiples sentencias, encerradas por las palabras **BEGIN** y **END**.

Además, podemos acceder a los procedimientos almacenados desde el Navegador de MySQL Workbench.



Ejecución

El comando **CALL** invoca un procedimiento definido previamente con **CREATE PROCEDURE**.

```
CALL nombre_proc ([valor1][, ...])
```

2.1.3 Ejemplo sin parámetros

Ejemplo de procedimiento sin parámetros:

```
USE pedidos;

DROP PROCEDURE IF EXISTS procedimientoSinParametros;
DELIMITER $$
CREATE PROCEDURE procedimientoSinParametros()
BEGIN
    SELECT COUNT(*) FROM clientes;
END
$$
DELIMITER ;

CALL procedimientoSinParametros();

-- Creamos e invocamos a un procedimiento que muestra la
```

```
-- cantidad de clientes
```

2.1.4 El delimitador: DELIMITER

¿Por qué es importante el uso del DELIMITER? Por defecto MySQL usa como DELIMITER el punto y coma (;), es decir, cada vez que encuentre punto y coma (;) ejecuta hasta ahí. Debido a que los procedimientos y funciones son varias líneas de códigos y algunas de ellas terminan con este delimitador, se ejecutaría solo hasta ahí, lo que ocasionaría un error. Es por esto que se hace necesario indicarle a MySQL que utilice otro DELIMITER que puede ser cualquier carácter que no se utilice en la definición de la rutina. Para el ejemplo usamos \$\$ y al finalizar la creación del procedimiento o función volvemos a cambiarlo por ';'.

2.1.5 Parámetros de entrada IN

Vamos a obtener los productos que tienen un determinado estado. Para crear el procedimiento, ejecuta estas sentencias SQL:

```
DROPPROCEDURE obtenerProductosPorEstado;
DELIMITER $$
CREATEPROCEDURE obtenerProductosPorEstado (IN nombre_estado
VARCHAR(255))
BEGIN
SELECT*
FROM productos
WHERE estado = nombre_estado;
END$$
DELIMITER

-- Creamos e invocamos a un procedimiento que muestra los
-- productos del estado recibido por el parámetro de entrada
```

El nombre del estado está contenido en el parámetro **nombre_estado** que hemos definido como **IN**. Suponiendo que quieras obtener los productos con estado *disponible*, tendrías que invocar al procedimiento de este modo:

```
CALL obtenerProductosPorEstado('disponible');
```

Este será el resultado:

id	nombre	estado	precio
1	Producto A	disponible	8.00

2	Producto B	disponible	1.50
---	------------	------------	------

OJO: Los parámetros de entrada son los parámetros por defecto, no es necesario poner **IN**

2.1.6 Ejemplo con parámetros

Vamos a aumentar el salario de todos los empleados en la cantidad que se especifique en el parámetro.

```
USE empleados;

DELIMITER $$
CREATE PROCEDURE procedimientoConParametro(aumento int)
BEGIN
UPDATE empleados SET salario = salario + aumento;
END
$$
DELIMITER ;

CALL procedimientoConParametro(100);
```

2.1.7 Ampliación: Parámetros de salida OUT

Vamos a obtener el número de productos según su estado. Para crear el procedimiento, ejecuta estas sentencias SQL:

```
DELIMITER $$
CREATEPROCEDURE contarProductosPorEstado(
IN nombre_estado VARCHAR(25),
OUT numero INT)
BEGIN
SELECTcount(id)
INTO numero
FROM productos
WHERE estado = nombre_estado;
END$$
DELIMITER
```

Al igual que antes, pasamos el estado como **nombre_estado**, definido como **IN**. También definimos **numero** como parámetro **OUT**. Suponiendo que quieras obtener el número de productos con estado *disponible*, debes llamar al procedimiento de este modo:

```
CALL contarProductosPorEstado('disponible', @numero);
SELECT @numero AS disponibles;
```

Este será el resultado:

disponibles
2

Para obtener el número de productos agotados, debemos invocar al procedimiento de este modo:

```
CALL contarProductosPorEstado('agotado', @numero);
SELECT @numero AS agotados;
```

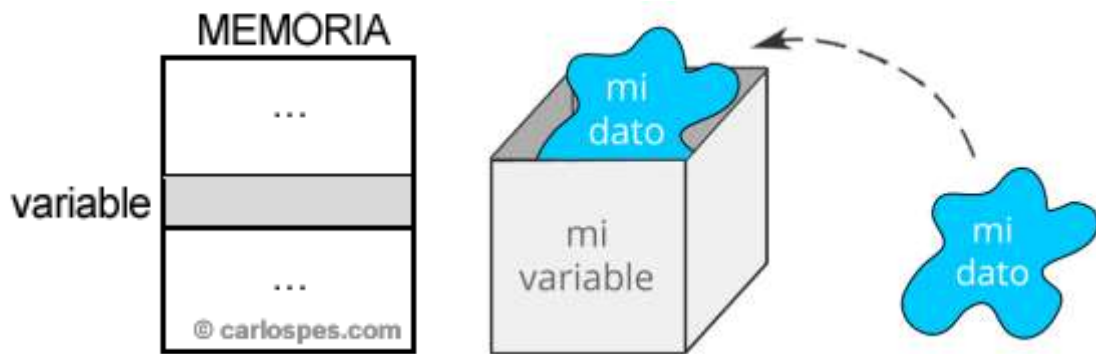
Este será el resultado:

agotados
1

2.2 Variables

2.2.1 Introducción

Se puede declarar y usar una variable dentro de una rutina. Las variables sirven para guardar datos temporalmente y utilizarlos en otras instrucciones dentro de la misma rutina.



Una **variable** es un espacio de memoria reservado por un programa al cual se le pone un nombre (**identificador**) y que puede contener diferentes datos (**valores**).

Estos valores van **cambiando a lo largo de la vida** de la variable, de ahí su nombre.

Igual que nosotros necesitamos post-it's para anotarnos citas o una agenda para anotar nuestros contactos, los **programas requieren alojar en memoria los datos con los que están trabajando**, así como los resultados de sus cálculos intermedios y finales.

2.2.2 Tipos de variables

Variables locales (SÍ LAS USAREMOS)

Las variables locales, a diferencia de las definidas por el usuario, no necesitan el prefijo `@` en sus nombres y deben ser declaradas antes de que puedan ser usadas. Para declarar una variable local, puedes usar la declaración `DECLARE` o usarla como un parámetro dentro de una declaración `CREATE PROCEDURE`.

Cuando se declara una variable local, opcionalmente, se le puede asignar un valor por defecto. Si no asignas ningún valor por defecto, la variable se inicializa con un valor NULL.

Recuerda que en los procedimientos y funciones (las veremos en adelante) **los parámetros se comportan como una variable local ya declarada** y cuyo valor ya se ha establecido al invocar el procedimiento función.

Cada variable vive dentro de un ámbito, delimitado por el bloque `BEGIN ... END` que contiene su declaración. Las usaremos como parámetros de los procedimientos almacenados

Variables definidas por el usuario (NO LAS USAREMOS)

El primer tipo son las variables definidas por el usuario, identificadas por un símbolo `@` usado como prefijo. En MySQL, se puede acceder a las variables definidas por el usuario sin necesidad de declararlas o inicializarlas previamente. Si lo hace, se asigna un valor NULL a la variable cuando se inicializa.

- ✓ Pueden usarse dentro o fuera de procedimientos.
- ✓ No se declaran previamente.
- ✓ No tienen un tipo de datos asignado.

Por ejemplo, si usas `SELECT` con una variable sin darle un valor, como en este caso:

```
-- La variable tiene valor nulo porque no se le ha asignado ninguno
SELECT @soyUnaVariable;
-- A la variable se le asigna el número 5
SET @soyUnaVariable = 5;
-- Muestro, por ejemplo, el doble del valor de la variable
SELECT @soyUnaVariable*2; -- Se muestra por pantalla un 10
```

Variables del sistema del servidor

Las variables del sistema se reconocen por su prefijo @@.

El servidor MySQL mantiene muchas variables del sistema configuradas a un valor predeterminado. Pueden ser de tipo **GLOBAL**, **SESSION** o **BOTH**.

Las variables globales afectan el funcionamiento general del servidor, mientras que las variables de sesión afectan su funcionamiento para las conexiones de clientes individuales.

Para ver los valores actuales utilizados por un servidor en ejecución, use la instrucción **SHOW VARIABLES** o **SELECT @@var_name**.

2.2.3 Sentencia DECLARE

Este comando se usa para declarar variables dentro de una rutina que más tarde se utilizarán para asignarles un valor.

```
DECLARE nombre_var tipo [DEFAULT valor];
```

Debéis recordad que TODAS las declaraciones de variables deben realizarse **al comienzo** del procedimiento/función/trigger.

Ejemplo

```

DELIMITER $$
CREATE PROCEDURE probandoDeclare()
BEGIN
  -- Cadena de texto con valor null
  DECLARE texto1 varchar(30);

  -- Cadena de texto con valor inicial 'hola'
  DECLARE texto2 varchar(30) DEFAULT 'hola';

  -- Número con valor null
  DECLARE numero1 integer;

  -- Número con valor inicial 1
  DECLARE numero2 integer DEFAULT 1;

  SELECT texto1, texto2, numero1, numero2;
END
$$
DELIMITER ;

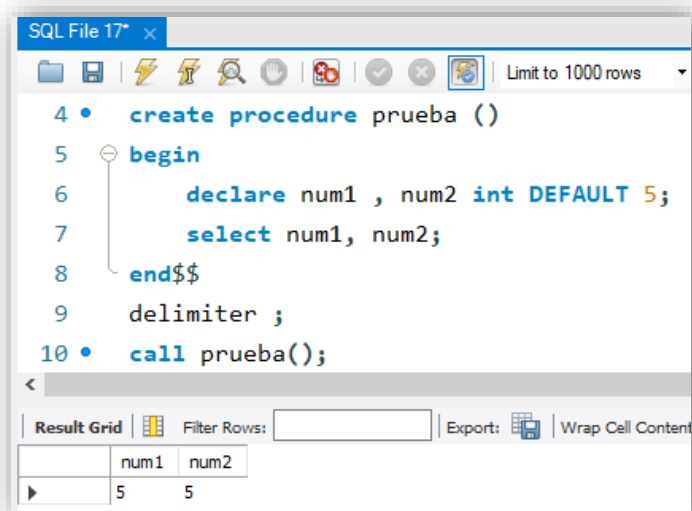
CALL probandoDeclare();

```

texto1	texto2	numero1	numero2
NULL	hola	NULL	1

Declaración múltiple

También es posible declarar varias variables en una sola sentencia DECLARE siempre que sean del mismo tipo y, en caso de tenerlo, el valor por defecto DEFAULT sea común a todas.



2.2.4 Sentencia SET

Este comando se usa para asignar un valor, una expresión o el resultado de una consulta a una variable.

```
SET nombre_var = valor o expresión;
SET nombre_var = (consulta);
```

- ✓ Ojo, **la consulta debe devolver un solo valor** (una sola fila de una sola columna). En otro caso podemos provocar los siguientes errores:

Error Code: 1242. Subquery returns more than 1 row

Error Code: 1241. Operand should contain 1 column(s)

- ✓ El SET variable = (**consulta**) no muestra datos por pantalla.

Ejemplos

```
DELIMITER $$
CREATE PROCEDURE probandoSet ()
BEGIN
    DECLARE texto1 varchar(30);
    DECLARE texto2 varchar(30) DEFAULT 'Hola';
    DECLARE numero1 integer;
    DECLARE numero2 integer DEFAULT 2;

    SET texto1 = 'Pepe';
    SET texto2 = CONCAT(texto2, ' ', texto1);
    SET numero1 = 3;
    SET numero2 = numero1 * numero2;

    SELECT texto1, texto2, numero1, numero2;
END
$$
DELIMITER ;

CALL probandoSet ();
```

texto1	texto2	numero1	numero2
Pepe	Hola Pepe	3	6

```

DELIMITER $$
CREATE PROCEDURE probandoSet2()
BEGIN
    DECLARE numJugadores int;
    DECLARE texto varchar(30);
    SET numJugadores = (SELECT COUNT(*) FROM jugadores);
    SET texto = CONCAT('El nº de jugadores es ', numJugadores);
    SELECT texto;
END
$$
DELIMITER ;

CALL probandoSet2();

```

texto
El nº de jugadores es 9

2.2.5 Sentencia SELECT ... INTO

Esta sintaxis **SELECT** almacena columnas seleccionadas directamente en variables. Las variables deben haber sido definidas previamente con la sentencia **DECLARE**.

```

SELECT columnal[,...] INTO nombre_var1[,...] table_expr FROM ...;

```

- ✓ La consulta de selección debe devolver **un solo registro** (fila) para que funcione.
- ✓ El número, tipo y orden de las variables debe ser el mismo que el de las columnas del SELECT.
- ✓ El **SELECT ... INTO** no muestra datos por pantalla.

Ejemplo

En este ejemplo vamos a mostrar todos los empleados cuyas ventas están entre la media y la mínima de las ventas todos los empleados.

```

USE empleados;

DELIMITER $$
CREATE PROCEDURE probandoSelectInto()
BEGIN
    -- Declaro dos variables del tipo número real
    DECLARE media float;
    DECLARE minimo float;
    -- Asigno a las variables la media y el mínimo de todas las ventas
    SELECT avg(ventas), min(ventas) INTO media, minimo FROM empleados;
    -- Muestro los empleados que cumplen la condición
    SELECT * FROM empleados WHERE ventas <media AND ventas >minimo;
END
$$
DELIMITER ;

CALL probandoSelectInto();

```

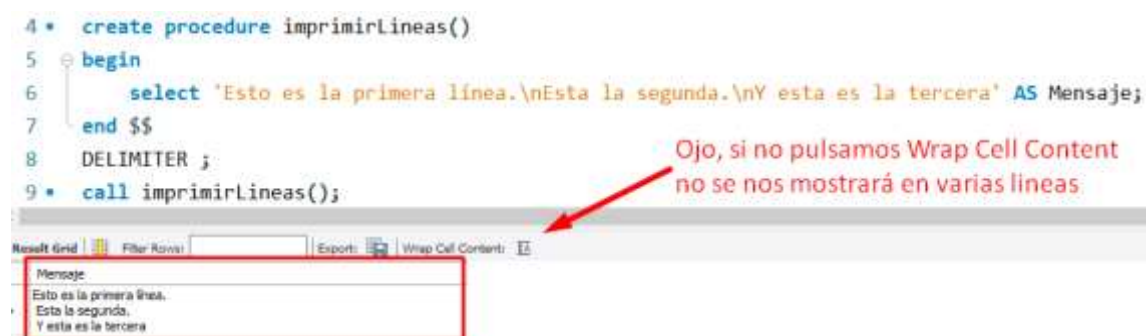
2.3 Mostrar por pantalla varias líneas de texto

A partir de ahora nos enfrentaremos a especificaciones de procedimientos, funciones o disparadores que impliquen mostrar mensajes con un formato específico, como por ejemplo **mensajes en varias líneas**.

Esto no es lo mismo que mostrar diferentes filas de una tabla como veremos a continuación.

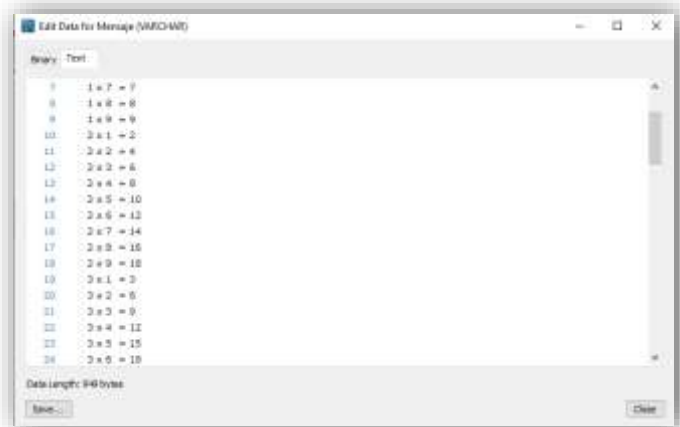
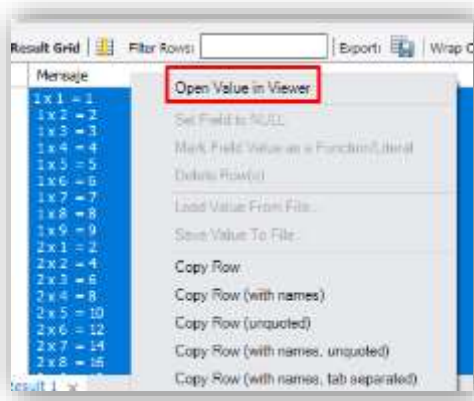
El principal carácter de escape que vamos a usar será '**\n**' que será el que nos proporcionará nuevas líneas para mostrar un mensaje con formatos interesantes.

Ejemplo



Si no se observan todos los datos

Si, en algún caso, no podéis ver todo el contenido de la celda por ocupar el texto muchas líneas, podéis hacer lo siguiente:



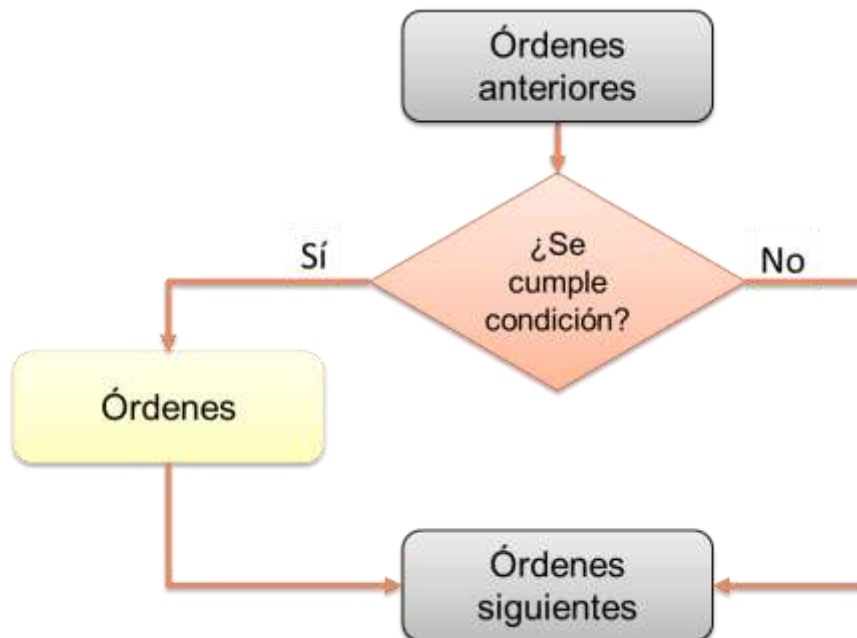
3 PROGRAMACIÓN DE BBDD EN MYSQL II

3.1 Estructuras de control condicional IF

```
IF condición THEN sentencias  
  
    [ELSEIF condicion THEN sentencias] ...  
  
    [ELSE sentencias]  
  
END IF;
```

IF implementa un constructor condicional básico. Si *condición* se evalúa a cierto, el listado de comandos SQL correspondiente se ejecuta. Si no coincide ninguna *condición* se ejecuta la lista de comandos de la cláusula **ELSE**.

3.1.1 Estructura IF

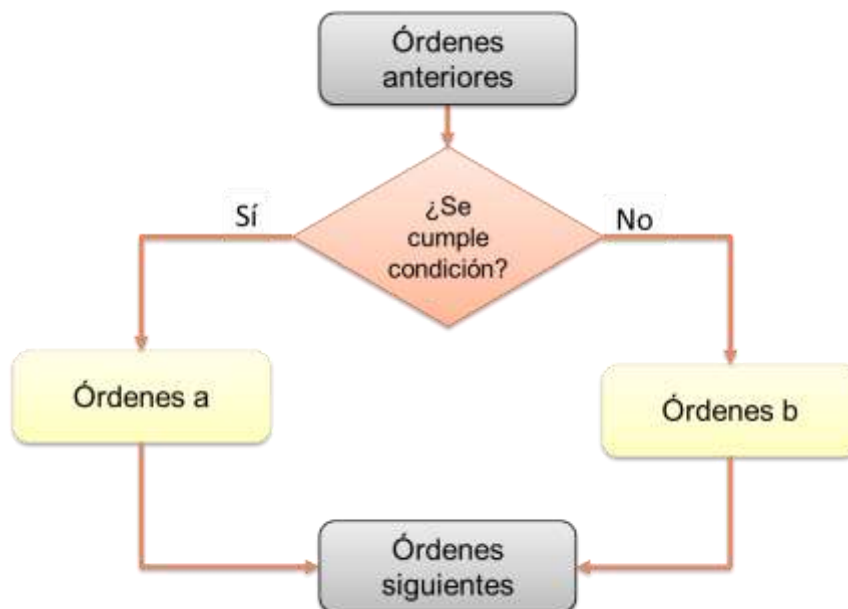


```
IF condición THEN  
    órdenes  
END IF;
```

Ejemplo

```
CREATEPROCEDURE esNegativo (num INTEGER)
BEGIN
DECLARE TEXTO VARCHAR(50);
IF num<0THEN
SET TEXTO =CONCAT('El número ',num,' es negativo');
ENDIF;
SELECT TEXTO AS Mensaje;
END
$$
DELIMITER ;
CALL esNegativo(0);
```

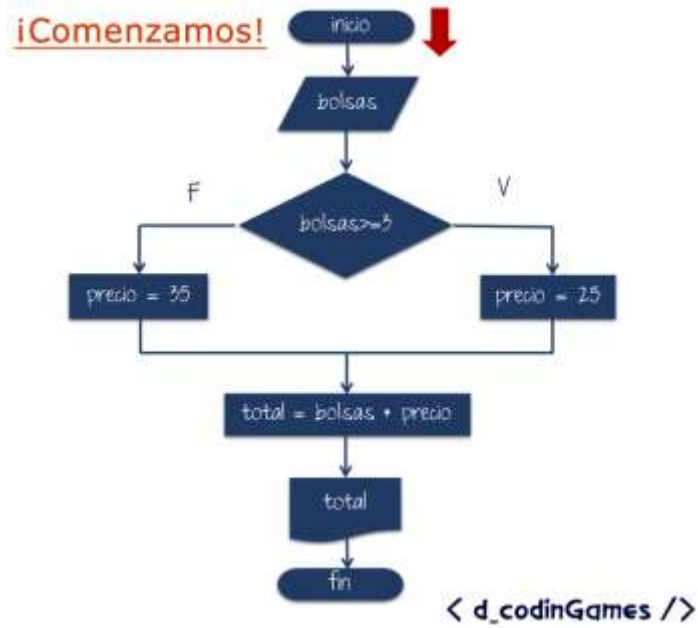
3.1.2 Estructura IF ... ELSE



```
IF condiciónTHEN
    órdenes a
ELSE
    órdenes b
END IF;
```

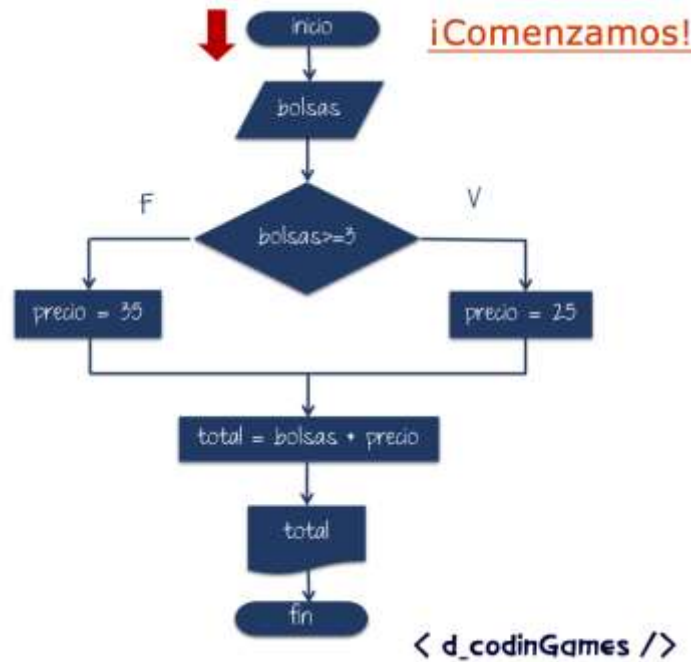
"En una tienda de chuches, la bolsa grande de papas fritas tiene un costo de \$35.00. Pero si el cliente se lleva más de tres bolsas, recibe un descuento de \$10.00 por cada bolsa. Elabore un algoritmo para calcular lo que un cliente debe pagar al comprar x bolsas grandes de papas fritas en dicha tienda"

Ejemplo con Bolsas = 4



Esta es una animación. Puede que no se vea correctamente según tu marca o versión de procesador de textos.

Ejemplo con Bolsas = 2



Esta es una animación. Puede que no se vea correctamente según tu marca o versión de procesador de textos.

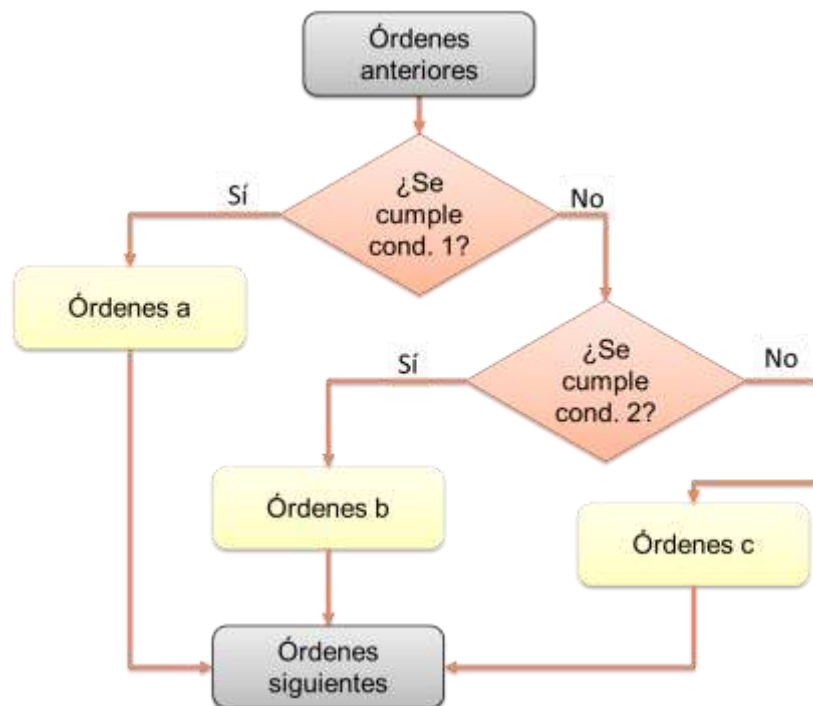
Ejemplo

```

CREATEPROCEDURE esPositivoOnegativo (num INTEGER)
BEGIN
  DECLARE TEXTO VARCHAR(50);
  IF num >= 0 THEN
    SET TEXTO = CONCAT('El número ', num, ' es positivo');
  ELSEIF num < 0 THEN
    SET TEXTO = CONCAT('El número ', num, ' es negativo');
  ENDIF;
  SELECT TEXTO AS Mensaje;
END
$$
DELIMITER ;
CALL esPositivoOnegativo (5);

```

3.1.3 Estructura IF ... ELSEIF ... ELSE



```

IF condición1 THEN
    órdenes a
ELSEIF condicion2 THEN
    órdenes b
ELSE
    órdenes c
END IF;

```

Ejemplo

```

DELIMITER $$
CREATE PROCEDURE clientesEmpleados()
BEGIN
    DECLARE numClientes, numEmpleados integer;

    SELECT COUNT(*) INTO numClientes FROM clientes;
    SELECT COUNT(*) INTO numEmpleados FROM empleados;

    IF numClientes > numEmpleados THEN
        SELECT 'Hay más clientes que empleados';
    ELSEIF numClientes < numEmpleados THEN
        SELECT 'Hay menos clientes que empleados';
    ELSE
        SELECT 'Hay el mismo número de clientes que de empleados';
    END IF;
END
$$
DELIMITER ;

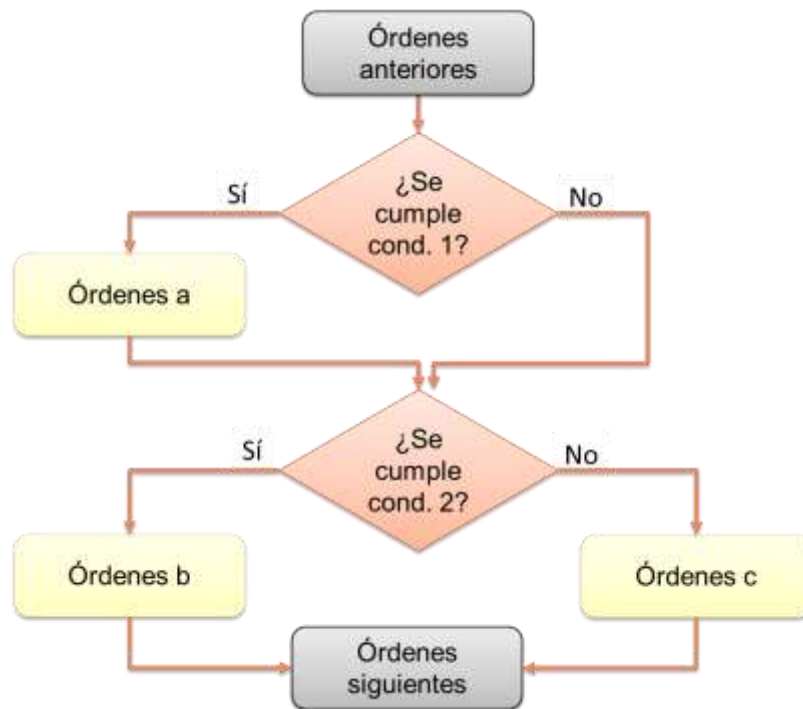
CALL clientesEmpleados();

```

3.1.4 Otras estructuras IF

IF ... END IF

IF ... END IF

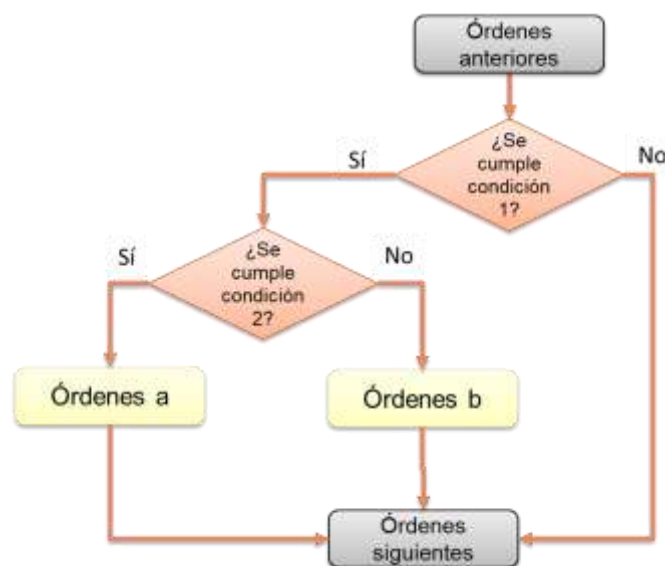


```

IF condición1 THEN órdenes a
END IF;

IF condición2 THEN órdenes b
  ELSE órdenes c
END IF;
  
```

IF ... IF ... END IF ... END IF



```

IF condición1 THEN
    IF condición2 THEN
        órdenes a
    ELSE
        órdenes b
    END IF;
END IF;

```

3.1.5 Estructuras IF EXISTS (consulta) y NOT IN (consulta) THEN

3.1.5.1 Aproximaciones menos elegantes

A veces nos podemos encontrar en la situación en la que queremos saber si una consulta devuelve o no resultados. Por ejemplo, si existe algún alumno con un ID.

Una primera aproximación podría ser usando COUNT(*) de la siguiente manera. En este caso hemos calculado cuántos alumnos tienen ese valor de clave primaria (en teoría será 1 o bien 0 si no existe).

```

delimiter $$
createprocedure existeAlumnoCount (codAlumno varchar(2))
BEGIN
declare numAlumnos int;
SET numAlumnos =(SELECTCOUNT(*)FROM alumnos
                  WHERE CODAL=codAlumno);
IFnumAlumnos =1THEN
SELECTCONCAT('Existe el alumno con codigo ', codAlumno);
ELSE
SELECTCONCAT('No Existe el alumno con codigo ', codAlumno);
ENDIF;
END $$
DELIMITER ;

```

Otra solución **mucho menos elegante** trataría de obtener en una variable el código del alumno que buscamos mediante la consulta. Si no existe, la consulta no devolverá nada y la variable permanecerá con valor nulo.


```

delimiter $$
createprocedure existeAlumnoNull(codAlumno varchar(2))
BEGIN
declare varCodAlumno varchar(2);
SET varCodAlumno =(SELECT CODAL FROM alumnos
                    WHERE CODAL=codAlumno);
IFvarCodAlumnois not nullTHEN
SELECTCONCAT('Existe el alumno con codigo ', codAlumno);
ELSE
SELECTCONCAT('No existe el alumno con codigo ', codAlumno);
ENDIF;
END $$
DELIMITER ;

```

3.1.5.2 Usando el IF EXISTS (consulta)

En este caso comprobamos si existe el alumno en cuestión con EXISTS de la misma forma que lo usamos en el WHERE o en el HAVING para subconsultas.

```

delimiter $$
createprocedure existeAlumnoExists(codAlumno varchar(2))
BEGIN
IFEXISTS(SELECT*FROM alumnos WHERE CODAL=codAlumno)THEN
SELECTCONCAT('Existe el alumno con codigo ', codAlumno);
ELSE
SELECTCONCAT('No Existe el alumno con codigo ', codAlumno);
ENDIF;
END $$
DELIMITER ;

```

3.1.5.3 Usando el NOT IN (consulta)

En este caso comprobamos si el código de alumno recibido por parámetro está entre los códigos de alumnos de toda la tabla.

```

delimiter $$
createprocedure existeAlumnoIn (codAlumno varchar(2))
BEGIN
IF codAlumno IN(SELECT CODAL FROM alumnos)THEN
SELECTCONCAT('Existe el alumno con codigo ', codAlumno);
ELSE
SELECTCONCAT('No Existe el alumno con codigo ', codAlumno);
ENDIF;
END $$
DELIMITER ;

```

3.2 Estructuras de control condicional CASE

Similar a la sentencia condicional IF anterior. Esta última se suele utilizar cuando el número de expresiones o condiciones a evaluar es elevado y por tanto la lectura del código se hace más fácil y agradable. **Pero su uso suele limitarse a los casos en los que se ejecutarán muy pocas instrucciones para cada caso.**

Existen 2 formas posibles si lo que se evalúa es la igualdad a una expresión o una condición.

3.2.1 Comprobar la igualdad de una expresión con sucesivos valores

En este caso nos interesa comparar un parámetro, variable o expresión con una serie de valores.

Sintaxis

```
CASE expresión
WHEN valor1 THEN instrucciones
[WHEN valor2 THEN instrucciones]
...
[ELSE instrucciones]
ENDCASE;
```

Ejemplo

```
delimiter $$
createprocedure tipoMedalla(puesto int)
begin
declare medalla varchar(10);
case puesto
when1then
set medalla='oro';
when2then
set medalla='plata';
when3then
set medalla='bronce';
else
set medalla='ERROR';
endcase;
SELECTCONCAT('Al puesto ',puesto,' le corresponde la
medalla de ',medalla)AS Mensaje;
end $$
delimiter;
```

3.2.2 Comprobar sucesivas condiciones

En este caso nos interesa establecer una condición menos restringida. Puede ser de igualdad o de cualquier otro tipo.

Sintaxis

```
CASE
WHEN condición1 THEN instrucciones
[WHEN condición2 THEN instrucciones]
...
[ELSE instrucciones]
ENDCASE;
```

Ejemplo

```
createprocedure cantidadDigitos(numero int)
begin
declare cantidad int;
case
when numero<10then
set cantidad=1;
when numero>=10and numero<100then
set cantidad=2;
when numero>=100and numero<1000then
set cantidad=3;
elseset cantidad=-1;
endcase;
IF(cantidad=-1) THEN
SELECT'El número tiene más de tres dígitos'AS Mensaje;
ELSE
SELECTCONCAT('El número ',numero,' tiene ',cantidad,'
                                                    dígitos')AS Mensaje;
ENDIF;
end $$
delimiter ;
```

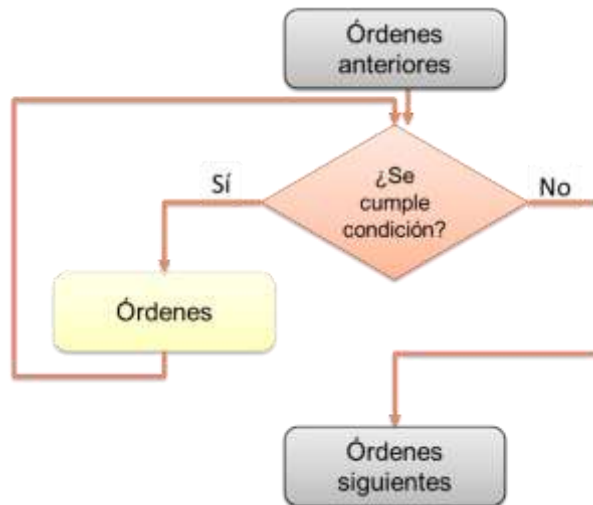
Importante: Aunque no es obligatorio os recomiendo que SIEMPRE uséis ELSE en los CASE (aunque solo sea para mostrar un error por valor inválido). Esto se debe a que si evaluamos un valor que no tiene correspondencia con ningún CASE obtendremos el siguiente error:

Error Code: 1339. Case not found for CASE statement

3.3 Estructuras de control iterativas(bucles)

Las estructuras programáticas iterativas son conocidas popularmente como “**bucles**”. Se trata de una secuencia de instrucciones de código que se

ejecuta repetidas veces, hasta que la condición asignada a dicho bucle deja de cumplirse.



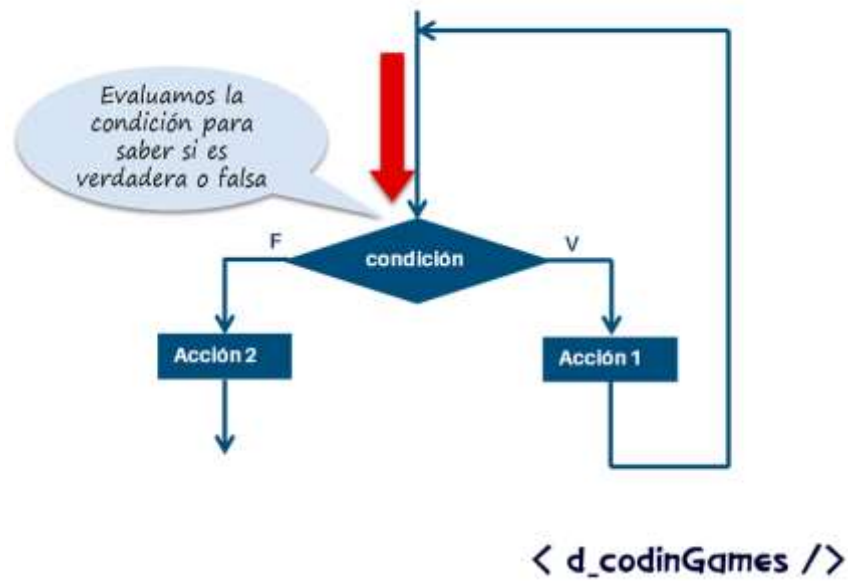
Cada una de las repeticiones del cuerpo de instrucciones del bucle (órdenes en el diagrama) se llama **iteración o ciclo**.

Después de un número determinado de veces la condición dejará de cumplirse y no repetirá la siguiente iteración. Ten en cuenta que el bucle solo ejecuta las instrucciones si se cumple la condición, teniendo esto en cuenta si la instrucción no se cumple cuando nuestro script llega al bucle ni siquiera hará una sola iteración. Popularmente diremos que "no ha llegado a entrar al bucle".



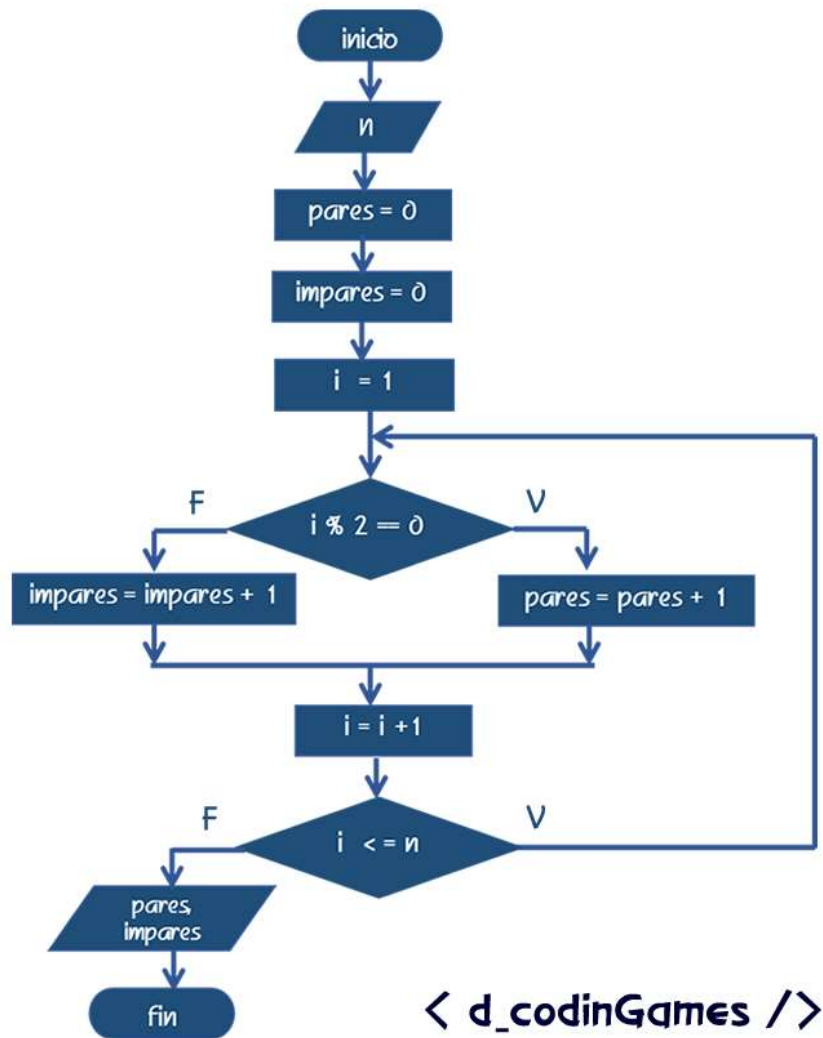
< d_codinGames />

Esta es una animación. Puede que no se vea correctamente según tu marca o versión de procesador de textos.



Esta es una animación. Puede que no se vea correctamente según tu marca o versión de procesador de textos.

"Elabora un programa que cuente el número de pares e impares que existen en el rango de 1 a n."



3.3.1 Bucle WHILE

Este tipo de bucles se conocen popularmente como bucles WHILE. Veamos cómo funciona.

Sintaxis

```

WHILE condición DO
  Instrucciones a repetir
  ...
END WHILE
  
```

Significaría algo así como “Mientras (WHILE) se cumple la condición ejecuta repetidamente (DO) las siguientes instrucciones”.

Ejemplos

```

DELIMITER $$
createprocedure iterar()
BEGIN
DECLARE contador intDEFAULT1;
DECLARE limite intDEFAULT5;
WHILE contador <= limite DO
SELECTCONCAT('Esta es la iteración ',contador)AS Mensaje;
SET contador=contador+1;
ENDWHILE;
END $$
DELIMITER ;

```



Como podéis observar si ejecutamos varios SELECTs nos aparecen como pestañas de resultados individuales. Vamos a propiciar que aparezca como un solo resultado.

```

DELIMITER $$
createprocedure iterar (limite int)
BEGIN
DECLARE contador intDEFAULT1;
DECLARE texto VARCHAR(200)DEFAULT'';
WHILE contador <= limite DO
SET texto =CONCAT(texto,'Esta es la iteración ',contador,'\n');
SET contador=contador+1;
ENDWHILE;
SELECT texto AS Mensaje;
END $$
DELIMITER ;

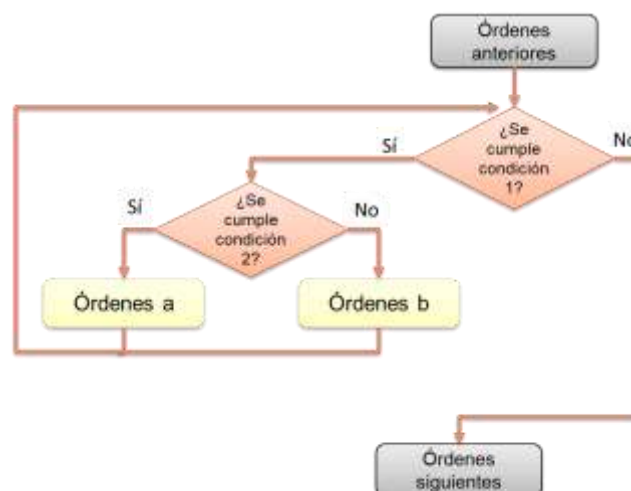
```

CALL iterar(5);

id	Filter Rows:
Mensaje	
Esta es la iteración 1	
Esta es la iteración 2	
Esta es la iteración 3	
Esta es la iteración 4	
Esta es la iteración 5	

Anidamiento de estructuras de control

Las distintas estructuras de control (IF, CASE, WHILE, etc.) pueden anidarse para poder crear programas tan complejos como se requieran.




```

WHILE (condición1) DO
IF (condición2) THEN
... órdenes a
ELSE
... órdenes b
ENDIF;
ENDWHILE;

```

3.3.2 Ampliación: Otras estructuras de control iterativas

3.3.2.1 Sentencia *LEAVE*

```

LEAVEetiqueta;

```

Este comando se usa para abandonar cualquier control de flujo etiquetado. Suele usarse con bucles.

3.3.2.2 Estructura de control iterativa *LOOP*

```

[etiqueta:] LOOP

sentencias

END LOOP[etiqueta];

```

LOOP implementa un constructor de bucle simple que permite la ejecución repetida de una lista de sentencias. El comando dentro del bucle se repite hasta que acaba el bucle, usualmente con un comando **LEAVE**.

Un comando **LOOP** puede etiquetarse.

Ejemplo

```

DELIMITER $$
CREATE PROCEDURE probandoLoop(numero integer)
BEGIN
    DECLARE contador integer DEFAULT 1;
    DELETE FROM test;
    bucle: LOOP
        IF contador > numero THEN
            LEAVE bucle;
        END IF;
        INSERT INTO test VALUES (contador);
        SET contador = contador + 1;
    END LOOP;
    SELECT * FROM test;
END
$$
DELIMITER ;
CALL probandoLoop(4);

```

3.4 Funciones

Las funciones se crean con el comando **CREATE FUNCTION**. Una función puede llamarse desde dentro de un comando como cualquier otra función (esto es, invocando el nombre de la función), y puede retornar un valor **escalar** (es decir, un solo dato).

Las funciones se asocian con una base de datos. Esto tiene varias implicaciones:

- Cuando se invoca la rutina, se realiza implícitamente **USE db_name** (y se deshace cuando acaba la rutina). Los comandos **USE** dentro de rutinas no se permiten.
- Cuando se borra una base de datos, todas las funciones asociadas con ella también se borran, al igual que ocurre con los procedimientos.

3.4.1 Permisos para funciones (Error DETERMINISTIC)

Para permitir la creación de funciones sin especificar modificadores **DETERMINISTIC**, **NO SQL** o **READS SQL DATA** incluiremos la línea:

```
SET GLOBAL log_bin_trust_function_creators =1;
```

al comienzo de los scripts.

También podéis incluir la línea:

```
log_bin_trust_function_creators= 1
```

al final del fichero **my.ini**¹ y reiniciaremos el motor.

3.4.2 Creación y ejecución

Los siguientes comandos crean una función. Por defecto, la función se asocia con la base de datos actual.

```
CREATE FUNCTION nombre_func ([parametro[,...]]) RETURNS tipoDato
BEGIN
    Cuerpo de la función (instrucciones)
    RETURN valor
END
parámetro: nombre del parámetro de entrada y su tipo (como en procedimientos)
tipoDato: Cualquier tipo válido de MySQL. Será el tipo del valor de salida
```

La lista de parámetros entre paréntesis debe estar siempre presente. Si no hay parámetros, se debe usar una lista de parámetros vacía **()**. Cada parámetro es un valor que el usuario puede introducir al ejecutar la función. Estos valores deben ser estáticos (ej. 20, 'hola', '2018-12-03').

La sintaxis **BEGIN ... END** se utiliza para escribir sentencias compuestas que pueden aparecer en el interior de rutinas. Una sentencia compuesta puede contener múltiples sentencias, encerradas por las palabras **BEGIN y END**.

La cláusula **RETURNS** es obligatoria en la creación de funciones. Se usa para indicar el tipo de retorno de la función, y el cuerpo de la función debe contener un comando **RETURN valor**.

3.4.3 Ejemplos

Ejemplo de función sin parámetros:

¹Se encuentra en "C:\ProgramData\MySQL\MySQL Server 8.0".
Recuerda que la carpeta ProgramData será oculta.

```

DELIMITER $$
CREATE FUNCTION numTrabajadores() RETURNS int
BEGIN
    RETURN SELECT COUNT(*) FROM trabajadores;
END
$$
DELIMITER ;

```

Ejemplo de función con parámetros:

```

DELIMITER $$
CREATE FUNCTION funcionConParametro(nombre varchar(30))
RETURNS varchar(50)
BEGIN
    RETURN CONCAT('Hola ', nombre, '!');
END
$$
DELIMITER ;

```

Normalmente, las funciones se utilizan dentro de otras consultas y sus parámetros son campos de la/s tabla/s que se están consultando. Supongamos que tenemos una tabla de personas que almacena un campo "nombre". Podríamos utilizar la función anterior para mostrar una lista de saludos a todas las personas:

```

SELECT nombre, funcionConParametro(nombre)
FROM personas;

```

De esta forma, la función se utiliza en cada uno de los resultados de la consulta, dando el siguiente resultado:

	nombre	funcionConParametro(nombre)
►	Pepe	Hola Pepe!
	María	Hola María!
	Antonio	Hola Antonio!
	Julia	Hola Julia!
	Juan	Hola Juan!

Esta es la principal diferencia entre un procedimiento almacenado y una función. Un procedimiento ofrece un único resultado, mientras que una función se aplica en varias filas de resultados.

3.4.4 Borrado de funciones

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

Este comando se usa para borrar una función. La cláusula **IF EXISTS** evita que ocurra un error si el procedimiento no existe.

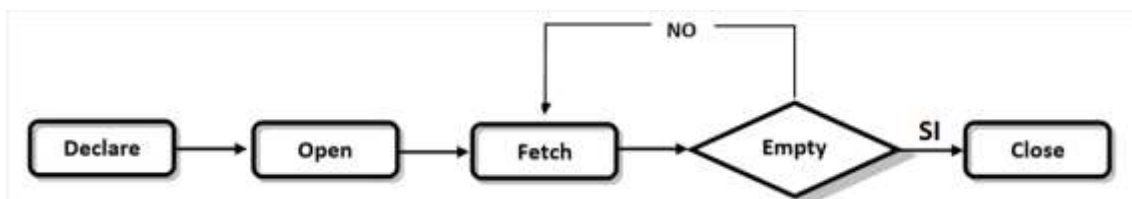
3.5 Cursores

Un cursor es una estructura de control que nos permite recorrer fila a fila una determinada consulta y guardar los datos de cada fila en variables para hacer uso de ellas.

Lo podéis entender como una variable que alberga todos los resultados de una consulta y, para acceder a sus valores, requerimos sacar las filas una a una con un bucle.

Cuando declaramos un cursor dentro de rutina debe aparecer antes de las declaraciones de los manejadores de errores (HANDLER) y después de la declaración de variables locales.

El manejador de errores nos permitirá saber cuándo hemos acabado de recorrer el cursor.



3.5.1 Declarar un cursor

El primer paso que tenemos que hacer para trabajar con cursores es declararlo. La sintaxis para declarar un cursor es:

```
DECLARE nombre_cursor CURSOR FOR consulta_select;
```

Este comando declara un cursor. Pueden definirse varios cursores en un procedimiento, pero cada cursor debe tener un nombre único.

El comando **SELECT** no puede tener una cláusula **INTO**.

3.5.2 Abrir un cursor (OPEN)

Una vez que hemos declarado un cursor tenemos que abrirlo con OPEN.

```
OPEN nombre_cursor;
```

Este comando abre un cursor declarado previamente. Es obligatorio abrir el cursor antes de comenzar a leer sus registros. Será cuando se abre el cursor cuando se ejecute la consulta y se almacene su resultado en el cursor.

3.5.3 Leer un cursor (FETCH)

Una vez que el cursor está abierto podemos ir obteniendo cada una de las filas con FETCH. La sintaxis es la siguiente:

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

Este comando lee el siguiente registro (si existe) usando el cursor abierto especificado. Es necesario disponer de tantas variables como columnas se estén seleccionando. El tipo de las variables también debe coincidir con el tipo de dato de las columnas seleccionadas. El número de variables y su orden, como es natural, también debe coincidir.

Cuando se está recorriendo un cursor y no quedan filas por recorrer se lanza el error NOT FOUND. Por eso cuando estemos trabajando con cursores será necesario declarar un *handler* para manejar este error.

```
DECLARE CONTINUE HANDLER FOR NOT FOUND ...
```

3.5.4 Cerrar un cursor

Cuando hemos terminado de trabajar con un cursor tenemos que cerrarlo.

```
CLOSE nombre_cursor;
```

3.5.5 Recorrer un cursor

Para recorrer un cursor es necesario utilizar una estructura repetitiva, por ejemplo, LOOP. En cada iteración del bucle realizamos los siguientes pasos:

1. Leemos un registro del cursor (FETCH)

2. Comprobamos que aún quedan filas por recorrer. Si no quedan más filas, terminamos el bucle.
3. Realizamos las operaciones necesarias con los valores del registro actual del cursor.

Los cursores deben declararse antes de declarar los handlers, y las variables y condiciones deben declararse antes de declarar cursores o handlers.

Veamos un ejemplo completo a continuación.

3.5.6 Ejemplo1: recorrido simple de un cursor

En este caso lo que buscamos obtener el número y nombre de todos empleados para introducirlos en una tabla. Usamos BDPedidos.

```
CREATEPROCEDURE cursor_demo1 ()
BEGIN
-- Declaración de variables
DECLARE finCursor BOOLEANDEFAULTFALSE;
DECLARE vNumEmp INT;
DECLARE vNomEmp VARCHAR(20);

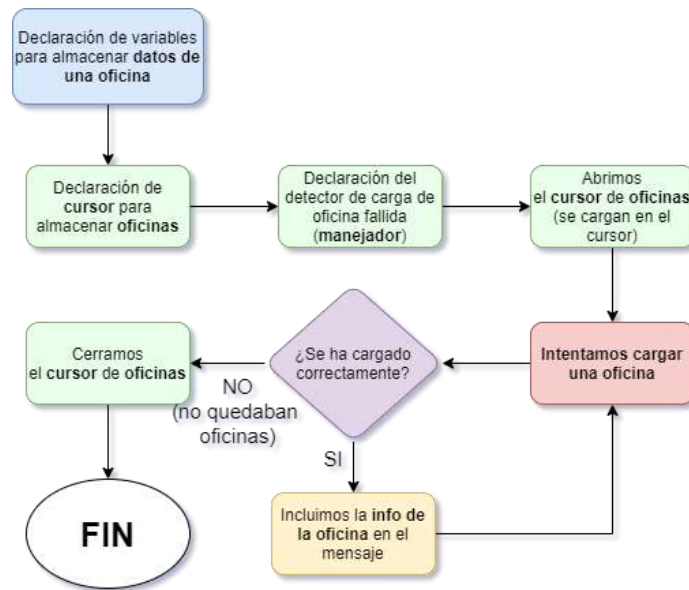
-- Declaración del cursor seleccionando el número y nombre
-- de todos los clientes
DECLARE cEmpleados CURSORFOR
        SELECT num_empleado, nombre FROM empleados;

-- Declaración del manejador de error. Si el cursor llega a su
-- fin el valor de la variable finCursor pasará a ser TRUE
DECLARECONTINUE HANDLER FORNOTFOUNDSET finCursor =TRUE;

-- Abrimos el cursor
OPEN cEmpleados;
-- Intento cargar la primera fila del cursor (el primer empleado)
FETCH cEmpleados INTO vNumEmp, vNomEmp;
-- Comenzamos el bucle para recorrer el resultado del cursor.
-- Mientras la última carga de una fila se haya realizado con
-- éxito, es decir, mientras queden empleados el bucle iterará
while(NOT finCursor)DO
-- Si no hemos terminado, insertamos el número y nombre
-- de cliente en una nueva tabla
INSERTINTOempleadosAntiguosVALUES(vNumEmp, vNomEmp);
-- Intento cargar otro empleado
FETCH cEmpleados INTO vNumEmp, vNomEmp;
ENDWHILE;
-- Cerramos el cursor (IMPORTANTE)
CLOSE cEmpleados;
END
```

3.5.7 Ejemplo 2: recorrido simple de un cursor

En este caso lo que buscamos obtener el número y ciudad de todas las oficinas. Usamos BDPedidos.



```

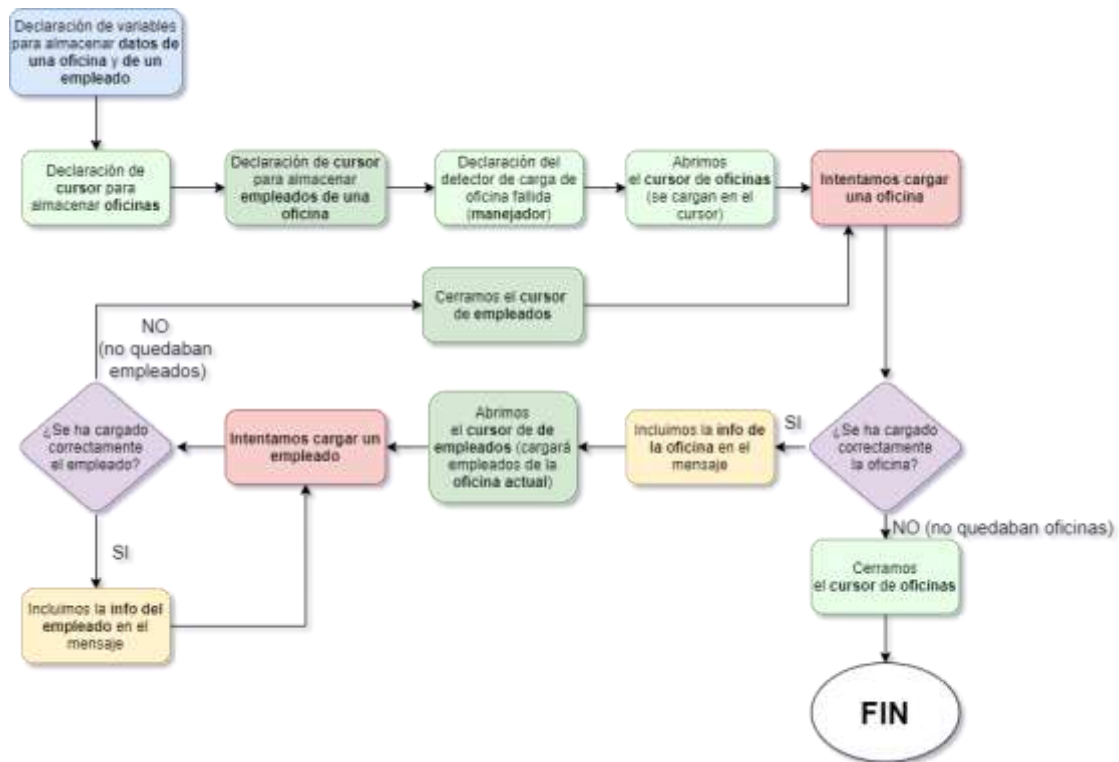
CREATEPROCEDURE cursor_demo2 ()
BEGIN
  -- Declaración de variables
  DECLARE finAlgunCursor BOOLEAN DEFAULT FALSE;
  DECLARE vNumOfi INT;
  DECLARE vCiudadOfi VARCHAR(20);
  DECLARE msg VARCHAR(10000) DEFAULT '';
  -- Declaración del cursor seleccionando el código
  -- y nombre de todas las oficinas
  DECLARE cOficinas CURSOR FOR SELECT num_oficina, ciudad FROM oficinas;

  -- Declaración del manejador de error. Si CUALQUIERA de los
  -- cursores llega a su fin, FINCURSOR pasará a ser TRUE
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET finAlgunCursor = TRUE;
  -- Abrimos el cursor de las oficinas
  OPEN cOficinas;
  -- Intento cargar la primera fila del cursor (la primera oficina)
  FETCH cOficinas INTO vNumOfi, vCiudadOfi;
  -- Bucle para recorrer el cursor de oficinas. Mientras
  -- la última carga de un cursor sea exitosa, continuará
  WHILE (NOT finAlgunCursor) DO
    -- Concateno info de una oficina al mensaje
    SET msg = CONCAT(msg, '\nOficina: ', vNumOfi, ' Ciudad: ', vCiudadOfi);
    -- Intento cargar otra oficina
    FETCH cOficinas INTO vNumOfi, vCiudadOfi;
  ENDWHILE;
  -- Cerramos el cursor oficina (IMPORTANTE)
  CLOSE cOficinas;
  select msg;
END $$

```

3.5.8 Ejemplo 3: recorrido anidado de cursores

En este caso lo que buscamos obtener es el número y ciudad de todas las oficinas y, de cada una, el número y nombre de cada empleado. Usamos BDPedidos. Se ha resaltado en **negrita** el código añadido al anterior ejercicio.



```

CREATEPROCEDURE cursor_demo3()
BEGIN
-- Declaración de variables
DECLARE finAlgunCursor BOOLEANDEFAULTFALSE;
DECLARE vNumOfi,vNumEmp INT;
DECLARE vCiudadOfi, vNomEmp VARCHAR(20);
DECLARE msg VARCHAR(10000)DEFAULT'';
-- Declaración del cursor seleccionando el código
-- y nombre de todas las oficinas
DECLARE cOficinas CURSORFORSELECT num_oficina, ciudad FROM oficinas;
-- Declaración del cursor de todos los clientes de
-- la oficina que estemos analizando en ese momento
DECLARE cEmpleados CURSORFORSELECT num_employado, nombre
FROM empleados WHERE oficina=vNumOfi;
-- Declaración del manejador de error. Si CUALQUIERA de los
-- cursores llega a su fin, FINCURSOR pasará a ser TRUE
DECLARECONTINUE HANDLER FORNOTFOUNDSET finAlgunCursor =TRUE;
-- Abrimos el cursor de las oficinas
OPEN cOficinas;
-- Intento cargar la primera fila del cursor (la primera oficina)
FETCH cOficinas INTO vNumOfi, vCiudadOfi;
-- Bucle para recorrer el cursor de oficinas. Mientras
-- la última carga de un cursor sea exitosa, continuará
while(NOT finAlgunCursor)DO-- Concateno info de una oficina al mensaje
SET msg =CONCAT(msg,'\nOficina: ',vNumOfi,' Ciudad: ',vCiudadOfi);
-- Abrimos el cursor de los empleados de la oficina actual
OPEN cEmpleados;
-- Intento cargar la primera fila del cursor (el primer
-- empleado de la oficina actual)
FETCH cEmpleados INTO vNumEmp, vNomEmp;
-- Bucle para recorrer el cursor de empleados de la oficina actual.
while(NOT finAlgunCursor)DO
SET msg =CONCAT(msg,'\nEmpleado: ',vNumEmp,' Nombre: ',vNomEmp);
-- Intento cargar otro empleado de la oficina actual
FETCH cEmpleados INTO vNumEmp, vNomEmp;
ENDWHILE;
CLOSE cEmpleados;
-- IMPORTANTE: DESACTIVAR FINCURSOR PARA QUE SIGA RECORRIENDO OFICINAS
SET finAlgunCursor =false;
-- Intento cargar otra oficina
FETCH cOficinas INTO vNumOfi, vCiudadOfi;
ENDWHILE;
-- Cerramos el cursor oficina (IMPORTANTE)
CLOSE cOficinas;
select msg;
END $$

```

4 PROGRAMACIÓN DE BBDD EN MYSQL III

4.1 Control de errores

4.1.1 Concepto de error

Durante la ejecución de procedimientos almacenados pueden darse errores que hagan que el procedimiento falle.

MySQL permite 'capturar' esos errores y tratarlos mediante programación permitiendo que el procedimiento continúe su ejecución.

Un caso típico podría ser cuando intentamos dar de alta (INSERT) una fila con una clave primaria duplicada. Al intentar hacer esto, el gestor 'rompe' la ejecución del procedimiento almacenado en la línea del INSERT e informa del problema. Otro ejemplo sería intentar modificar una tabla que no existe, etc.

4.1.2 Información de los errores

La lista de todos los posibles códigos de error de MySQL lo podéis consultar [en este enlace](#).

Cuando ocurre un error, la información del error incluye varios elementos: un código de error, un valor de SQLSTATE y una cadena de mensaje.

- ✓ **Código de error:** este valor es numérico. Es específico de MySQL y no es portátil a otros sistemas de bases de datos.
- ✓ **Valor de SQLSTATE:** este valor es una cadena de cinco caracteres (por ejemplo, '42S02'). Los valores de SQLSTATE se toman de ANSI SQL y ODBC y están más estandarizados que los códigos de error numéricos.
 - SQLWARNING: Cualquier SQLState que empiece por '01' (warning). *No detiene la ejecución.*
 - NOT FOUND: Cualquier SQLState que empiece por '02' (por ejemplo, en los cursores, cuando no hay más datos devuelve el SQLState '02000'). *No detiene la ejecución salvo que sea lanzado con SIGNAL.*
 - SQLEXCEPTION: Cualquier SQLState que NO empiece por '00', '01' o '02'. *Detiene la ejecución.*
- ✓ **Cadena de mensaje:** esta cadena proporciona una descripción textual del error.

4.1.3 Los manejadores de errores (HANDLERS)

Lo que haremos en MySQL es 'declarar' primero cuales son los errores que se van a capturar y cuáles serán las instrucciones que se deben ejecutar en caso de que se produzca.

Esto hará que el procedimiento almacenado continúe la ejecución con la siguiente línea a la que provocó el error.

Deben declararse después de las declaraciones de variables y cursores ya que

referencian a estos en su declaración.

El manejador puede ser de tres tipos:

- ✓ **CONTINUE:** La excepción o error generado no interrumpe la ejecución del procedimiento. Tras ejecutar el código `BEGIN . . . END` del manejador, continuará la ejecución de la instrucción siguiente a la que causó el error.
- ✓ **EXIT:** Permite poner fin a la ejecución del bloque de instrucciones o programa en el que se genera la excepción. Tras ejecutar el código `BEGIN . . . END` del manejador, no ejecutará la instrucción siguiente a la que causó el error.
- ✓ **UNDO:** No está soportada por el momento.

El manejador de error indica mediante un conjunto de instrucciones lo que hay que hacer en caso de que se produzca ese error.

Sintaxis:

```

DECLARE acción_del_manejador HANDLER
FOR condición_de_error [, condición_de_error]...
BEGIN
    Instrucciones de tratamiento de errores
END;

acción_del_manejador:{
CONTINUE
| EXIT
| UNDO
}

condición_de_error:{
    código_de_error_mysql
| SQLSTATE[VALUE] valor_sqlstate
| nombre_de_condición
| SQLWARNING
| NOTFOUND
| SQLException
}

```

4.1.4 Tipos de declaración de manejadores

4.1.4.1 Ejemplo indicando el número de error de MySQL

En este ejemplo estamos declarando un *handler* que se ejecutará cuando se produzca el error 1051 de MySQL, que ocurre cuando se intenta acceder a una tabla que no existe en la base de datos. En este caso la acción del *handler* es **CONTINUE** lo que quiere decir que después de ejecutar las instrucciones especificadas en el cuerpo del *handler* el procedimiento almacenado continuará su ejecución.

```

DECLARECONTINUE HANDLER FOR1051
BEGIN
    -- body of handler
END;

```

4.1.4.2 Ejemplo indicando el **SQLSTATE**

También podemos indicar el valor de la variable **SQLSTATE**. Por ejemplo, cuando se intenta acceder a una tabla que no existe en la base de datos, el valor de la variable **SQLSTATE** es 42S02.

```
DECLARECONTINUE HANDLER FOR SQLSTATE '42S02'
BEGIN
  -- body of handler
END;
```

4.1.4.3 Ejemplo con NOTFOUND

Es equivalente a indicar todos los valores de SQLSTATE que empiezan con 02. Lo usaremos cuando estemos trabajando con cursores para controlar qué ocurre cuando un cursor alcanza el final del *data set* (contenido del cursor). Si no hay más filas disponibles en el cursor, entonces ocurre una condición de NO DATA con un valor de SQLSTATE igual a 02000.

Para detectar esta condición usamos un *handler*. Hasta ahora hemos usado siempre la instrucción **SET** `finCursor =TRUE;` para activar la “bandera” y salir del bucle que recorre el cursor.

```
DECLARECONTINUE HANDLER FOR NOT FOUND
BEGIN
  -- body of handler
END;
```

4.1.4.4 Ejemplo con SQLWARNING

Es equivalente a indicar todos los valores de SQLSTATE que empiezan con 01.

```
DECLARECONTINUE HANDLER FOR SQLWARNING
BEGIN
  -- body of handler
END;
```

4.1.4.5 Ejemplo con SQLEXCEPTION

Es equivalente a indicar todos los valores de SQLSTATE que **no** empiezan por 00, 01 y 02.

```

DECLARECONTINUE HANDLER FOR SQLEXCEPTION
BEGIN
    -- body of handler
END;

```

4.1.5 Ejemplo de declaración de un manejador

```

CREATEPROCEDURE aniadir_empleado (numEmp int, nombre
                                varchar(14),mujer boolean)

BEGIN
    DECLARE genero char(1);
    -- Declaramos un manejador para errores SQLSTATE 23000
    -- (1022 en MySQL)
    -- Si hay error continuará la ejecución pero mostrará el error
    DECLARECONTINUE HANDLER FORSQLSTATE'23000'
    BEGIN
        -- Informamos del error
        SELECT'Error inserción'AS TIPO_ERROR,
            'Clave primaria duplicada.'AS CAUSA_ERROR,
            numEmp AS ID_EMPLEADO;
    END;
    -- Procesamos los parámetros de entrada
    IF(mujer) THEN
        SET genero ='F';
    ELSE
        SET genero ='M';
    ENDIF;
    -- Intentamos insertar el empleado/a
    INSERTINTO employees(emp_no,name,gender)
    VALUES (numEmp,nombre,genero);
END$$

```

4.1.6 Obtención de información del error: SQLWARNING y SQLEXCEPTION

Si queremos acceder a la información del error concreto que provocó la ejecución del *handler* tan solo tenemos que usar un código similar a este:

```

DECLARECONTINUE HANDLER FOR1643, 1452 -- Ejemplo con varios nº de error
BEGIN
-- Las variables num_error, estado_sql y msgErrordeberán haber sido
-- declaradas previamente
GETDIAGNOSTICS CONDITION 1 num_error = MYSQL_ERRNO,
                        estado_sql= RETURNED_SQLSTATE,
                        msgError= MESSAGE_TEXT;

END;

```

4.1.7 Ejemplo de obtención de información del error por parte del manejador (GET DIAGNOSTICS)

```

CREATEPROCEDURE infoError(valor int)
BEGIN
    DECLARE numErrorint;
    DECLARE codError VARCHAR(5);
    DECLARE msgError VARCHAR(100);
    DECLAREEXIT HANDLER FORSQLEXCEPTION
    BEGIN
    GETDIAGNOSTICS CONDITION 1
    numError = MYSQL_ERRNO,
    codError = RETURNED_SQLSTATE,
    msgError = MESSAGE_TEXT;
    SELECTCONCAT('Nº error: ',numError,
    'Código de error: ',codError,
    ' Mensaje: ',msgError)AS 'Mensaje de error';
    END;
    INSERTINTO test VALUES(valor);
    SELECT'Inserción realizada con éxito';
    END$$
DELIMITER ;
CALL infoError(7);

```

4.1.8 Lanzamiento de excepciones (SIGNAL)

SIGNAL es la manera que tiene MySQL de que un usuario pueda lanzar una excepción.

Puede llevar como dato, un nombre de excepción (definida con DECLARE CONDITION o un código SQLState de 5 caracteres).

Si empleamos el código de 5 caracteres, este no debe comenzar con '00' (tanto si empleamos un código directamente como si empleamos un nombre de excepción basado en un código que empiece por 00) ya que esto indica que es correcto y por tanto no lanzará la excepción.

Existe un código estándar para indicar que la excepción está definida por el usuario (no es ninguna de las que ya existen): 45000

Veamos varios ejemplos:

```
SIGNAL SQLSTATE '45000';
```

```
DECLARE no_existe_tabla CONDITION FORSQLSTATE '42S02';
SIGNAL no_existe_tabla;
```

Además de provocar una excepción, podemos indicar el número de error MYSQL y un texto asociado a la excepción.

```
SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT = 'Esa tabla no existe',
MYSQL_ERRNO = 1051;
```

Como siempre, el uso que hagamos de las excepciones va a depender del tipo de programa que estemos desarrollando.

Como comenté antes, si estamos a desarrollar un programa cliente, el control de las excepciones lo haremos en dicho programa, y tendremos que programar una respuesta a la misma.

Ojo

SIGNAL no acepta funciones para asignar valor a MESSAGE_TEXT, por lo que deberás usar, una cadena literal o bien una variable.

```
SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT = CONCAT('El ID', id_a_insertar, 'ya existe'),
MYSQL_ERRNO = 1051;
```

Lo haremos así:

```
DECLARE msg_error VARCHAR(100);
. . .
SET msg_error = CONCAT('El ID', id a insertar, 'ya existe');
SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT = msg_error,
MYSQL_ERRNO = 1051;
```

4.1.9 Ejemplo de lanzamiento y captura de excepciones de usuario

```

CREATEPROCEDURE errorSignal(param int)
BEGIN
DECLARE num_err int;
DECLARE est_sql varchar(6);
DECLARE msgErr varchar(100);
DECLAREEXIT HANDLER FORSQLEXCEPTION
BEGIN
GETDIAGNOSTICS CONDITION 1est_sql= RETURNED_SQLSTATE,
                        num_err=MYSQL_ERRNO,msgErr = MESSAGE_TEXT;
SELECTCONCAT('Cód err: ', num_err, ' Est. SQL: ',
                        est_sql,' Mensaje: ',msgErr);

END;
IFEXISTS(SELECT*FROM test WHERE campoPK=param) THEN
SIGNAL SQLSTATE'45000'
SET MESSAGE_TEXT ='Ya existe una fila con esa PK',
MYSQL_ERRNO =1062;
ENDIF;
INSERTINTO test VALUES (param);
SELECT'Inserción realizada con éxito';
END $$

```

4.1.10 Ampliación: Declaración de excepciones

Cuando capturamos excepciones, podremos capturar dos tipos de excepciones:

- Ya definidas por el gestor (por ejemplo, `DUPLICATE ENTRY FOR KEY`, que se produce cuando intentamos añadir dos filas con la misma clave primaria)
- Excepciones personalizadas por el usuario. Por ejemplo, puedo lanzar una excepción cuando intente añadir un animal con una altura negativa.

En cualquiera de los dos casos, cada excepción va a estar asociada a un número ya predefinido por el gestor.

Para hacer más legible el manejo de excepciones, puedo asociar dicho número a un nombre de excepción y después controlar la captura por dicho nombre y no por el número asociado.

Podéis consultar la lista de códigos con el tipo de excepción asociado [en este enlace](#). Por ejemplo, el código de error MySQL que aparece cuando se intenta borrar una tabla que no existe es el 1051.



Dentro de un procedimiento voy a poder capturar esta excepción, pero en vez de capturarla (veremos la instrucción SQL a continuación) por su número (1051) puedo hacerlo por un nombre de la forma:

```
DECLARE no_existe_tabla CONDITION FOR1051;
```

Veremos a continuación como capturar la excepción empleando este nombre.

En la lista de mensajes de error aparecen dos tipos de errores asociados a cada mensaje:

- ✓ Error Code: Son números específicos de Mysql que no valen para otros gestores.
- ✓ SQLState Code: Cadena de 5 dígitos basado en ANSI SQL y ODBC y por lo tanto sus códigos son 'estandarizados'.

Emplear uno u otro va a depender de nuestro programa cliente. Si este va a ser empleado en sistemas gestores diferentes (por ejemplo, ORACLE, SQL Server) podría ser mejor emplear los SQLState Code. En caso contrario, mejor los específicos de cada gestor.

En el caso de querer dar un nombre a la excepción empleando el SQLState la sintaxis cambia:

```
DECLARE no_existe_tabla CONDITION FORSQLSTATE'42S02';
```

4.2 Transacciones I

4.2.1 Introducción

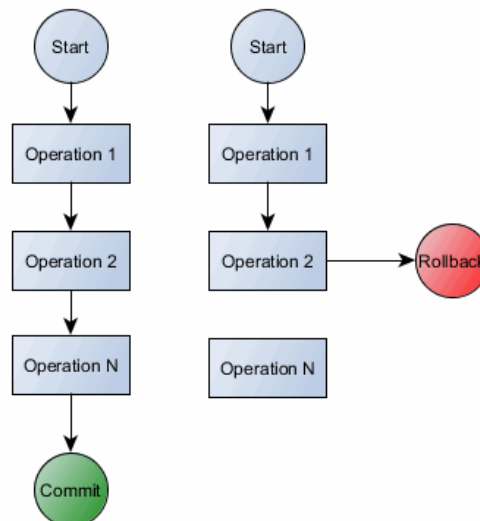
En términos teóricos, una transacción es un conjunto de tareas relacionadas que se realizan de forma satisfactoria o incorrecta como una unidad.

Una transacción SQL es un conjunto de sentencias SQL que se ejecutan formando una unidad lógica de trabajo (LUW del inglés Logic Unit of Work), es decir, en forma **indivisible o atómica**.

MySQL nos permite realizar transacciones en sus tablas si hacemos uso del motor de almacenamiento **InnoDB** (MyISAM no permite el uso de transacciones).

El uso de transacciones nos permite realizar operaciones de forma segura y recuperar datos si se produce algún fallo en el servidor durante la transacción, pero por otro lado las transacciones pueden aumentar el tiempo de ejecución de las instrucciones.

Las transacciones deben cumplir las cuatro propiedades **ACID**.



Existen 4 propiedades necesarias, que son conocidas como **propiedades ACID**:

1. Atomicidad (Atomicity)
2. Consistencia (Consistency)
3. Aislamiento (Isolation)
4. Durabilidad (Durability).

Estas propiedades garantizan un comportamiento predecible, reforzando la función de las transacciones como proposiciones de todo o nada.

4.2.2 Propiedades ACID

1. **Atomicidad:** Una transacción es una unidad de trabajo el cual se realiza en su totalidad o no se realiza en ningún caso. Las operaciones asociadas a una transacción comparten normalmente un objetivo común y son

interdependientes. Si el sistema ejecutase únicamente una parte de las operaciones, podría poner en peligro el objetivo final de la transacción.

2. **Consistencia:** Una transacción es una unidad de integridad porque mantiene la coherencia de los datos, transformando un estado coherente de datos en otro estado de datos igualmente coherente.
3. **Aislamiento:** Una transacción es una unidad de aislamiento, permitiendo que transacciones concurrentes se comporten como si cada una fuera la única transacción que se ejecuta en el sistema. El aislamiento requiere que parezca que cada transacción sea la única que manipula el almacén de datos, aunque se puedan estar ejecutando otras transacciones al mismo tiempo. Una transacción nunca debe ver las fases intermedias de otra transacción.
4. **Durabilidad:** Una transacción también es una unidad de recuperación. Si una transacción se realiza satisfactoriamente, el sistema garantiza que sus actualizaciones se mantienen, aunque el equipo falle inmediatamente después de la confirmación. El registro especializado permite que el procedimiento de reinicio del sistema complete las operaciones no finalizadas, garantizando la permanencia de la transacción.

En términos más prácticos, una transacción está formada por una serie de instrucciones DML. Una transacción comienza con la primera instrucción DML que se ejecute y finaliza con una operación COMMIT (si la transacción se confirma) o una operación ROLLBACK (si la operación se cancela). Hay que tener en cuenta que cualquier instrucción DDL (definición de bases de datos) o DCL (definición de usuarios y permisos) da lugar a un COMMIT implícito, es decir todas las instrucciones DML ejecutadas hasta ese instante pasan a ser definitivas.

Un ejemplo típico de esto es una transacción bancaria. Por ejemplo, si una cantidad de dinero es transferida de la cuenta de una persona a otra, se requerirán por lo menos dos consultas:

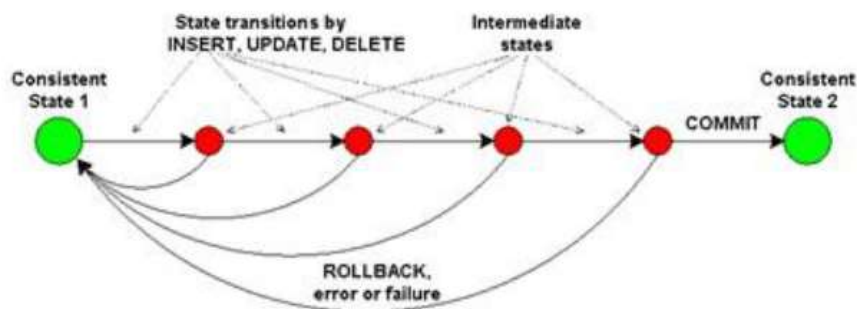
```
UPDATE cuentas SET balance = balance - cantidad_transferida WHERE
cliente = personal;

UPDATE cuentas SET balance = balance + cantidad_transferida WHERE
cliente = persona2;
```

Estas dos consultas deben trabajar bien, pero ¿qué sucede si ocurre algún imprevisto y “se cae” el sistema después de que se ejecuta la primera consulta, pero la segunda aún no se ha completado?

La **persona1** tendrá una cantidad de dinero eliminada de su cuenta, y creerá que ha realizado su pago, sin embargo, la **persona2** estará enfadada puesto que pensará que no se le ha depositado el dinero que le deben.

En este ejemplo tan sencillo se ilustra la necesidad de que las consultas sean ejecutadas de manera conjunta, o en su caso, que no se ejecute ninguna de ellas. Es aquí donde las transacciones toman un papel muy importante.



4.2.3 START TRANSACTION

Por defecto, cada vez que ejecutamos una sentencia en MySQL se hace permanente automáticamente. Esta situación viene determinada por la variable de configuración “autocommit”.

La instrucción START TRANSACTION deshabilita el autocommit, permitiendo ejecutar un conjunto de sentencias sin que tengan efecto permanente sobre la base de datos hasta que no se ejecute una instrucción COMMIT o ROLLBACK.

Para poder trabajar con transacciones en MySQL es necesario utilizar InnoDB.

4.2.4 COMMIT

La instrucción COMMIT hace que los cambios realizados por la transacción sean definitivos, irrevocables. Se dice que tenemos una **transacción confirmada**. Sólo se debe utilizar si estamos de acuerdo con los cambios, conviene asegurarse mucho antes de realizar el COMMIT ya que las instrucciones ejecutadas pueden afectar a miles de registros. Además, el cierre correcto de la sesión da lugar a un COMMIT, aunque siempre conviene ejecutar explícitamente esta instrucción a fin de asegurarnos de lo que hacemos.

4.2.5 ROLLBACK

Esta instrucción regresa a la instrucción anterior al inicio de la transacción, normalmente el último COMMIT, la última instrucción DDL o DCL o al inicio de sesión. Anula definitivamente los cambios, por lo que conviene también asegurarse de esta operación. Un abandono de sesión incorrecto o un problema de comunicación o de caída del sistema dan lugar a un ROLLBACK implícito.

4.2.6 Estado de los datos durante la transacción

Si se inicia una transacción usando comandos DML hay que tener en cuenta que:

- Se puede volver a la instrucción anterior a la transacción cuando se desee.
- Las instrucciones de consulta SELECT realizadas por el usuario que inició la transacción muestran los datos ya modificados por las instrucciones DML.
- El resto de usuarios ven los datos tal cual estaban antes de la transacción, de hecho, los registros afectados por la transacción aparecen bloqueados hasta que la transacción finalice. Esos usuarios no podrán modificar los valores de dichos registros.
- Tras la transacción todos los usuarios ven los datos tal cual quedan tras el fin de transacción. Los bloqueos son liberados y los puntos de ruptura borrados.

4.2.7 Cómo revertir la instrucción/transacción que genera un error

Podemos utilizar el manejo de errores para decidir si hacemos ROLLBACK de una transacción. En el siguiente ejemplo vamos a capturar los errores que se produzcan de tipo `SQLException` y `SQLWarning`.

Ejemplo:

```

DELIMITER $$
CREATEPROCEDURE transaccion_en_mysql()
BEGIN
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
-- ERROR
ROLLBACK;
END;

DECLARE EXIT HANDLER FOR SQLWARNING
BEGIN
-- WARNING
ROLLBACK;
END;

STARTTRANSACTION;
-- Sentencias SQL (Pueden incluir un SIGNAL)
COMMIT;
END
$$

```

En lugar de tener un manejador para cada tipo de error, podemos tener uno común para todos los casos.

```

DELIMITER $$
CREATEPROCEDURE transaccion_en_mysql()
BEGIN
DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING
BEGIN
-- ERROR, WARNING
ROLLBACK;
END;

STARTTRANSACTION;
-- Sentencias SQL (Pueden incluir un SIGNAL)
COMMIT;
END
$$

```

4.3 Transacciones II

4.3.1 SAVEPOINT, ROLLBACK TO SAVEPOINT y RELEASE SAVEPOINT

Si trabajamos con tablas InnoDB en MySQL también es posible hacer uso de las sentencias: SAVEPOINT, ROLLBACK TO SAVEPOINT y RELEASE SAVEPOINT.

- ✓ SAVEPOINT: Nos permite establecer un punto de recuperación dentro de la transacción, utilizando un identificador. Si en una transacción existen

dos SAVEPOINT con el mismo nombre sólo se tendrá en cuenta el último que se ha definido.

- ✓ **ROLLBACK TO SAVEPOINT:** Nos permite hacer un ROLLBACK deshaciendo sólo las instrucciones que se han ejecutado hasta el SAVEPOINT que se indique.
- ✓ **RELEASE SAVEPOINT:** Elimina un SAVEPOINT.

A continuación, se muestra la sintaxis que aparece en la [documentación oficial para crear SAVEPOINT](#).

SAVEPOINT

Esta instrucción permite establecer un punto de ruptura. El problema de la combinación ROLLBACK/COMMIT es que un COMMIT acepta todo y un ROLLBACK anula todo. SAVEPOINT permite señalar un punto intermedio entre el inicio de la transacción y la situación actual. Su sintaxis es:

```
-- ... instrucciones DML ...  
SAVEPOINTnombre  
-- ... instrucciones DML ...
```

Para regresar a un punto de ruptura concreto se utiliza ROLLBACK TO SAVEPOINT seguido del nombre dado al punto de ruptura. También es posible hacer ROLLBACK TO nombre de punto de ruptura. Cuando se vuelve a un punto marcado, las instrucciones que siguieron a esa marca se anulan definitivamente.

Ejemplo de uso:

```

START TRANSACTION;

CREATETABLET (FECHADATE) ;

INSERTINTOTVALUES ('2017-01-01');
INSERTINTOTVALUES ('2017-02-01');

SAVEPOINTfebrero;

INSERTINTOTVALUES ('2017-03-01');
INSERTINTOTVALUES ('2017-04-01');

SAVEPOINTabril;

INSERTINTOTVALUES ('2017-05-01');

ROLLBACKTOfebrero;

-- También puede escribirse ROLLBACK TO SAVEPOINT febrero;
-- En este ejemplo sólo se guardan en la tabla los 2 primeros
-- registros o filas.

```

Sintaxis extendida

```

SAVEPOINTidentifier
ROLLBACK [WORK] TO [SAVEPOINT] identifier
RELEASE SAVEPOINTidentifier

```

Otro ejemplo

```

DROPDATABASEIFEXISTS test;
CREATEDATABASE test CHARACTERSET utf8mb4;
USE test;

CREATETABLE producto (
  idINT UNSIGNED AUTO_INCREMENT PRIMARYKEY,
  nombre VARCHAR(100) NOTNULL,
  precio DOUBLE
);

INSERTINTO producto (id, nombre) VALUES (1, 'Primero');
INSERTINTO producto (id, nombre) VALUES (2, 'Segundo');
INSERTINTO producto (id, nombre) VALUES (3, 'Tercero');
-- 1. Comprobamos las filas que existen en la tabla
SELECT*
FROM producto;

```

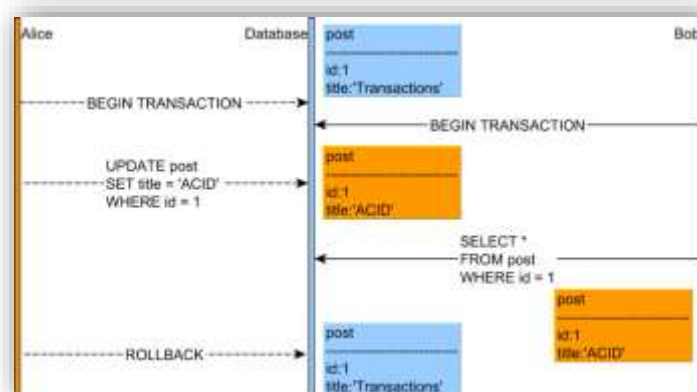
```
-- 2. Ejecutamos una transacción que incluye un SAVEPOINT
START TRANSACTION;
INSERT INTO producto (id, nombre) VALUES (4, 'Cuarto');
SAVEPOINT sp1;
INSERT INTO producto (id, nombre) VALUES (5, 'Cinco');
INSERT INTO producto (id, nombre) VALUES (6, 'Seis');
ROLLBACK TO sp1;

-- 3. ¿Qué devolverá esta consulta?
SELECT*
FROM producto;
```

4.3.2 Acceso concurrente a los datos

Cuando dos transacciones distintas intentan acceder concurrentemente a los mismos datos pueden ocurrir los siguientes problemas:

4.3.2.1 Dirty Read (Lectura sucia)



Sucede cuando una segunda transacción lee datos que están siendo modificados por una transacción antes de que haga COMMIT.

Dicho de otra manera, se produce cuando antes de finalizar la transacción1, otra accede a los mismos datos, por tanto, la segunda transacción consulta unos datos que luego serán modificados, bien por una modificación de los datos, bien por una ROLLBACK.

Ejemplo

Si tenemos en la tabla RECAMBIOS una columna denominada Stock cuyo valor inicial es 12 unidades, supongamos que dos transacciones concurrentes actualizan el valor del stock de la siguiente forma:

Transacción 1: Compra de 100 unidades del **recambio A** y luego deshace la transacción.

Transacción 2: Lee el valor del Stock del **recambio A**.

Ejecución correcta

TRANS.	OPERACIÓN	VALOR Stock
Trans 1	SELECT Stock	12
Trans 1	UPDATE Stock = 12 + 100	112
Trans 1	ROLLBACK	12
Trans 2	SELECT Stock	12

Lectura sucia por ROLLBACK

TRANS.	OPERACIÓN	VALOR Stock
Trans 1	SELECT Stock	12
Trans 1	UPDATE Stock = 12 + 100	112
Trans 2	SELECT Stock	112
Trans 1	ROLLBACK	12

⇒ Error de coherencia

Transacción 1: Compra de 100 unidades del **recambio A** y luego compra otros 50 y confirma.

Transacción 2: Lee el valor del Stock del **recambio A**.

Ejecución correcta

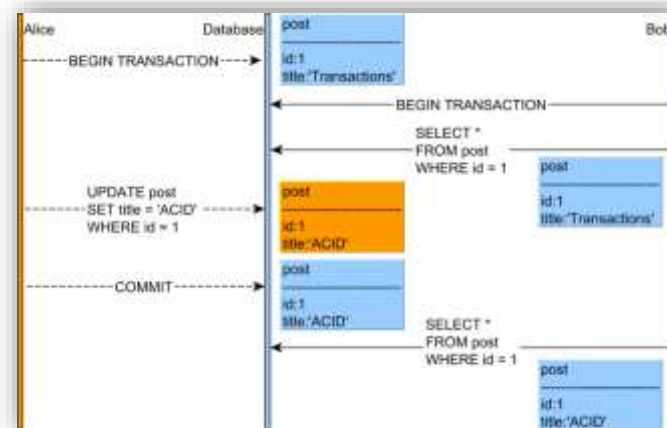
Lectura sucia por modificación

TRANS.	OPERACIÓN	VALOR Stock
Trans 1	SELECT Stock	12
Trans 1	UPDATE Stock = 12 + 100	112
Trans 1	UPDATE Stock = 112 + 50	162
Trans 1	COMMIT	162
Trans 2	SELECT Stock	162

TRANS.	OPERACIÓN	VALOR Stock
Trans 1	SELECT Stock	12
Trans 1	UPDATE Stock = 12 + 100	112
Trans 2	SELECT Stock	122
Trans 1	UPDATE Stock = 112 + 50	162
Trans 1	COMMIT	162

Error de coherencia

4.3.2.2 Nonrepeatable Read (Lectura No Repetible)



Sucede en el caso de que una transacción vuelve a leer datos que leyó previamente y encuentra que han sido modificados por otra transacción. Se produce, por ejemplo, cuando la transacción 1 lee un registro, otra modifica el mismo registro y confirma la modificación, y la transacción 1 vuelve a leer el registro.

Ejemplo

Si tenemos en la tabla RECAMBIOS una columna denominada Stock cuyo valor inicial es 12 unidades, supongamos que dos transacciones concurrentes acceden el valor del stock de la siguiente forma:

Transacción 1: Requiere una lectura sucesiva al stock del **recambio A**.

Transacción 2: Compra de 100 unidades del **recambio A** y confirma la compra.

Ejecución correcta

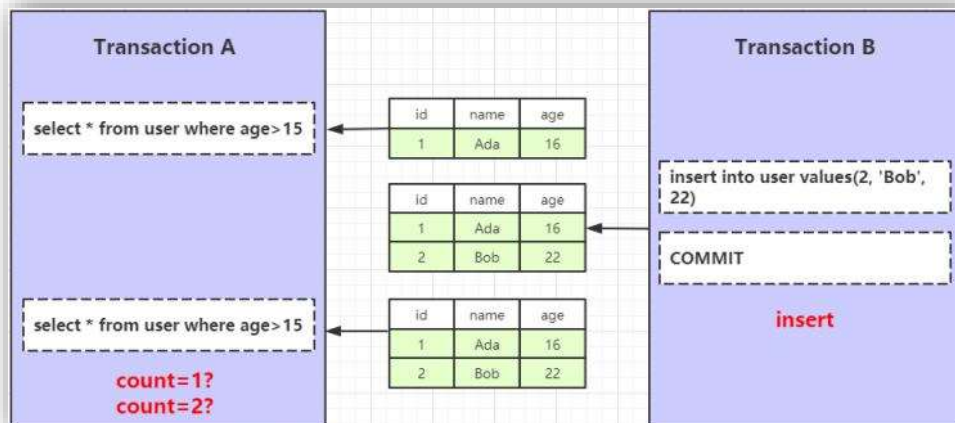
TRANS.	OPERACIÓN	VALOR Stock
Trans 1	SELECT Stock	12
Trans 1	SELECT Stock	12
Trans 2	UPDATE Stock = 12 + 100	112
Trans 2	COMMIT	112

Lectura no repetible por la Transacción 1

TRANS.	OPERACIÓN	VALOR Stock
Trans 1	SELECT Stock	12
Trans 2	UPDATE Stock = 12 + 100	112
Trans 2	COMMIT	112
Trans 1	SELECT Stock	112

Error de coherencia

4.3.2.3 Phantom Read (Lectura fantasma)



Sucede cuando una transacción lee unos datos que no existían cuando se inició la transacción. Por ejemplo, cuando una transacción incluye operaciones de agregado (sumar, contar, etc.) sobre unos datos, mientras otra transacción los actualiza. Si la función lee unos datos antes de actualizar y otros después el resultado es inconsistente.

Ejemplo

Si tenemos en la tabla RECAMBIOS los siguientes recambios:

IdRecambio	VALOR Stock
X	12
Y	15
Z	20

Transacción 1: Calcula la cantidad de recambios con stock > 15.

Transacción 2: Actualiza el Stock del **recambio Y** de 15 a 17.

Ejecución correcta

TRANS.	OPERACIÓN	Resultado
Trans 1	Nº Recambios con stock > 15	Z (20 ud.)
Trans 1	Nº Recambios con stock > 15	Z (20 ud.)
Trans 2	Modifico Stock de Y a 17 ud.	OK
Trans 2	COMMIT	OK

Lectura fantasma al incluirse un nuevo recambio entre los sucesivos conteos

TRANS.	OPERACIÓN	Resultado
Trans 1	Nº Recambios con stock > 15	Z (20 ud.)
Trans 2	Modifico Stock de Y a 17 ud.	OK
Trans 2	COMMIT	OK
Trans 1	Nº Recambios con stock > 15	Y (17 ud.) Z (20 ud.)

Error de coherencia

Ojo: Aunque se haya hubiera el registro Z se sigue pudiendo modificar el registro Y

4.3.3 Niveles de aislamiento

Para evitar que sucedan los problemas de acceso concurrente que hemos comentado en el punto anterior podemos establecer diferentes niveles de aislamiento.

- ✓ **Read Uncommitted.** En este nivel no se realiza ningún bloqueo, por lo tanto, permite que sucedan los tres problemas.
- ✓ **Read Committed.** Bloquea los registros modificados (y no los leídos) por una transacción hasta que termina la transacción que los modificó. En este caso los datos leídos por una transacción pueden ser modificados por otras transacciones, por lo tanto, se pueden dar los problemas *Phantom Read* y *Non Repeatable Read*.

Ejecución correcta

TRANS.	OPERACIÓN	VALOR Stock
Trans 1	SELECT Stock	12
Trans 1	UPDATE Stock = 12 + 100	112
Trans 1	ROLLBACK	12
Trans 2	SELECT Stock	12

Se impide la "Lectura no repetible"
porque se bloquea el registro
escrito por Trans. 1

TRANS.	OPERACIÓN	VALOR Stock
Trans 1	SELECT Stock	12
Trans 1	UPDATE Stock = 12 + 100	112
Trans 2	SELECT Stock	112
Trans 1	ROLLBACK	12

- ✓ **Repeatable Read.** Bloquealos registros modificados y los leídos por una transacción se bloquean hasta que termina la transacción que los modificó.

En este nivel ningún registro modificado ni leído con un SELECT por una transacción 1 puede ser leído **ni modificado** en otra transacción 2 hasta que termine la transacción 1. Por lo tanto, **sólo puede suceder el problema del Phantom Read.**

Ejecución correcta

TRANS.	OPERACIÓN	VALOR Stock
Trans 1	SELECT Stock	12
Trans 1	SELECT Stock	12
Trans 1	COMMIT	
Trans 2	UPDATE Stock = 12 + 100	112
Trans 2	COMMIT	112

Se impide la "Lectura no repetible"
se bloquea el registro leído por
Trans. 1

TRANS.	OPERACIÓN	VALOR Stock
Trans 1	SELECT Stock	12
Trans 2	UPDATE Stock = 12 + 100	112
Trans 2	COMMIT	112
Trans 1	SELECT Stock	112
Trans 1	COMMIT	

- ✓ **Serializable.** En este caso las transacciones se ejecutan unas detrás de otras, sin que exista la posibilidad de concurrencia.

Ejecución correcta

TRANS.	OPERACIÓN	Resultado
Trans 1	Nº Recambios con stock > 15	Z (20 ud.)
Trans 1	Nº Recambios con stock > 15	Z (20 ud.)
Trans 2	Modifico Stock de Y a 17 ud.	OK
Trans 2	COMMIT	OK

“Lectura fantasma” no sucede porque la Trans. 2 no empieza antes de la Trans 2.

TRANS.	OPERACIÓN	Resultado
Trans 1	Nº Recambios con stock > 15	Z (20 ud.)
Trans 2	Modifico Stock de Y a 17 ud.	OK
Trans 2	COMMIT	OK
Trans 1	Nº Recambios con stock > 15	Z (20 ud.)

Ojo: Aunque se haya bloqueado el registro Z se sigue pudiendo modificar el registro Y

El nivel de aislamiento que utiliza **InnoDB** por defecto es **Repeatable Read**.

4.3.4 Políticas de bloqueo

Cuando una transacción accede a los datos lo hace de forma exclusiva, de modo que una transacción no podrá acceder a los datos que están siendo utilizados por una transacción hasta que ésta haya terminado.

El bloqueo de los datos se puede realizar a nivel de:

- ✓ Base de datos.
- ✓ Tabla.
- ✓ Fila.
- ✓ Columna.

InnoDB realiza por defecto un bloqueo a nivel de fila.

Ejemplo

En este ejemplo vamos a simular que hay dos usuarios que quieren acceder de forma concurrente a los mismos datos de una tabla. Para simular los dos usuarios vamos a iniciar dos terminales para conectarnos a un servidor MySQL. Desde el **terminal A** vamos a ejecutar las siguientes sentencias SQL:

```

DROPDATABASEIFEXISTS test;
CREATEDATABASE test CHARACTERSET utf8mb4;
USE test;

CREATETABLE cuentas (
  idINTEGER UNSIGNED PRIMARYKEY,
  saldo DECIMAL(11,2) CHECK (saldo >=0)
);

INSERTINTO cuentas VALUES (1, 1000);
INSERTINTO cuentas VALUES (2, 2000);
INSERTINTO cuentas VALUES (3, 0);

-- 1. Ejecutamos una transacción para transfereir dinero entre dos cuentas
STARTTRANSACTION;
UPDATE cuentas SET saldo = saldo -100WHEREid=1;
UPDATE cuentas SET saldo = saldo +100WHEREid=2;

```

NOTA: Observe que la transacción no que estamos ejectionando en el terminal A todavía no ha finalizado, porque no hemos ejecutado COMMIT ni ROLLBACK.

Ahora desde el **terminal B** ejecute las siguientes sentencias SQL:

```

-- 1. Seleccionamos la base de datos
USE test;

-- 2. Observamos los datos que existen en la tabla cuentas
SELECT*
FROM cuentas;

-- 3. Intentamos actualizar el saldo de una de las cuentas que está siendo
utilizada en la transacción del terminal A
UPDATE cuentas SET saldo = saldo -100WHEREid=1;

```

¿Qué es lo que ha ocurrido en el **terminal B**? ¿Puedo acceder a los datos para consultarlos? ¿Y para modificarlos? ¿Puedo modificar desde el **terminal B** una cuenta bancaria que no esté siendo utilizada por la transacción del **terminal A**?

Ahora ejecute COMMIT en el **terminal A** para finalizar la transacción que estaba sin finalizar. ¿Qué es lo que ha sucedido?

4.4 Disparadores

4.4.1 Sintaxis básica de creación de un disparador

```
CREATE TRIGGER trigger_name {BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name FOR EACH ROW
trigger_body;
```

4.4.2 Concepto

Un **trigger** es un objeto de la base de datos que está asociado con una tabla y que se activa cuando ocurre un evento sobre la tabla. Por ejemplo, puedes definir un disparador que se invoque automáticamente antes de que se inserte una nueva fila en una tabla concreta.

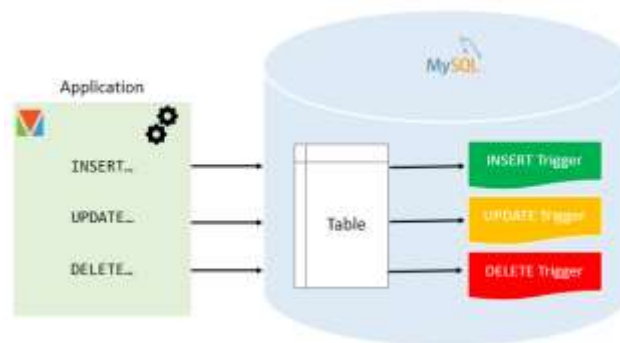
Ojo: en operaciones masivas como, por ejemplo, una instrucción **UPDATE** o **DELETE** sobre varias filas, el disparador se ejecutará tantas veces como filas sean afectadas por el disparador.

Tiempo de ejecución:

- ✓ **BEFORE:** El disparador se ejecuta **antes** de la instrucción que lo disparó.
- ✓ **AFTER:** El disparador se ejecuta **después** de la instrucción que lo disparó.

Las instrucciones sobre la tabla que lanzarán el disparador serán:

- ✓ **INSERT:** El trigger se activa cuando se inserta una nueva fila sobre la tabla asociada.
- ✓ **UPDATE:** El trigger se activa cuando se actualiza una fila sobre la tabla asociada.
- ✓ **DELETE:** El trigger se activa cuando se elimina una fila sobre la tabla asociada.



4.4.3 Características y usos

Los activadores proporcionan otra forma de comprobar la integridad de los datos. Es decir, son capaces de comprobar condiciones de error ante una modificación de los datos de una tabla que las restricciones básicas DDL no pueden.

Los desencadenadores pueden resultar útiles para auditar los cambios de datos en las tablas. Es decir, pueden configurarse para que, ante ciertas operaciones sobre una tabla, se deje constancia en otra tabla (tabla "log") de la operación, la hora, el usuario que la llevo a cabo, etc.

4.4.4 Acceso a los datos afectados por la instrucción que lanzó el trigger

MySQL pondrá a disposición del disparador unos identificadores especiales llamados **NEW** y **OLD**. Estos identificadores tendrán los mismos campos que la tabla afectada por la instrucción y se acceden de la siguiente manera:

NEW.nombre_campo **OLD.nombre_campo**

A continuación, describiremos cómo se comportan cada uno de ellos según la instrucción que los originó:

- ✓ **INSERT**: En una inserción, **NEW** contendrá los datos de la fila que se insertará (ejecución **BEFORE**) o ya insertada (ejecución **AFTER**). **OLD** no estará disponible.
- ✓ **DELETE**: En una inserción, **OLD** contendrá los datos de la fila que se eliminará (ejecución **BEFORE**) o ya eliminada (ejecución **AFTER**). **NEW** no estará disponible.
- ✓ **UPDATE**: Debes ver las actualizaciones como un borrado de una fila seguidos de una inserción de esa fila modificada. Por tanto, **OLD** contendrá los datos de la fila antes de la actualización y **NEW** contendrá los datos nuevos de la fila (tras el **UPDATE**).

Instrucción desencadenante	OLD	NEW
INSERT	NO	SÍ
UPDATE	SÍ	SÍ
DELETE	SÍ	NO

Atención

OLD siempre será de solo lectura. Sin embargo, si el disparador es **BEFORE** podemos asignar un valor a **NEW** diferente del que tiene.

Si modificamos el valor de algún campo de **NEW** la inserción o la actualización se realizarán con los datos modificados de **NEW**.

Ejemplo con UPDATE

```

use test;
CREATETABLElog(
  id int NOTNULL AUTO_INCREMENT,
  msg varchar(100)NOTNULL,
PRIMARYKEY(id)
);
delimiter $$
createtriggger ej1_trigger BEFOREUPDATEon producto
FOREACHROW
begin
-- Modificación del valor a insertar
SETNEW.nombre=UPPER(NEW.nombre);

-- Inserción del valor a insertar
-- y del valor antiguo en LOG
INSERTINTOlog(msg)
VALUES (CONCAT('Valor anterior: ',OLD.nombre,
              ' Valor nuevo: ',NEW.nombre));

end $$
DELIMITER ;

-- Disparo del trigger con un UPDATE
UPDATE producto SET nombre='Luis Dorado'
WHERE nombre LIKE 'Luis';

```

1

SELECT * FROM producto;

id	nombre	precio
1	Luis	NULL
NULL	NULL	NULL

SELECT * FROM test.log;

id	msg
NULL	NULL

2

UPDATE producto SET nombre='Luis Dorado' WHERE nombre LIKE 'Luis';

3

SELECT * FROM producto;

id	nombre	precio
1	LUIS DORADO	NULL

SELECT * FROM test.log;

id	msg
1	Valor anterior: Luis Valor nuevo: LUIS DORADO

4.4.5 Ejemplo guiado

Crean una **base de datos** llamada **test** que contenga una **tabla** llamada **alumnos** con las siguientes columnas.

Tabla **alumnos**:

- **id** (entero sin signo)
- **nombre** (cadena de caracteres)
- **apellido1** (cadena de caracteres)
- **apellido2** (cadena de caracteres)
- **nota** (número real)

Una vez creada la tabla escriba **dos triggers** con las siguientes características:

1. Trigger 1: **trigger_check_nota_before_insert**
 - ✓ Se ejecuta sobre la tabla **alumnos**.
 - ✓ Se ejecuta **antes** de una operación de **inserción**.
 - ✓ Si el nuevo valor de la nota que se quiere insertar es negativo, se guarda como 0.
 - ✓ Si el nuevo valor de la nota que se quiere insertar es mayor que 10, se guarda como 10.
2. Trigger 2: **trigger_check_nota_before_update**
 - ✓ Se ejecuta sobre la tabla **alumnos**.
 - ✓ Se ejecuta **antes** de una operación de **actualización**.
 - ✓ Si el nuevo valor de la nota que se quiere actualizar es negativo, se guarda como 0.
 - ✓ Si el nuevo valor de la nota que se quiere actualizar es mayor que 10, se guarda como 10.

Una vez creados los triggers escriba varias sentencias de inserción y actualización sobre la tabla **alumnos** y verifica que los *triggers* se están ejecutando correctamente.

```
-- Paso 1
DROP DATABASE IF EXISTS test;
CREATE DATABASE test;
USE test;
```

```
-- Paso2
CREATE TABLE alumnos (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(50) NOT NULL,
  apellido1 VARCHAR(50) NOT NULL,
  apellido2 VARCHAR(50), nota FLOAT
);
```

```
-- Paso 3
DELIMITER $$
DROP TRIGGER IF EXISTS trigger_check_nota_before_insert$$
CREATE TRIGGER trigger_check_nota_before_insert BEFORE INSERT
ON alumnos FOR EACH ROW
BEGIN
  IF NEW.nota < 0 THEN
    set NEW.nota = 0;
  ELSEIF NEW.nota > 10 THEN
    set NEW.nota = 10;
  END IF;
END
```

```
DELIMITER $$
DROP TRIGGER IF EXISTS trigger_check_nota_before_update$$
CREATE TRIGGER trigger_check_nota_before_update BEFORE UPDATE
ON alumnos FOR EACH ROW
BEGIN
  IF NEW.nota < 0 THEN
    set NEW.nota = 0;
  ELSEIF NEW.nota > 10 THEN
    set NEW.nota = 10;
  END IF;
END
```

```
-- Paso 4 DELIMITER ;
INSERT INTO alumnos VALUES (1, 'Pepe', 'López', 'López', -1);
INSERT INTO alumnos VALUES (2, 'María', 'Sánchez', 'Sánchez', 11);
INSERT INTO alumnos VALUES (3, 'Juan', 'Pérez', 'Pérez', 8.5);
```

```
-- Paso 5
SELECT * FROM alumnos;
```

```
-- Paso 6
UPDATE alumnos SET nota = -4 WHERE id = 1;
UPDATE alumnos SET nota = 14 WHERE id = 2;
UPDATE alumnos SET nota = 9.5 WHERE id =3;
```

```
-- Paso 7
SELECT * FROM alumnos;
```

4.4.6 Limitaciones

- ✓ El disparador no puede referirse a tablas directamente por su nombre, incluyendo la misma tabla a la que está asociado. Sin embargo, se pueden emplear las palabras clave **OLD** y **NEW**.
- ✓ El disparador no puede invocar **procedimientos almacenados** utilizando la sentencia **CALL**. (Esto significa, por ejemplo, que no se puede utilizar un procedimiento almacenado para eludir la prohibición de referirse a tablas por su nombre).
- ✓ El disparador no puede utilizar sentencias que inicien o finalicen una transacción, tal como **START TRANSACTION**, **COMMIT**, o **ROLLBACK**.
- ✓ Un disparador no puede usar un **SELECT** que muestre resultados.

4.4.7 Cancelar instrucciones o revertir transacciones

MySQL no permitelos comandos **START TRANSACTION**, **COMMIT** o **ROLLBACK** en sus disparadores.

Para ello tendrás que provocar una excepción con **SIGNAL** que cancelará la instrucción:


```

droptriggerifexists ej2_trigger;
delimiter $$
createtrigger ej2_trigger BEFOREUPDATEon producto
                                FOREACHROW
begin
  IFOLD.nombre LIKE 'Luis' THEN
    SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = '¡A Luis ni tocarlo eh!';
  ENDIF;
end $$
DELIMITER ;
UPDATE producto SET nombre='Pepe' WHERE nombre LIKE 'Luis';

```

75 09:52:02 UPDATE producto SET nombre='Pepe' WHERE nombre LIKE 'Luis' Error Code: 1644. ERROR: ¡A Luis ni tocarlo eh!

Recuerda, los disparadores forman parte de la transacción por lo que cuando lanzamos un SIGNAL, **también se revertirán las instrucciones del trigger anteriores a SIGNAL**.

Recuerda también que, si no usamos transacciones de manera explícita **cada instrucción será una instrucción autoconfirmable**. Por lo que el trigger formará parte de esa transacción.

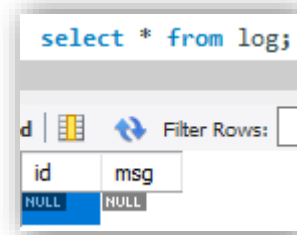
4.4.7.1 *Escribir en una tabla ANTES de lanzar el error con SIGNAL*

Si intentáis realizar una inserción (para logging por ejemplo) antes de SIGNAL (después no podríamos porque SIGNAL detendría la ejecución) no podremos porque SIGNAL también revertirá ese INSERT por formar parte de la transacción que disparó el trigger.

```

use test;
droptableifexistslog;
CREATETABLElog(
  id int NOTNULL AUTO_INCREMENT,
  msg varchar(100)NOTNULL,
PRIMARYKEY(id)
);
dropttriggerifexists ej2_trigger;
delimiter $$
createttrigger ej2_trigger BEFOREUPDATEon producto
FOREACHROW
begin
declare vMsg varchar(50);
IFOLD.nombre LIKE'Luis'THEN
set vMsg='ERROR: ¡A Luis ni tocarlo eh!';
INSERTINTOlog (msg) VALUES (vMsg);
      SIGNAL SQLSTATE'45000'SET MESSAGE_TEXT = vMsg;
ENDIF;
end $$
DELIMITER ;
UPDATE producto SET nombre='Pepe'WHERE nombre LIKE'Luis';

```



The screenshot shows a database query result for the command 'select * from log;'. The result is an empty table with two columns: 'id' and 'msg'. Both columns are marked as 'NULL' in the first row.

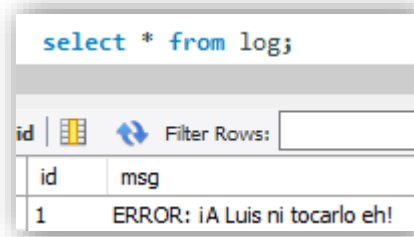
id	msg
NULL	NULL

No escribiré nada porque SIGNAL provocará una reversión de toda la transacción, incluido el INSERT. Para solucionarlo podemos usar tablas que no soportan transacciones usando el motor MyISAM.

```

use test;
droptableifexistslog;
CREATETABLElog(
    id int NOTNULL AUTO_INCREMENT,
    msg varchar(100)NOTNULL,
PRIMARYKEY(id)
)ENGINE= MyISAM;
droptriggerifexists ej2_trigger;
delimiter $$
createtriggert ej2_trigger BEFOREUPDATEon producto
                                FOREACHROW
begin
declare vMsg varchar(50);
IFOLD.nombre LIKE 'Luis' THEN
set vMsg='ERROR: ¡A Luis ni tocarlo eh!';
INSERTINTOlog(msg)VALUES(vMsg);
SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = vMsg;
ENDIF;
end $$
DELIMITER ;
UPDATE producto SET nombre='Pepe' WHERE nombre LIKE 'Luis';

```



select * from log;	
id	msg
1	ERROR: ¡A Luis ni tocarlo eh!

4.5 Ampliación: Transacciones

4.5.1 Desactivar AUTOCOMMIT para uso obligado transacciones

Algunos Sistemas Gestores de Bases de Datos, como MySQL (si trabajamos con el motor **InnoDB**) tienen activada por defecto la variable **AUTOCOMMIT**. Esto quiere decir que **automáticamente se aceptan todos los cambios realizados después de la ejecución de una sentencia SQL y no es posible deshacerlos**.

Aunque la variable **AUTOCOMMIT** está activada por defecto al inicio de una sesión SQL, podemos configurarlo para indicar si queremos trabajar con transacciones implícitas o explícitas.

Podemos consultar el valor actual de AUTOCOMMIT haciendo:

```
SELECT @@AUTOCOMMIT;
```

Para desactivar la variable AUTOCOMMIT hacemos:

```
SET AUTOCOMMIT =0;
```

Si hacemos esto siempre tendríamos una transacción abierta y los cambios sólo se aplicarían en la base de datos ejecutando la sentencia COMMIT de forma explícita.

Para activar la variable AUTOCOMMIT hacemos:

```
SET AUTOCOMMIT =1;
```

Para poder trabajar con transacciones en MySQL es necesario utilizar **InnoDB**.

Modificar las preferencias de MySQL Workbench

Podemos desactivarlo en el menú Query:

