

ANEXO: PROGRAMACIÓN MYSQL

Bases de Datos
CFGs DAW

Raquel Torres
raquel.torres@ceedcv.es
Versión:180508.1925

Licencia



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

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:



Importante



Atención



Interesante

Revisiones

ÍNDICE DE CONTENIDO

1.Lenguaje de programación en MySQL.....	4
1.1 Procedimientos y funciones.....	4
1.2 A tener en cuenta antes de comenzar.....	5
1.3 Operadores.....	7
1.3.1 Operadores matemáticos.....	7
1.3.2 Operadores de relación.....	7
1.3.3 Operadores lógicos.....	7
1.3.4 Operador de concatenación.....	7
1.4 Estructuras de control.....	8
1.4.1 Alternativas. IF.....	8
1.4.2 Alternativa múltiple. CASE.....	8
1.4.3 Bucle WHILE.....	9
1.4.4 Bucle REPEAT.....	9
1.5 Ejemplos.....	9
1.5.1 Ejemplo 1.....	9
1.5.2 Ejemplo 2.....	10
1.5.3 Ejemplo 3.....	11
1.5.4 Ejemplo 4.....	12
1.5.5 Ejemplo 5.....	14
1.5.6 Ejemplo 6.....	15
1.5.7 Ejemplo 7.....	16
2.Cursores.....	17
2.1 Cursores implícitos.....	17
2.1.1 Ejemplo 8.....	17
2.1.2 Ejemplo 9.....	18
2.2 Cursores explícitos.....	19
2.2.1 Declaración de cursores.....	19
2.2.2 Apertura de un cursor.....	19
2.2.3 Recorrido del cursor.....	20
2.2.4 Cerrando un cursor.....	20
2.2.5 Ejemplo 10.....	21
2.2.6 Ejemplo 11.....	23
2.2.7 Ejemplo 11a.....	26
3.Triggers.....	27

ANEXO: PROGRAMACIÓN MYSQL

1. LENGUAJE DE PROGRAMACIÓN EN MYSQL

1.1 Procedimientos y funciones

Aunque el lenguaje de programación por excelencia para bases de datos es PL/SQL, otras bases de datos como MySQL también permiten crear procedimientos y funciones. Una vez visto cómo trabajar en Oracle, vamos a indicar la sintaxis necesaria para realizar las mismas acciones en MySQL.

MySQL sigue la sintaxis SQL:2003 para procedimientos almacenados que también utiliza IBM DB2.

En MySQL disponemos de las instrucciones *CREATE PROCEDURE* y *CREATE FUNCTION* que nos permitirán crear procedimientos y funciones.

La sintaxis que utilizaremos para los procedimientos será:

```
CREATE PROCEDURE nombre_procedimiento ([[IN|OUT| INOUT]
nombre_parametro tipo_parametro,...])
BEGIN
    ... instrucciones;
END
```

La sintaxis para las funciones será:

```
CREATE FUNCTION nombre_funcion ([[IN|OUT| INOUT] nombre_parametro
tipo_parametro,...])
RETURNS tipo_de_dato
BEGIN
    ... instrucciones;
END
```

Antes de comenzar a crear procedimientos vamos a ver algunas consideraciones importantes.

1.2 A tener en cuenta antes de comenzar...

- Debemos establecer el delimitador de comienzo y fin del procedimiento, normalmente es el \$, \$\$ o ·.
- Para mostrar un mensaje por pantalla se utilizará la instrucción *SELECT*. Por ejemplo *SELECT 'Hola';* mostrará “Hola” en pantalla.
- Cada instrucción terminará con punto y coma (;) como en Oracle.
- Los bloques también comienzan con *BEGIN* y terminan con *END*.
- Las funciones, al igual que en PL/SQL, devolverán un valor y deben incluir una instrucción *RETURN* que devuelva el resultado de la función.
- Los parámetros de los procedimientos pueden ser de entrada (*IN*, valor por defecto, si no se indica ninguno), de salida (*OUT*) y de entrada/salida (*INOUT*). Pero ATENCIÓN: No se pueden utilizar estos especificadores para las funciones (a diferencia de PL/SQL), luego en las funciones siempre son parámetros de entrada.
- Podemos incluir comentarios comenzando la línea con #, con -- desde donde aparezcan hasta el final de la línea y /* texto */ para un grupo de líneas.
- Ejecutaremos un procedimiento desde la línea de comandos con el comando *CALL* seguido del nombre del procedimiento y entre paréntesis los parámetros que necesite. Para ejecutar una función bastará con llamarla dentro de una instrucción *SELECT*.
- Para eliminar un procedimiento que hemos creado emplearemos *DROP PROCEDURE nombre_procedimiento* y para eliminar una función *DROP FUNCTION nombre_función*.
- Para ver el código de un procedimiento usaremos *SHOW CREATE PROCEDURE nombre_procedimiento* y para ver el código de una función *SHOW CREATE FUNCTION nombre_función*.
- La sentencia *DECLARE* irá dentro del bloque *BEGIN - END*, al comienzo, y primero deben declararse las variables, después los cursores y por último los handlers (manejadores de excepciones). Cada declaración llevará su instrucción *DECLARE* correspondiente (no como en PL/SQL que con un declare se podían declarar todas las variables y cursores que necesitásemos). Podemos declarar varias variables en una misma línea siempre que sean del mismo tipo. La sintaxis de esta sentencia es:

```
DECLARE nombre_var tipo_variable [DEFAULT valor]
```

Si no ponemos un valor por defecto la variable tomará el valor NULL.

- La declaración de los cursores, que se realiza después de las variables, tendrá la siguiente sintaxis:

```
DECLARE nombre_cursor CURSOR FOR instruccion_select_del_cursor;
```

- Para abrir un cursor emplearemos la instrucción *OPEN* seguida del nombre del cursor.

```
OPEN nombre_cursor;
```

- Para obtener la siguiente fila de datos del cursor utilizaremos *FETCH* con la siguiente sintaxis:

```
FETCH nombre_cursor INTO variable_col1, variable_col2,...;
```

- Para comprobar si ya no hay datos, podremos hacerlo con un manejador mediante el *SQLSTATE 02000* o bien con una condición *NOT FOUND* (lo veremos más adelante)
- Para cerrar un cursor utilizaremos la instrucción *CLOSE* seguida del nombre del cursor.

```
CLOSE nombre_cursor;
```

1.3 Operadores

1.3.1 Operadores matemáticos

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

1.3.2 Operadores de relación

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

1.3.3 Operadores lógicos

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

1.3.4 Operador de concatenación

Operador de concatenación MySQL.	
CONCAT(VA1OR1,VA1OR2,VA1OR3,...)	Concatena los valores en una cadena.

1.4 Estructuras de control

1.4.1 Alternativas. IF

La sintaxis del IF es como en PL/SQL:

```

IF condicion THEN
    instrucciones...
[ELSEIF condicion then
    instrucciones...]
[ELSE
    instrucciones...]

```

```
END IF
```

1.4.2 Alternativa múltiple. CASE

La sintaxis del CASE es como en PL/SQL:

```
CASE [expresion]
WHEN {condicion1|valor1} THEN
    bloque_instrucciones_1
WHEN {condicion2|valor2} THEN
    bloque_instrucciones_2
...
[ELSE
    bloque_instrucciones_por_defecto]
END CASE;
```

Esta instrucción es muy versátil, más que los CASE de otros lenguajes de programación. Esta instrucción puede comparar el valor de una variable o el resultado de una expresión con los valores que siguen a la palabra reservada *WHEN*, o bien puede omitir ese valor o expresión inicial y plantear condiciones independientes en cada uno de los *WHEN*.

1.4.3 Bucle WHILE

El bucle *WHILE* realizará las instrucciones que contiene, mientras la condición sea verdadera. Su sintaxis es:

```
WHILE condicion DO
    instrucciones...
END WHILE;
```

1.4.4 Bucle REPEAT

El bucle *REPEAT* realizará las instrucciones que contiene, hasta que la condición sea verdadera. Su sintaxis es:


```
REPEAT
    instrucciones ...
UNTIL condicion
END REPEAT;
```

1.5 Ejemplos

1.5.1 Ejemplo 1

Realizar una función que sume dos números reales y devuelva el resultado.

```
DROP FUNCTION IF EXISTS SUMA_DOS_NUMEROS;
DELIMITER $$
CREATE FUNCTION SUMA_DOS_NUMEROS (N1 FLOAT, N2 FLOAT)
RETURNS FLOAT
BEGIN
    DECLARE SUMA FLOAT DEFAULT 0;
    SET SUMA = N1 + N2;
    RETURN SUMA;
END$$
DELIMITER;
```

Veamos algunas diferencias con PL/SQL a tener en cuenta:

Aquí no disponemos el *OR REPLACE FUNCTION*, con lo cual primero comprobamos si existe una función con el mismo nombre y si existe la borramos. Hay que tener en cuenta que la primera vez que ejecutemos el script, como la función no existe nos mostrará un warning (aviso).

Tenemos que indicar los delimitadores que utilizaremos para indicar donde termina la función o el procedimiento, tal como dijimos los más habituales son \$\$, \$, //. Aquí hemos elegido \$\$.

Hay que cambiar el delimitador porque las instrucciones de las funciones y procedimientos deben terminar con punto y coma, con lo cual MySQL no sabría identificar cuando acaba la función o el procedimiento. Para cambiar el delimitador se emplea la palabra reservada *DELIMITER*. Además, al final, cuando haya terminado el script volveremos a colocar como delimitador el punto y coma para volver a la normalidad.

Para ejecutar la función emplearemos una instrucción *SELECT* con el nombre de la función seguida de un paréntesis con los valores de los parámetros que deseemos utilizar.

La ejecución será:

```
mysql> source c:\src\mysql_fp01.sql
Query OK, 0 rows affected, 1 warning (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

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

1.5.2 Ejemplo 2

Vamos a realizar el mismo ejemplo pero con un procedimiento. En este caso el procedimiento se llamará *SUMA_PRO_DOS_NUMEROS*.

```
DROP PROCEDURE IF EXISTS SUMA_PRO_DOS_NUMEROS;
DELIMITER $$
CREATE PROCEDURE SUMA_PRO_DOS_NUMEROS (IN N1 FLOAT, IN N2 FLOAT)
BEGIN
    DECLARE SUMA FLOAT DEFAULT 0;
    SET SUMA = N1 + N2;
    SELECT SUMA;
END$$
DELIMITER ;
```

Tampoco tenemos el *OR REPLACE PROCEDURE*, con lo cual primero comprobamos si existe un procedimiento con el mismo nombre y si existe lo borramos.

Igual que antes, establecemos el delimitador que vamos a utilizar y escribimos nuestro procedimiento, ahora en los parámetros si podemos indicar si son *IN*, *OUT* o *INOUT*. Para mostrar el resultado de la variable en pantalla utilizamos *SELECT*.

Para llamar al procedimiento utilizaremos, desde la línea de comandos, *CALL* seguido del nombre del procedimiento y los valores de los parámetros entre paréntesis.

Veamos la ejecución.

```
mysql> source c:\src\mysql_fp02.sql
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> CALL SUMA_PRO_DOS_NUMEROS(3,5);
+-----+
| SUMA |
+-----+
|    8 |
+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

1.5.3 Ejemplo 3

Realizar un procedimiento que recibe un número y nos dice si es par o impar.

```
DROP PROCEDURE IF EXISTS PAR_IMPAR;
DELIMITER $$
CREATE PROCEDURE PAR_IMPAR (IN N INT)
BEGIN
    DECLARE COMOES VARCHAR(10);
    IF MOD(N,2) = 0 THEN
        SET COMOES='PAR';
    ELSE
        SET COMOES='IMPAR';
    END IF;
    SELECT COMOES AS 'PAR O IMPAR';
END$$
DELIMITER ;
```

El resultado será:

```
mysql> source c:\src\mysql_fp03.sql
Query OK, 0 rows affected, 1 warning (0.00 sec)

Query OK, 0 rows affected (0.03 sec)

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

Query OK, 0 rows affected (0.00 sec)

mysql> CALL PAR_IMPAR(4);
+-----+
| PAR O IMPAR |
+-----+
| PAR         |
+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

1.5.4 Ejemplo 4

Realizar un procedimiento que recibe una nota numérica entre 0 y 10 y nos dice si es un suspenso, suficiente, bien, etc.

```
DROP PROCEDURE IF EXISTS NOTA_TEXTO;
DELIMITER $$
CREATE PROCEDURE NOTA_TEXTO (IN N INT)
    BEGIN
        DECLARE NOTA VARCHAR(15);
        CASE N
            WHEN 0 THEN
                SET NOTA = 'Insuficiente';
            WHEN 1 THEN
                SET NOTA = 'Insuficiente';
            WHEN 2 THEN
                SET NOTA = 'Insuficiente';
            WHEN 3 THEN
                SET NOTA = 'Insuficiente';
            WHEN 4 THEN
                SET NOTA = 'Insuficiente';
            WHEN 5 THEN
                SET NOTA = 'Suficiente';
```

```
        WHEN 6 THEN
            SET NOTA = 'Bien';
        WHEN 7 THEN
            SET NOTA = 'Notable';
        WHEN 8 THEN
            SET NOTA = 'Notable';
        WHEN 9 THEN
            SET NOTA = 'Sobresaliente';
        WHEN 10 THEN
            SET NOTA = 'Sobresaliente';
        ELSE
            SET NOTA = 'No Válida';

    END CASE;
    SELECT NOTA;

END$$
DELIMITER ;
```

El resultado será:

```
mysql> source c:\src\mysql_fp04.sql
Query OK, 0 rows affected (0.02 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> call nota_texto(4);
+-----+
| NOTA  |
+-----+
| Insuficiente |
+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> call nota_texto(8);
+-----+
| NOTA  |
+-----+
| Notable |
+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

Pero y si la nota en lugar de ser un valor entero fuese un valor real, ¿cómo lo haríamos? Muy sencillo, utilizaríamos el CASE con condiciones. Veamos cómo hacerlo en el siguiente ejercicio:

1.5.5 Ejemplo 5

El mismo que el anterior pero con la nota como un número real.

```
DROP PROCEDURE IF EXISTS NOTA_TEXTO;
DELIMITER $$
CREATE PROCEDURE NOTA_TEXTO (IN N FLOAT)
BEGIN
    DECLARE NOTA VARCHAR(15);
    CASE
        WHEN N < 5 THEN
            SET NOTA = 'Insuficiente';
        WHEN N < 6 THEN
            SET NOTA = 'Suficiente';
        WHEN N < 7 THEN
            SET NOTA = 'Bien';
        WHEN N < 9 THEN
            SET NOTA = 'Notable';
        WHEN N <= 10 THEN
            SET NOTA = 'Sobresaliente';
        ELSE
            SET NOTA = 'No Válida';
    END CASE;
    SELECT NOTA;
END$$
DELIMITER ;
```

El resultado será:

```
Query OK, 0 rows affected (0.00 sec)
mysql> source c:\src\mysql_fp05.sql
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
mysql> CALL NOTA_TEXTO(6.5);
+-----+
| NOTA |
+-----+
| Bien |
+-----+
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
mysql> CALL NOTA_TEXTO(7.5);
+-----+
| NOTA |
+-----+
| Notable |
+-----+
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
mysql> CALL NOTA_TEXTO(9.5);
+-----+
| NOTA |
+-----+
| Sobresaliente |
+-----+
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.02 sec)
```

1.5.6 Ejemplo 6

Realizar un procedimiento que recibe un número entero y muestra en pantalla los números desde el 1 hasta el numero recibido incluido. Utilizar un bucle *WHILE* para este ejercicio.

```
DROP PROCEDURE IF EXISTS DEL_1_AL_N;
DELIMITER $$
CREATE PROCEDURE DEL_1_AL_N (N INT)
BEGIN
    DECLARE I INT DEFAULT 0;
    WHILE I <= N DO
        SET I = I+1;
        SELECT I;
    END WHILE;
END$$
DELIMITER ;
```

El resultado será:

```
mysql> source c:\src\mysql_fp06.sql
Query OK, 0 rows affected (0.01 sec)

Query OK, 0 rows affected (0.00 sec)

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

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

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

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

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

1.5.7 Ejemplo 7

Ahora vamos a realizar un procedimiento que recibe un parámetro de tipo entero y muestra en pantalla los números impares entre el 1 y el número recibido. Para este ejercicio utilizaremos un bucle *REPEAT*.

```
DROP PROCEDURE IF EXISTS IMPARES_DEL_1_AL_N;
DELIMITER $$
CREATE PROCEDURE IMPARES_DEL_1_AL_N (N INT)
BEGIN
    DECLARE I INT DEFAULT 1;
    IF N >= 1 THEN
        REPEAT
            SELECT I;
            SET I = I+2;
        UNTIL I > N
        END REPEAT;
    END IF;
END$$
DELIMITER ;
```

El resultado será:

```
mysql> source c:\src\mysql_fp07.sql
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
mysql> CALL IMPARES_DEL_1_AL_N(6);
+-----+
| I |
+-----+
| 1 |
+-----+
1 row in set (0.00 sec)

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

+-----+
| I |
+-----+
| 5 |
+-----+
1 row in set (0.01 sec)
Query OK, 0 rows affected (0.01 sec)
```


2. CURSORES

En MySQL, al igual que en Oracle, podemos distinguir consultas que sólo devuelven una fila (cursores implícitos) y consultas que devolverán ninguna, una o más filas (cursores explícitos). Veamos cómo podemos trabajar con ellos en MySQL.

Cuando las consultas sólo pueden devolver una sola línea (cursor implícito) realizaremos la misma acción que en Oracle, guardando el resultado en variables con la palabra reservada **INTO** en la consulta. Igual que en Oracle, la asignación de los datos será posicional (por el lugar que ocupan) y será responsabilidad del programador colocar las variables en el orden adecuado y con el tamaño y el tipo adecuado a los datos que van a recibir.

2.1 Cursores implícitos

2.1.1 Ejemplo 8

Vamos a realizar un procedimiento que recibirá como parámetro una GAMA y nos mostrará el precio medio de venta de los productos que pertenecen a dicha gama.

```
DROP PROCEDURE IF EXISTS Precio_Medio_Gama;
DELIMITER $$
CREATE PROCEDURE Precio_Medio_Gama (IN La_Gama VARCHAR(50))
BEGIN
    DECLARE PM FLOAT DEFAULT 0;
    SELECT AVG(precioventa) INTO PM from productos where Gama = La_Gama;
    SELECT PM as 'Precio Medio';
END$$
DELIMITER ;
```

Fíjate en los parámetros de entrada del procedimiento, La_Gama está declarado como VARCHAR(50) ¿Es igual que en Oracle?.

El resultado será:

```
mysql> source c:\src\mysql_fp08.sql
Query OK, 0 rows affected, 1 warning (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
mysql> call Precio_Medio_Gama('Herramientas');
+-----+
| Precio Medio |
+-----+
|          13 |
+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> call Precio_Medio_Gama('Frutales');
+-----+
| Precio Medio |
+-----+
|    23.9259 |
+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.02 sec)
```

2.1.2 Ejemplo 9

Ahora vamos a realizar un ejemplo que recibe como parámetro el código de un cliente y muestra en pantalla su nombre, su teléfono y el límite de crédito que tiene asignado.

```
DROP PROCEDURE IF EXISTS Datos_Cliente;
DELIMITER $$
CREATE PROCEDURE Datos_Cliente (IN CodCli INT)
BEGIN
    DECLARE NOMBRE VARCHAR(50);
    DECLARE TEL VARCHAR(15);
    DECLARE LIMITE FLOAT DEFAULT 0;
    SELECT NOMBRECLIENTE, TELEFONO, LIMITECREDITO
    INTO NOMBRE, TEL, LIMITE
    FROM CLIENTES where CODIGOCLIENTE = CodCli;
    SELECT NOMBRE, TEL, LIMITE;
END$$
DELIMITER ;
```

El resultado será:

```
mysql> source c:\src\mysql_fp09.sql
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
mysql> call Datos_cliente(20);
+-----+-----+-----+
| NOMBRE                | TEL          | LIMITE |
+-----+-----+-----+
| AYMERICH GOLF MANAGEMENT, SL | 964493072    | 20000  |
+-----+-----+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.02 sec)

mysql> call Datos_cliente(10);
+-----+-----+-----+
| NOMBRE                | TEL          | LIMITE |
+-----+-----+-----+
| DaraDistribuciones    | 675598001    | 50000  |
+-----+-----+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

2.2 Cursores explícitos

Cuando el resultado de una consulta esté formado por varias filas (cursores explícitos) tendremos que operar con ellas utilizando los cursores. El cursor debe ser declarado, abierto, utilizado y finalmente cerrado.

Veamos cómo realizar todas estas operaciones.

2.2.1 Declaración de cursores

La declaración de los cursores que utilizemos en el procedimiento debe realizarse después de la declaración de las variables.

La declaración del cursor también se realiza en la instrucción *DECLARE* siguiendo la sintaxis:

```
DECLARE nombre_cursor CURSOR FOR instrucción_select_del_cursor;
```

Por ejemplo:

```
DECLARE herramientas CURSOR FOR nombre, precioventa FROM PRODUCTOS WHERE
gama='herramientas';
```

2.2.2 Apertura de un cursor

Los cursores se abrirán con la palabra reservada *OPEN* seguida del nombre del cursor. Por ejemplo:

```
OPEN herramientas;
```

2.2.3 Recorrido del cursor

Para obtener cada una de las filas que forman el resultado de la consulta debemos ir pidiéndolas, una a una, con un *FETCH*. La sintaxis de esta instrucción es:

```
FETCH nombre_cursor INTO var1, var2, var3,...
```

Por ejemplo:

```
FECHT herramientas INTO Nombre_Producto, Precio;
```

(Nombre_producto y Precio estarán declaradas previamente).

Cada vez que hacemos un *FETCH* obtenemos una línea, luego para conseguir todas las líneas del cursor deberemos emplear un bucle (*WHILE* o *REPEAT*).

2.2.4 Cerrando un cursor

Una vez que hemos recorrido el cursor debemos cerrarlo para liberar los recursos que está consumiendo. Para ello emplearemos la instrucción *CLOSE* seguida del nombre del cursor, por ejemplo:

```
CLOSE herramientas;
```

Luego, para recorrer un cursor cuyo resultado son varias líneas deberemos hacer lo siguiente (ejemplo incompleto):

```
DECLARE Nombre_Producto VARCHAR(70);
DECLARE Precio FLOAT DEFAULT 0;
DECLARE herramientas CURSOR FOR SELECT nombre, precioventa FROM PRODUCTOS WHERE
gama='herramientas';
BEGIN
    OPEN herramientas;
    WHILE condicion DO
        FECHT herramientas INTO Nombre_Producto, Precio;
    END WHILE;
    CLOSE herramientas;
END
```

Parece que tenemos un pequeño problemilla, ¿que condición ponemos para saber que hemos terminado de procesar las filas que nos suministra el cursor?

En Oracle contábamos con los *ATRIBUTOS*, pero MySQL no los tiene. MySQL soluciona este problema mediante los manejadores (*handlers*). Podemos definir un manejador para cada situación que pueda darse al procesar la información.

Ahora veremos cómo gestionar qué ocurre cuando terminamos de procesar todos los registros de un cursor para completar el ejemplo. Más adelante veremos cómo gestionar otras situaciones.

Los manejadores se deben declarar después de los cursores. La sintaxis de un manejador es la siguiente:

```
DECLARE tipo_manejador HANDLER FOR condicion instruccion;
```

El tipo_manejador puede ser **CONTINUE** que continúa la ejecución normal después de ejecutar la instrucción del manejador y **EXIT** que termina la ejecución del comando *BEGIN ... END*.

Las condiciones pueden ser controladas por el código que devuelve MySQL con **SQLSTATE** o bien por las abreviaciones **SQLWARNING** que es una abreviación para todos los códigos **SQLSTATE** que comienzan con 01, **NOT FOUND** es una abreviación para todos los códigos **SQLSTATE** que comienzan con 02 y **SQLEXCEPTION** es una abreviación para todos los códigos **SQLSTATE** no tratados por **SQLWARNING** o **NOT FOUND**.

La instrucción suele ser un cambio de valor en una variable que actuará como un switch que nos permitirá determinar qué hacer cuando cambie de valor.

En nuestro caso, el *Handler* que vamos a utilizar más frecuentemente es:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET TERMINADO = 1;
```

Lo que hace este handler es que cuando cuando se solicita una fila y ya no hay (**NOT FOUND**) se colocará la variable *TERMINADO* a 1.

2.2.5 Ejemplo 10

Crear un procedimiento que reciba el código de un pedido y nos indique el número total de productos que los forman. (Aunque para resolver este ejercicio no son necesarios los cursores, los vamos a utilizar a modo de ejemplo).

```
DROP PROCEDURE IF EXISTS Numero_Productos;
DELIMITER $$
CREATE PROCEDURE Numero_Productos (IN CodPed INT)
BEGIN
    DECLARE NUMEROPROD INT DEFAULT 0;
    DECLARE NUMEROTOTAL INT DEFAULT 0;
    DECLARE TERMINADO INT DEFAULT 0;
    DECLARE CANTIDADES CURSOR FOR SELECT CANTIDAD
        FROM DETALLEPEDIDOS
        WHERE CODIGOPEDIDO = CodPed;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET TERMINADO = 1;
    OPEN CANTIDADES;
    WHILE TERMINADO = 0 DO
        FETCH CANTIDADES INTO NUMEROPROD;
        IF TERMINADO = 0 THEN
            SET NUMEROTOTAL = NUMEROTOTAL + NUMEROPROD;
        END IF;
    END WHILE;
    CLOSE CANTIDADES;
    SELECT NUMEROTOTAL AS CANTIDAD_TOTAL;
END$$
DELIMITER ;
```

Lo que hacemos es declarar una variable *TERMINADO* con un valor inicial de 0 que cambiará a 1 cuando se acaben de procesar las filas del cursor (por el *HANDLER* con *NOT FOUND*). De esta forma estaremos en el bucle *WHILE* hasta que no haya más filas que procesar. (Cuidado, sólo acumularemos la cantidad cuando no hayamos llegado al final, por eso colocamos un *IF TERMINADO = 0 THEN*, si no lo hiciésemos estaríamos sumando dos veces el último valor !!!)

El resultado será:

```
mysql> SOURCE C:\SRC\MYSQL_FP10.SQL
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
mysql> call Numero_productos(10);
+-----+
| CANTIDAD_TOTAL |
+-----+
|          40    |
+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

mysql> call Numero_productos(15);
+-----+
| CANTIDAD_TOTAL |
+-----+
|          21    |
+-----+
1 row in set (0.00 sec)

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

Prueba a repetir el ejercicio pero con un bucle *REPEAT*. ¿Qué bucle crees que es más sencillo de utilizar para recorrer un cursor?

2.2.6 Ejemplo 11

Se trata de realizar un procedimiento que recibe el código de un cliente y nos muestra su estado, es decir, el nombre del cliente, el importe total de los pedidos realizados, los pagos que ha realizado y el importe que tiene pendiente, así como si ha superado o no el crédito que tiene indicado en su ficha de cliente.

```
DROP PROCEDURE IF EXISTS Estado_Clientea;
DELIMITER $$
CREATE PROCEDURE Estado_Clientea(IN CodCli INT)
BEGIN
    DECLARE IMPORTETOTAL FLOAT DEFAULT 0;
    DECLARE IMPORTE FLOAT DEFAULT 0;
    DECLARE NPEDIDO INT DEFAULT 0;
    DECLARE PAGOTOTAL FLOAT DEFAULT 0;
    DECLARE UNPAGO FLOAT DEFAULT 0;
    DECLARE LIMITE FLOAT DEFAULT 0;
    DECLARE SITUACION VARCHAR(100);
    DECLARE NOMBRE VARCHAR(50);
    DECLARE TERMINADO INT DEFAULT 0;
    DECLARE DATOSCLIENTES CURSOR FOR SELECT
```

```
NOMBRECLIENTE,LIMITECREDITO
FROM CLIENTES
WHERE CODIGOCLIENTE = CodCli;

DECLARE DATOSPEDIDO CURSOR FOR SELECT P.CODIGOPEDIDO,
SUM(D.CANTIDAD * D.PRECIOUNIDAD)
FROM DETALLEPEDIDOS D, PEDIDOS P
WHERE D.CODIGOPEDIDO = P.CODIGOPEDIDO AND
P.CODIGOCLIENTE = CodCli
GROUP BY P.CODIGOPEDIDO
ORDER BY P.CODIGOPEDIDO;

DECLARE DATOSPAGOS CURSOR FOR SELECT CANTIDAD
FROM PAGOS
WHERE CODIGOCLIENTE = CodCli;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET TERMINADO = 1;

--PROCESAMOS EL CLIENTE
OPEN DATOSCLIENTES;
FETCH DATOSCLIENTES INTO NOMBRE, LIMITE;
CLOSE DATOSCLIENTES;

--PROCESANDO LOS PEDIDOS DEL CLIENTE
SET TERMINADO = 0;
OPEN DATOSPEDIDO;
WHILE TERMINADO = 0 DO
    FETCH DATOSPEDIDO INTO NPEDIDO, IMPORTE;
    IF TERMINADO = 0 THEN
        SET IMPORTETOTAL = IMPORTETOTAL + IMPORTE;
    END IF;
END WHILE;
CLOSE DATOSPEDIDO;

--PROCESANDO LOS PAGOS
SET TERMINADO = 0;
OPEN DATOSPAGOS;
WHILE TERMINADO = 0 DO
    FETCH DATOSPAGOS INTO UNPAGO;
    IF TERMINADO = 0 THEN
```



```
        SET PAGOTOTAL = PAGOTOTAL + UNPAGO;
    END IF;
END WHILE;

--CALCULAMOS LA SITUACION
IF IMPORTETOTAL > PAGOTOTAL THEN
    IF (IMPORTETOTAL - PAGOTOTAL) > LIMITE THEN
        SET SITUACION = CONCAT('EL LIMITE ES: ',LIMITE,' SE HA
        SOBREPASADO EN: ',IMPORTETOTAL - PAGOTOTAL - LIMITE);
    ELSE
        SET SITUACION = CONCAT('QUEDA PENDIENTE: ',IMPORTETOTAL
        - PAGOTOTAL,' NO SE HA EXCEDIDO.');
```

Este ejercicio no es necesario hacerlo con cursores, pero se ha realizado así por motivos didácticos.

Me gustaría que te fijases en varios aspectos: se definen tres cursores *DATOSCLIENTES*, *DATOSPEDIDO* y *DATOSPAGOS* y el mismo manejador es utilizado para comprobar si se han terminado de procesar las líneas del cursor.

Además en *DATOSCLIENTES* no hemos realizado un bucle, pues como sólo es una línea, con un *FETCH* ya obtenemos los datos y no necesitamos hacer un bucle (no hubiese sido necesario crear un cursor para los datos de los clientes).

Antes de procesar *DATOSPEDIDO* ponemos la variable *TERMINADO* a 0 por si hubiese sido utilizada en otro cursor y ya hubiese tomado valor 1. Lo mismo haremos antes de procesar *DATOSPAGOS*.

Además en cada uno de los bucles *WHILE* pondremos un *IF* para comprobar que no hemos llegado al final del cursor y hay que acumular las variables.

El resultado será:

```
mysql> SOURCE C:\SRC\MYSQL_FP11.SQL
Query OK, 0 rows affected (0.02 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
mysql> CALL ESTADO_CLIENTEA(1);
+-----+-----+-----+-----+
| NOMBRE          | IMPORTETOTAL | PAGOTOTAL | SITUACION |
+-----+-----+-----+-----+
| DGPRODUCTIONS  | 6165         | 4000      | QUEDA PENDIENTE: 2165 NO SE HA EXCEDIDO. |
+-----+-----+-----+-----+
1 row in set (0.02 sec)

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

Prueba a crear el mismo ejercicio pero utilizando el bucle *REPEAT*.

Veamos cómo se haría el mismo ejercicio sin cursores:

2.2.7 Ejemplo 11a

```
DROP PROCEDURE IF EXISTS Estado_Clienteb;
DELIMITER $$
CREATE PROCEDURE Estado_Clienteb(IN CodCli INT)
BEGIN
    DECLARE IMPORTETOTAL FLOAT DEFAULT 0;
    DECLARE PAGOTOTAL FLOAT DEFAULT 0;
    DECLARE LIMITE FLOAT DEFAULT 0;
    DECLARE SITUACION VARCHAR(100);
    DECLARE NOMBRE VARCHAR(50);
    SELECT NOMBRECLIENTE,LIMITECREDITO INTO NOMBRE, LIMITE
        FROM CLIENTES
        WHERE CODIGOCLIENTE = CodCli;
    SELECT SUM(CANTIDAD * PRECIOUNIDAD) INTO IMPORTETOTAL
        FROM DETALLEPEDIDOS D, PEDIDOS P
        WHERE D.CODIGOPEDIDO = P.CODIGOPEDIDO AND
        P.CODIGOCLIENTE = CodCli;
    SELECT SUM(CANTIDAD) INTO PAGOTOTAL
        FROM PAGOS
        WHERE CODIGOCLIENTE = CodCli;
    IF IMPORTETOTAL > PAGOTOTAL THEN
```

```

        IF (IMPORTETOTAL - PAGOTOTAL) > LIMITE THEN
            SET SITUACION = CONCAT('EL LIMITE ES: ',LIMITE,' SE HA
            SOBREPASADO EN: ',IMPORTETOTAL - PAGOTOTAL - LIMITE);
        ELSE
            SET SITUACION = CONCAT('QUEDA PENDIENTE: ',IMPORTETOTAL -
            PAGOTOTAL,' NO SE HA EXCEDIDO.');
```

```

        END IF;
    ELSE
        SET SITUACION = CONCAT('TODO PAGADO. LIMITE DE CREDITO: ',LIMITE);
    END IF;
    SELECT NOMBRE, IMPORTETOTAL, PAGOTOTAL, SITUACION;
END$$
DELIMITER ;

```

El resultado será:

```

mysql> SOURCE C:\SRC\MYSQL_FP11a.SQL
Query OK, 0 rows affected (0.02 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
mysql> CALL ESTADO_CLIENTEB(1);
+-----+-----+-----+-----+
| NOMBRE          | IMPORTETOTAL | PAGOTOTAL | SITUACION          |
+-----+-----+-----+-----+
| DGPRODUCTIONS  | 6165         | 4000      | QUEDA PENDIENTE: 2165 NO SE HA EXCEDIDO. |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.03 sec)

```

Como puedes observar el resultado es el mismo.

3. TRIGGERS

MySQL también cuenta con *Triggers* y son muy parecidos a los que hemos visto en ORACLE. Igual que antes, un disparador es un bloque de código que se ejecuta cuando en una tabla ocurre un evento de tipo *INSERT*, *DELETE* o *UPDATE*.

La sintaxis de los triggers en MySQL es:

```
CREATE TRIGGER nombre_trigger {BEFORE | AFTER} {INSERT | UPDATE | DELETE}
```

```

ON nombre_tabla
FOR EACH ROW
BEGIN
INSTRUCCIONES ...;
END

```

Como puedes apreciar, las diferencias con todo lo visto en Oracle son mínimas, lo único es que en el código no hay que poner el signo dos puntos (:) antes de *NEW* y *OLD* para referirnos a los campos que forman parte de la instrucción.

Veamos el ejemplo que hemos creado en Oracle para controlar el número de empleados de cada departamento, pero ahora en MySQL:

Primero añadimos el campo *NUMEMP* de tipo entero con valor por defecto 0.

```

mysql> alter table departamentos add column numemp int default 0;
Query OK, 4 rows affected (0.52 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> select * from departamentos;
+-----+-----+-----+-----+
| CodDpto | Nombre           | Ubicacion           | numemp |
+-----+-----+-----+-----+
| ADM     | Administración   | Planta quinta U2    | 0      |
| ALM     | Almacén          | Planta baja U1      | 0      |
| CONT    | Contabilidad     | Planta quinta U1    | 0      |
| IT      | Informática      | Planta sótano U3    | 0      |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

Actualizamos los datos del nuevo campo.

```

mysql> update departamentos set numemp=3 where coddpto='IT';
Query OK, 1 row affected (0.06 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> update departamentos set numemp=1 where coddpto='CONT';
Query OK, 1 row affected (0.08 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> update departamentos set numemp=1 where coddpto='ALM';
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

```

Y creamos el *trigger*:

```

DELIMITER $$
CREATE TRIGGER nuevo_empleado
AFTER INSERT
ON EMPLEADOS

```

```

FOR EACH ROW
BEGIN
    UPDATE DEPARTAMENTOS SET NUMEMP = NUMEMP + 1
    WHERE CODDPTO = NEW.DPTO;
END$$
DELIMITER ;

```

El resultado será:

```

mysql> SOURCE C:\src\trigger_01.sql
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
mysql> insert into empleados
-> values('112233445P','Pedro Gil','Informática','2013/10/10',
-> 'IT','MAD20',1200);
Query OK, 1 row affected (0.11 sec)

mysql> select * from departamentos;
+-----+-----+-----+-----+
| CodDpto | Nombre          | Ubicacion          | numemp |
+-----+-----+-----+-----+
| ADM     | Administración  | Planta quinta U2   | 0      |
| ALM     | Almacén         | Planta baja U1     | 1      |
| CONT    | Contabilidad    | Planta quinta U1   | 1      |
| IT      | Informática     | Planta sótano U3   | 4      |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

Para eliminar un *trigger* bastará con ejecutar la instrucción:

```
DROP TRIGGER nombre_disparador;
```

Por ejemplo:

```
DROP TRIGGER nuevo_empleado;
```

También puedes ver los disparadores existentes con la instrucción:

```
SHOW TRIGGERS;
```

Igual que en Oracle, crea un disparador para que descuente un empleado del departamento indicado cuando sea eliminado un empleado de la tabla.

Vamos a realizar ahora en MySQL, el ejemplo que ya hemos realizado en Oracle sobre la actualización de un empleado. Si durante la actualización del empleado se le cambia de departamento deberemos actualizar el número de empleados de los departamentos involucrados:

```

DELIMITER $$
CREATE TRIGGER cambio_en_empleado
    AFTER UPDATE
    ON EMPLEADOS
    FOR EACH ROW
BEGIN
    IF NEW.DPTO != OLD.DPTO THEN
        UPDATE DEPARTAMENTOS SET NUMEMP = NUMEMP - 1
            WHERE CODDPTO = OLD.DPTO;
        UPDATE DEPARTAMENTOS SET NUMEMP = NUMEMP + 1
            WHERE CODDPTO = NEW.DPTO;
    END IF;
END$$
DELIMITER ;

```

Una vez ejecutado el script, actualizaremos el departamento del nuevo empleado y veremos si ha funcionado nuestro nuevo disparador:

```

mysql> SOURCE C:\src\trigger_02.sql
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE EMPLEADOS SET DPTO = 'CONT'
    -> WHERE DNI = '112233445P';
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM DEPARTAMENTOS;
+-----+-----+-----+-----+
| CodDpto | Nombre          | Ubicacion          | numemp |
+-----+-----+-----+-----+
| ADM     | Administración  | Planta quinta U2   | 0      |
| ALM     | Almacén         | Planta baja U1     | 1      |
| CONT    | Contabilidad    | Planta quinta U1   | 2      |
| IT      | Informática     | Planta sótano U3   | 3      |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```