UT7 PROGRAMACIÓN BASES DE DATOS

ISABEL Mª SOLANA LUMBRERAS

OBJETIVOS DE ESTA UNIDAD

- CONOCER LOS LENGUAJES DE PROGRAMACIÓN DE BASES DE DATOS.
- APRENDER LAS TÉCNICAS BÁSICAS DE PROGRAMACIÓN QUE INCORPORA MYSQL (Y PL/SQL ORACLE).
- CONOCER APLICACIONES DE PROGRAMACIÓN DE BASES DE DATOS REALES.
- IMPLEMENTAR Y GESTIONAR OBJETOS DE CONTROL DE BASES DE DATOS: PROCEDIMIENTOS, FUNCIONES, DISPARADORES (TRIGGERS) Y EVENTOS.

INDICE

TODA APLICACIÓN INFORMÁTICA CONSTA DE DOS PARTES DIFERENCIADAS: DATOS Y CÓDIGO.

código

```
mysql> SELECT * FROM alumnos;
+------------------+-------+
| nombre           | edad  |
+------------------+-------+
| Jose Sanchez     |    22 |
| Alberto Jurado   |    19 |
| Carlos Martín    |    20 |
+------------------+-------+
3 rows in set (0,00 sec)
```

datos

Los datos, se organizan en bases de datos de tipo relacional, orientado a objetos u otros tipos.
El código o funciones se utiliza para manipular dichos datos. Mediante distintos lenguajes como C,php, perl, … para bbdd relacionales. Y de modo más avanzado se utiliza C++, Java o .NET para bbdd objeto relacionales.

También pueden usarse lenguajes propios del SGBD que permiten integrar datos y funcionalidad dentro de la misma base de datos. Ventajas:

INDEPENDENCIA DEL SISTEMA OPERATIVO (INSTALO EN SGBD Y NO NECESITO NADA MÁS)

APLICACIONES MÁS LIGERAS (PARTE DE LA CARGA DEL RPOCESO LA TIENE EL SERVIDOR => APLICACIONES CON MENOS CÓDIGO)

FÁCIL MANTENIMIENTO (SÓLO AFECTA AL SISTEMA GESTOR Y NO A OTRAS APLICACIONES AJENAS A ÉL)

Vemos a continuación en el apartado 1 un repaso de la tecnología actual así como un repaso de distintos lenguajes de programación en varios SGBD. Después nos centramos en MySQL.

7.1 LENGUAJES DE PROGRAMACIÓN Y BASES DE DATOS

- EN LOS PRIMEROS TEMAS ESTUDIAMOS QUE LAS BASES DE DATOS SON UN CONJUNTO ORGANIZADO DE DATOS, QUE USA UNA EMPRESA U ORGANIZACIÓN. PERO ¿ QUÉ HACEMOS CON ESOS DATOS? ¿ CÓMO LOS GESTIONAMOS?
- MEDIANTE LENGUAJES DE PROGRAMACIÓN. A LO LARGO DE LA HISTORIA HAN EXISTIDO MUCHOS: ACCESS, COBOL, NATURAL, … ALGUNOS SE DENOMINAN ANFITRIONES PORQUE NO SON PROPIOS DEL SISTEMA GESTOR.

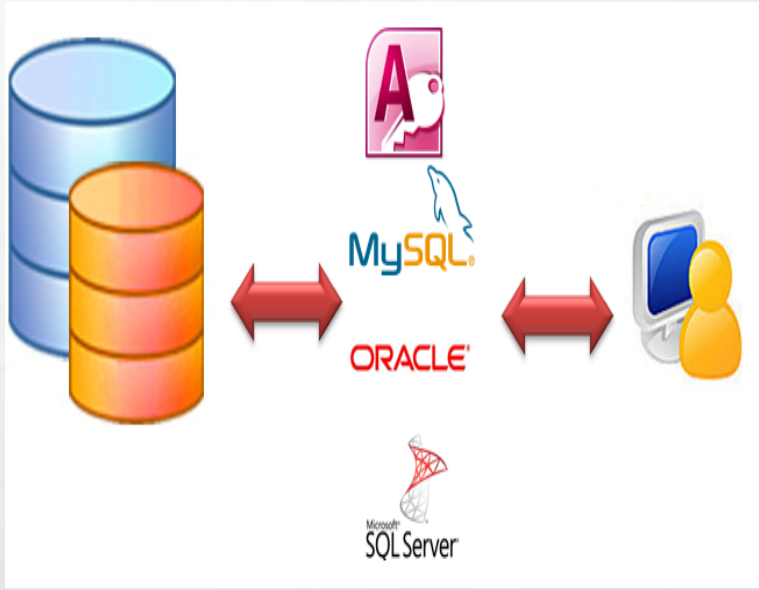
- **Los propios SGBD incorporan más potentes lenguajes propios, integrados en el software:**

MySQL: Permite definir rutinas, disparadores, vistas y eventos mediante un lenguaje propio.

ORACLE: Incorpora PL/SQL para programar los objetos de la base de datos.

SQL Server: Incorpora Transact-SQL y T-SQL para implementar sentencias SQL y programar también rutinas, disparadores y otros objetos.

PosgreSQL: permite, mediante el uso de módulos, ser compilado para usar lenguajes diversos, normalmente PL/PGSQL pero también PL/PHP, PL/R, PL/Java.



Todo esto está cambiando mucho en la actualidad con la aparición del conocido BigData, cuyo significado es “gran volumen de datos”, tanto estructurados como no estructurados, y que inundan los negocios cada día. La pregunta es ¿ cómo gestionan las organizaciones dichos datos? Que os suene que se utilizan otros lenguajes de programación como R y Phyton.

Son lenguajes que no vamos a considerar en nuestro curso. Pero conviene que lo tengamos en cuenta y que nos suenen. Phyton es muy fácil de aprender si se tienen conocimientos previos de programación.

7.2 PROCEDIMIENTOS Y FUNCIONES ALMACENADOS EN MYSQL

- RUTINAS (PROCEDIMIENTOS O FUNCIONES) -> “CONJUNTO DE COMANDOS SQL QUE PUEDEN GUARDARSE EN EL SERVIDOR.” PARA EJECUTARLOS CUANDO LOS NECESITE.
- PUEDEN MEJORAR EL RENDIMIENTO (MENOS COMUNICACIÓN CLIENTE-SERVIDOR)
- MAYOR CARGA DEL SERVIDOR QUE REALIZA MÁS TRABAJO QUE EL CLIENTE.

DIFERENCIAS ENTRE PROCEDIMIENTOS Y FUNCIONES

PROCEDIMIENTOS

- Pueden devolver algo o no
- Para devolver valores usan OUT Ó INOUT

FUNCIONES

- Siempre devuelven algo (RETURNS/RETURN)
- Sólo devuelven cero o un único valor de retorno
- Se pueden utilizar en expresiones.
- Se pueden incluir en otras funciones o procedimientos.
- Se pueden incluir en cláusulas SELECT, UPDATE, DELETE e INSERT.

ESQUEMA GRAL DE RUTINA ALMACENADA EN SGBD

Nombre rutina+ parámetros (entrada, salida, ó e/s)

Declaración e inicialización de variables

Procesamiento de datos
Bloques BEGIN/END con instrucciones de control (condicionales y repetitivas)

FIN
Mediante la instrucción RETURN para devolver un valor en el caso de funciones almacenadas

7.2.1 SINTAXIS Y EJEMPLOS DE RUTINAS ALMACENADAS

- - PODRÍAMOS CREAR NUESTRAS RUTINAS CON UN EDITOR DE TEXTO TIPO NOTEPAD O BIEN COMO VAMOS A HACER EN ESTA UNIDAD CON MYSQL WORKBENCH O BIEN CON LA CONSOLA DE MYSQL DE XAMPP.

13.1.17 CREATE PROCEDURE and CREATE FUNCTION Statements

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


CREATE [DEFINER = user] FUNCTION sp_name ([func_parameter[,...]])
RETURNS type [characteristic ...]
routine_body proc_parameter: [ IN | OUT | INOUT ]
param_name type func_parameter: param_name type type: Any valid MySQL data type
characteristic: COMMENT 'string' | LANGUAGE SQL | [NOT] DETERMINISTIC | { CONTAINS SQL |
NO SQL | READS SQL DATA | MODIFIES SQL DATA } | SQL SECURITY { DEFINER | INVOKER }
routine_body: Valid SQL routine statement
```

Sp_name es el nombre de la rutina almacenada
Parameter son los parámetros que tienen un tipo (in, out, inout), un nombre.
Routine_body es el cuerpo de la rutina (sentencias SELECT).
Comienza con Begin y termina con End.
Contains SQL/no SQL : si tienes sentencias SQL.
SQL Security: si necesita permisos del creador o del invocador.

Parámetros de una rutina

- IN: entrada
- OUT: salida
- INOUT: entrada/salida

Variables en una rutina

- Tipos: de datos
- Alcance variables: determinado por bloque BEGIN/END

EJEMPLOS

- **Vamos a ir viendo ejemplos típicos de rutinas guardándolas en la base de datos test de MySQL Workbench. (si no tenéis esta base de datos la creáis en vuestro programa gestor, la activáis y guardáis ahí los ejemplos).**

EJEMPLO 7.1 PROCEDIMIENTO QUE IMPRIME POR PANTALLA LA CADENA 'HOLA MUNDO'

```
DELIMITER $$

DROP PROEDURE IF EXISTS hola_mundo$$

CREATE PROCEDURE test.hola_mundo()

BEGIN

SELECT 'hola mundo'

END $$
```

Guardaré el procedimiento en un fichero llamado hola_mundo.sql que podré ejecutar cuando quiera con el comando
CALL nombre de fichero

Ejemplo:

```
mysql> CALL hola_mundo.sql
```

Para comprobar que el procedimiento existe en la base de datos o bien comprobar su contenido escribiremos:

```
mysql> SHOW CREATE PROCEDURE hola_mundo;
```

Ejemplo 7.2 Mostrar la versión de MySQL

```
CREATE PROCEDURE version
SELECT version();
$$
```

Ejemplo 7.3 Procedimiento para obtener la fecha actual y un número aleatorio.

```
DELIMITER $$
CREATE PROCEDURE fecha()
LANGUAGE SQL                    /* Porque usa una sentencia sql como es SELECT */
NOT DETERMINISTIC               /* No siempre devuelve lo mismo */
COMMENT 'A procedure'
SELECT CURRENT_DATE, RAND()  /* fecha actual y número aleatorio (por eso cada vez devuelve algo distinto) */
$$
```

Ejemplo 7.4 FUNCIÓN QUE RECIBE UN VALOR DE ESTADO COMO ENTRADA Y COMPRUEBA DICHO VALOR, ASOCIANDO UN TEXTO A estado EN FUNCIÓN DEL VALOR DE LA VARIABLE DE ENTRADA

```
DELIMITER $$
CREATE FUNCTION estado(in_estado CHAR(1))
RETURNS VARCHAR(20)
LANGUAGE SQL
DETERMINISTIC
BEGIN
DECLARE resultado VARCHAR(20);
COMMENT ‘ el estado debe ser P, 0 ó N’;
IF in_estado = ‘P’ THEN          /* IF-THEN-ELSE o IF-THEN-ELSEIF-ELSE son sentencias de selección */
  SET resultado=‘caducado’;
ELSEIF in_estado=‘O’ THEN
   SET resultado=‘activo’;
ELSEIF in_estado=‘N’ THEN
  SET resultado=‘nuevo’;
ELSE SET resultado='vble entrada P,O ó N';
END_IF;
RETURN(estado); END;$$
```

Las funciones se guardan en MySQL en una carpeta llamada functions
Para comprobar que la función existe en la base de datos o bien comprobar su contenido escribiremos:

```
mysql> SHOW CREATE FUNCTION estado;
```

Ejemplo 7.5 RECIBO UN NUMERO COMO ENTRADA Y DEVUELVE 0 SI ES PAR Y 1 SI ES IMPAR.

```
DELIMITER $$
CREATE FUNCTION esimpar(numero int)
RETURNS int
DETERMINISTIC
NO SQL
BEGIN
DECLARE impar INT;
      IF MOD(numero,2)=0 THEN SET impar=1;
      ELSE SET impar=0;
      END IF;
      RETURN (impar);
END;
$$
```

Para comprobar que la función devuelve un 0 cuando el número es impar o un 1 cuando es par puedo escribir usando directamente la cláusula SELECT

```
mysql> SELECT esimpar(42);
```

o bien asignar el valor a una variable de sesión @x, **ejemplo 7.6**

```
mysql> SET @x=esimpar(42);
```

En este caso el valor de retorno queda guardado en @x y puedo consultarlo cuando quiera, por ejemplo escribiendo de nuevo SELECT @x.

Sin embargo, es más común llamar a las funciones desde otras funciones o procedimientos como este **ejemplo 7.7**:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS muestra_estado$$
CREATE PROCEDURE muestra_estado(in numero int)
BEGIN
IF (esimpar(numero)) THEN
      SELECT CONCAT(numero," es impar");
else
      SELECT CONCAT(numero," es par");
end if; END;$$
```

IMPORTANTE

- De este modo las funciones permiten reducir la complejidad del código encapsulando parte de este código.
- Se hace así más legible un programa.
- Más fácil de mantener.
- Más fácil de reutilizar.

La forma de llamar a este procedimiento en mysql sería:

Mysql> call muestra_estado(42)

Y me devolvería

42 es par

En este caso se recibe una variable entera de entrada llamada *parámetro1*. A continuación se declaran sendas variables *variable1* y *variable2* de tipo entero y se testea el valor del parámetro. En caso de que sea 17 se asigna su valor al a variable *v1* y si no la variable *v2* se le asigna el valor 30.

```
DELIMITER $$
DROP PROCEDURE IF EXISTS proc1 $$
CREATE PROCEDURE proc1 /*nombre */
(IN parametro1 INTEGER)        /*parametros */
BEGIN                           /*comienzo de bloque */
DECLARE variable1 INTEGER;     /*variables */
DECLARE variable2 INTEGER;     /*variables */
IF parametro1 = 17 THEN  /*instrucción condicional */
SET variable1 = parametro1;    /*asignación */
ELSE
SET variable2 = 30;            /*asignación */
END IF;                   /*fin de condicional*/
INSERT INTO t VALUES
(variable1),(variable2); /*instruccion sql */
END $$                    /*final de bloque*/
DELIMITER ;$$
```

Obviamente el ejemplo es incoherente y solo tiene propósitos didácticos.

7.2.2 PARAMETROS Y VARIABLES

- IGUAL QUE EN OTROS LENGUAJES DE PROGRAMACIÓN, LOS PROCEDIMIENTOS Y FUNCIONES USAN VARIABLES Y PARÁMETROS QUE DETERMINAN LA SALIDA DEL ALGORITMO.

Nuevas cláusulas para el manejo de variables

DECLARE: crea una nueva variable con un nombre y un tipo (char, varchar, int, float, …). Puede llevar DEFAULT (indica un valor por defecto) o bien, si no indica nada, valdará NULL.

```
DECLARE a, b INT DEFAULT 5;   // Crea 2 vbles. enteras con valor 5 por defecto
```

SET: permite asignar valores a las variables utilizando el operador de igualdad. (ver ejemplo 7.8)

Nuevas cláusulas para el manejo de variables

DECLARE: crea una nueva variable con un nombre y un tipo (char, varchar, int, float, …). Puede llevar DEFAULT (indica un valor por defecto) o bien, si no indica nada, valdará NULL.

```
DECLARE a, b INT DEFAULT 5;   // Crea 2 vbles. enteras con valor 5 por defecto
```

SET: permite asignar valores a las variables utilizando el operador de igualdad. (ver ejemplo 7.8)

Alcance de las variables

Las variables tienen un alcance determinado por el bloque BEGIN END en el que se encuentren.
Sólo podremos ver una variables de un procedimiento fuera de este si la hemos definido como OUT o bien porque hayamos asignado como entrada una variable de sesión (@)

```
DELIMITER $$
DROP PROCEDURE IF EXISTS proc5 $$
CREATE PROCEDURE proc5()
BEGIN
  DECLARE X1 CHAR(5) DEFAULT 'fuera';
BEGIN
  DECLARE X2 CHAR(6) DEFAULT 'dentro';
  SELECT X2;
END;
SELECT X1;
END; $$
```

Probad a ejecutar CALL proc5 metiendo SELECT X2 fuera del end en el que se encuentra.
Y veréis cómo no os dejar porque no reconoce dicha variable fuera de su ámbito.

ACTIVIDADES 7.1

- Sobre la base de pruebas *test* cree un procedimiento para mostrar el año actual.
- Cree y muestre una variable de usuario con *SET*. ¿Debe ser de sesión o puede ser global?
- Use un procedimiento que sume uno a la variable anterior cada vez que se ejecute.
- En este caso la variable es de entrada/salida ya que necesitamos su valor para incrementarlo y además necesitamos usarlo después de la función para comprobarlo.
- Cree un procedimiento que muestre las tres primeras letras de una cadena pasada como parámetro en mayúsculas.
- Cree un procedimiento que muestre dos cadenas pasadas como parámetros concatenadas y en mayúscula.
- Cree una función que devuelva el valor de la hipotenusa de un triángulo a partir de los valores de sus lados
- Cree una función que calcule el total de puntos en un partido tomando como entrada el resultado en formato 'xxx-xxx'.

7.2.3 INSTRUCCIONES CONDICIONALES

- CUANDO EL VALOR DE UNA O MÁS VARIABLES O PARÁMETROS DETERMINARÁ EL RESULTADO HAY QUE USAR INSTRUCCIONES CONDICIONALES DE TIPO SIMPLE “IF” O BIEN, SI EXISTEN VARIAS ALTERNATIVAS, USAR “IF-THEN-ELSE”, O, SI HAY MÚLTIPLES POSIBILIDADES, USAR “CASE”.

IF-THEN-ELSE

- **Podemos usar instrucciones condicionales usando IF, o, de manera más completa IF-THEN-ELSE**

If-then-else

```
IF expr1 THEN
...
ELSEIF expr2 THEN
...
ELSE
...
END IF
```

En el siguiente ejemplo insertamos o actualizamos la tabla de prueba t en la base de datos test según el valor de entrada:

```
DELIMITER $$
CREATE PROCEDURE proc7 (IN par1 INT)
BEGIN
DECLARE var1 INT;
SET var1 = par1 + 1;
IF var1 = 0 THEN                          // insercion
INSERT INTO t VALUES (17,12);
END IF;
IF par1 = 0 THEN
UPDATE t SET valor1 = valor1 + 1;         // actualizar valor1
ELSE
UPDATE t SET valor2 = valor2 + 1;         // actualizar valor2
END IF;
END; $$
```

CASE

- **Cuando hay muchas condiciones es más apropiado el uso de CASE (en lugar de varios if anidados)**

CASE

```
CASE expression
WHEN val1 THEN
…
[WHEN val2 THEN
...]
[ELSE
…]
END CASE;
```

El valor de expresión puede ser val1 o bien val2… en otro caso que no sea ninguno de los when entra Por ELSE.

Ejemplo con CASE. USAMOS INSERT INTO para guardar el valor de una variable en una tabla:

```
DELIMITER $$
CREATE PROCEDURE proc8 (IN par1 INT)
BEGIN
DECLARE var1 INT;
SET var1 = par1 + 1;
CASE var1                                   // inserción
WHEN 0 THEN INSERT INTO t VALUES (17,12);
WHEN 1 THEN INSERT INTO t VALUES (20,21);
ELSE        INSERT INTO t VALUES (0,0);
END CASE;
END; $$
```

ACTIVIDADES 7.2

7.2.1 Crear una función que devuelva un 1 ó un 0 si un número es o no divisible por otro. => 2 parámetros de entrada
7.2.2 Use las estructuras condicionales para mostrar el día de la semana según un valor de entrada que sea numérico: 1 para domingo, 2 para lunes, … (IF ó CASE)
7.2.3 Cree una función que devuelva el mayor de 3 números pasados como parámetros.
7.2.4 Usando las tablas de la base de datos liga, cree una función que devuelva 1 si ganó el visitante y 0 en caso contrario.
7.2.5 Cree una función que diga si una frase, pasada como parámetro, es palíndroma.
Nota: una palabra o frase palíndroma es, por ejemplo "Dábale arroz a la zorra el abad" porque se lee lo mismo de izquierda a derecha que de derecha a izquierda.
7.2.6 Cree una función en la base de datos liga que compruebe si los partidos ganados por un equipo coinciden con el campo partidosganados de la tabla equipo. (mayor dificultad)
7.2.7 Use una función para insertar registros de movimientos en una cuenta de un cliente, comprobando previamente que la fecha es menor que la actual y que la operación no deja la cuenta en negativo. La función devolverá un 0 en caso de error de entrada y un 1 en cualquier otro caso. (para empezar, cree en la base de datos test una tabla llamada insertar_mov con los campos cantidad, ccli, cc, fecham y después trabaje con estos campos)

7.2.4 INSTRUCCIONES REPETITIVAS O LOOPS

- LOS BUCLES (LOOPS) PERMITEN ITERAR (REPETIR) UN CONJUNTO DE INSTRUCCIONES UN NÚMERO DETERMINADO DE VECES. MYSQL TIENE PARA ELLO 3 TIPOS DE INSTRUCCIONES:

SIMPLE LOOP, REPEAT UNTIL, WHILE LOOP.

SIMPLE LOOP

Sintaxis básica

[etiqueta:] LOOP
instrucciones
END LOOP
[etiqueta];

La etiqueta permitiría identificar El bucle con ese nombre.
También existe otro tipo de LOOP que se denomina Infinite_loop Pero como no se recomienda no lo vemos.

En el siguiente ejemplo etiquetamos el loop con el nombe loop_label

```
DELIMITER $$
CREATE PROCEDURE proc9 ()
BEGIN
DECLARE cont INT;
SET cont = 0;
loop_label: LOOP
  INSERT INTO t VALUES (cont);  // inserta una fila en la tabla t
  SET cont = cont+1;
  IF cont >= 5 THEN
    LEAVE loop_label;
  END IF;
END LOOP;
END; $$
```

REPEAT UNTIL LOOP

Sintaxis básica

[etiqueta:] REPEAT
instrucciones
UNTIL expresion
END REPEAT [etiqueta]

En el siguiente ejemplo se muestran los números impares de 0 a 10

```
DELIMITER $$
CREATE PROCEDURE proc10 ()
BEGIN
DECLARE i INT;
SET i = 0;
loop1: REPEAT
  SET i = i+1;
  IF MOD(i,2) <> 0 THEN  /* numero impar */
    SELECT CONCAT(i," es impar");
  END IF;
UNTIL i>=10
END REPEAT;
END; $$
```

WHILE LOOP

Sintaxis básica

WHILE Expresión DO
instrucciones
END WHILE [etiqueta]

En el siguiente ejemplo se muestran los números impares de 0 a 10 (mismo ejemplo anterior pero con WHILE LOOP)

```
DELIMITER $$
CREATE PROCEDURE proc10b ()
BEGIN
DECLARE i INT;
SET i = 1;
loop1: WHILE i<=10 DO
  IF MOD(i,2) <> 0 THEN  /* numero impar */
    SELECT CONCAT(i," es impar");
  END IF;
  SET i=i+1;
UNTIL i>=10
END WHILE loop1;
END; $$
```

ACTIVIDADES 7.3

7.3.1 Sobre la base test cree un procedimiento que muestre la suma de los primeros n números enteros, siendo n un parámetro de entrada. (LOOP)

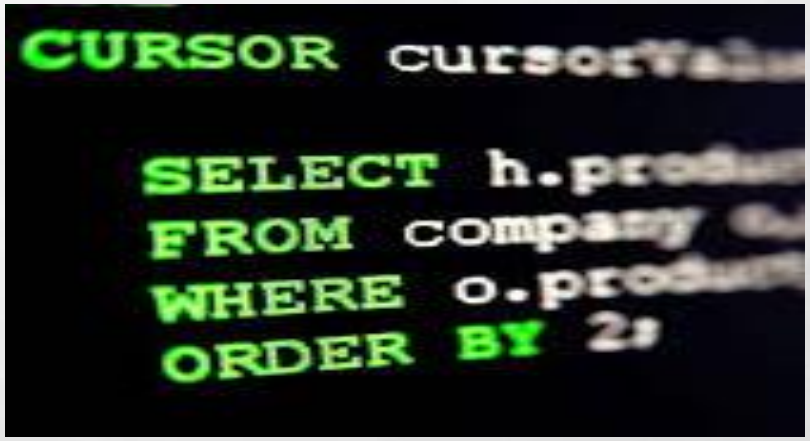
7.3.2 Haga un procedimiento que muestre la suma de los términos 1/n con n entre 1 y m. Es decir, 1/1+½+1/3+…+1/m Siendo m el parámetro de entrada. Nota: m no puede ser cero. (LOOP)

7.3.3 Cree una función que determine si un número es primo devolviendo 0 ó 1.

7.3.4 Usando la función anterior, cree otra que calcule la suma de los primeros m números primos empezando en el 1.

7.2.5 SQL EN RUTINAS: CURSORES

- CUANDO USAMOS INSTRUCCIONES SQL QUE AFECTAN A DATOS DE TABLAS , INSERT, UPDATE, DELETE, ...
- SI QUEREMOS RECUPERAR MÁS DE UNA FILA DE UNA TABLA VAMOS A NECESITAR EL USO DE CURSORES.
- UN CURSOR NORMALMENTE SE ASOCIA A UN CONJUNTO DE FILAS O A UNA CONSULTA SOBRE UNA TABLE DE UNA BASE DE DATOS.



EJEMPLO USO CURSOR

Sintaxis básica

DECLARE cursor1 CURSOR FOR
SELECT campo1, … FROM tabla;

IMPORTANTE: Debe definirse después de declarar todas las variables necesarias en el procedimiento.

Creación de un cursor formado por los campos id y título de la tabla noticias cuyo id coincida con el parámetro id_noticia de entrada.

```
DELIMITER $$
CREATE PROCEDURE cursor_demo(id_noticia INT)
BEGIN
DECLARE vid INT(11);
DECLARE vtitulo VARCHAR(255);
DECLARE c1 CURSOR FOR
SELECT id, titulo
FROM noticias
WHERE id=id_noticia;
END; $$
```

Ejecutaríamos el procedimiento como siempre
CALL cursor_demo(66); // para una determinada noticia

COMANDOS RELACIONADOS CON CURSORES

Para manipular los cursores disponemos de una serie de comandos

OPEN: inicializa el conjunto de resultados asociados al cursor

```
OPEN nombre_cursor
```

FETCH: extrae la siguiente fila de valores del conjunto de resultados del cursor.
Tiene un puntero interno que mueve una posición.

```
FETCH nombre_cursor INTO variable;
```

CLOSE: cierra el cursor, libera la memoria que ocupa y ya no se puede acceder a sus datos.

```
CLOSE nombre_cursor;
```

Si se va a invocar a más de una fila se va a necesitar un bucle

Y para evitar errores cuando queramos leer una fila y no haya más datos, podemos usar lo que se conoce como "manejador de errores" o "handler":

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

Esta sentencia
* **Establece la variable l_last_row_fetched a 1. (indica la última fila)**
* **Permite al programa continuar su ejecución. (sin dar error)**

EJEMPLO CURSOR CON CONTROL DE ERRORES

```
DELIMITER $$
CREATE PROCEDURE cursor_demo3()
BEGIN
DECLARE tmp VARCHAR(200);
DECLARE lrf BOOL;
DECLARE nn INT;
DECLARE cursor2 CURSOR FOR
SELECT titulo FROM noticias;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET lrf=1;
SET lrf=0, nn=0;
OPEN cursor2;
l_cursor: LOOP
   FETCH cursor2 INTO tmp;
   SET nn=nn+1;
   IF lrf=1 THEN leave l_cursor;
END IF;
END LOOP l_cursor;
CLOSE cursor2;
SELECT nn;
END; $$
```

EJEMPLO MÁS SOFISTICADO: DA EL Nº DE NOTICIAS PUBLICADAS DE CADA AUTOR (2 CURSORES)

```
DELIMITER $$
CREATE PROCEDURE noticias_autor()
reads sql data
BEGIN
DECLARE vautor, vnoticia, na_count INT;
DECLARE fin BOOL;
DECLARE autor_cursor CURSOR FOR SELECT id_autor FROM autores;
DECLARE noticia_cursor CURSOR FOR SELECT autor_id FROM noticias WHERE autor_id=vautor;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin=1;
OPEN autor_cursor;
autor_loop: LOOP
        FETCH autor_cursor INTO vautor;
        IF fin=1 THEN leave autor_loop;
        END IF;
        OPEN noticia_cursor;
        SET na_count=0;
        noticias_loop: LOOP
                FETCH noticia_cursor INTO vnoticia;
                IF fin=1 THEN leave autor_loop;
                END IF;
                SET na_count = na_count+1;
        END LOOP noticias_loop;
        CLOSE noticia_cursor;
        SET fin=0;
        SELECT CONCAT("El autor", vautor, "tiene", na_count, "noticias");
END LOOP autor_loop;
CLOSE autor_cursor;
END; $$
```

Para cada autor de la tabla autores se obtiene un cursor con sus noticias. Se usará para contar el número de noticias y devolver el resultado usando un SELECT que muestra la información correspondiente a cada autor.

7.2.6 GESTION DE RUTINAS ALMACENADAS

- LAS RUTINAS SE MANIPULAN CON CREATE, DROP Y SHOW.
- ELIMINACIÓN DE RUTINAS

  DROP {PROCEDURE|FUNCTION}[IF EXISTS] NOMBRE_RUTINA
- CONSULTAR RUTINAS

  SHOW CREATE {PROCEDURE|FUNCTION} NOMBRE_RUTINA

EN GENERAL, TODOS LOS COMANDOS DEL TIPO “SHOW” LOS PODEMOS EMPLEAR. (VER MANUALES SQL)

ACTIVIDADES 7.4

7.4.1 Desarrolle, usando cursores, un procedimiento que muestre los datos del cliente, la cuenta y el saldo de los clientes con saldo negativo en alguna de sus cuentas. (sobre la bd ebanca)

7.4.2 Calcule, con un procedimiento, la suma de todos los ingresos y cargos (por separado) en cada una de las cuentas de la base de datos ebanca.

7.2.7 MANEJO DE ERRORES

- LOS PROGRAMAS, DE MANERA GENERAL, INTERRUMPEN ABRUPTAMENTE SU EJECUCIÓN CUANDO SE PRODUCE UN ERROR. SE TERMINA LA EJECUCIÓN DEL PROGRAMA.
- SI QUEREMOS QUE LOS ERRORES SEAN CONTROLADOS Y NO SALIRNOS DEL PROGRAMA SINO INFORMAR DEL DETERMINADO ERROR Y LUEGO, CONTINUAR LA EJECUCIÓN DEBEMOS USAR CONTROLADORES DE EXCEPCIONES.

EJEMPLO MANEJADOR DE ERRORES

Si tuviésemos un procedimiento similar al de abajo insertar_autor y lo ejecutásemos mediante CALL insertar_autor(id_autor, 'login_de_prueba') y nos diese un error del tipo ERROR 1062 (23000) : 'Duplicate entry 'titulo1' for key 1

Podríamos solucionarlo añadiendo las líneas de código que aparecen en azul

```
DELIMITER $$
CREATE PROCEDURE insertar_autor (pautor INT, plogin VARCHAR(45), OUT estado VARCHAR(45))
MODIFIES SQL DATA
BEGIN
DECLARE CONTINUE HANDLER FOR 1062 SET estado='Duplicate entry 'titulo' for key 1';
SET estado='OK';
INSERT INTO AUTORES(id_autor, login) VALUES (pautor, plogin);
END; $$
```

De este modo, al llamar al procedimiento insertar_autor no devolvería en la variable estado el valor almacenado en el procedimiento de modo que podemos controlar el error y saber qué pasó. Si se cargó o no el valor. Y actuar en consecuencia.

SINTAXIS DE MANEJADOR

```
DECLARE {CONTINUE | EXIT} HANDLER FOR
        { SQLSTATE sqlstate_code  | MySQL error code | condition_name }
         handler_actions
```

TIPO DE MANEJADOR: EXIT o CONTINUE

CONDICIÓN DEL MANEJADOR: estado SQL (SQLSTATE), error propio de MySQL o código de error definido por el usuario.

ACCIONES DEL MANEJADOR: Acciones a tomar cuando se active el manejador.

TIPOS DE MANEJADOR

EXIT

Cuando se encuentra un error, el bloque que se está ejecutando actualmente se termina.
Si es el bloque principal, se termina el programa. Si otro programa invocó a este, se devuelve el control al programa que le haya invocado.
Adecuado para errores catastróficos.

CONTINUE

En este caso la ejecución continúa con la declaración siguiente a la que produjo el error.
Adecuado si se puede optar por un procesamiento alternativo.
Ver ejemplo siguiente de procedimiento que trata de insertar un autor cuya clave está repetida en la tabla. Si el autor ya existe, se trata con el manejador EXIT que, en caso de activarse, establece el valor de la variable duplicate_key a 1 y devuelve el control al bloque BEGIN/END exterior.

Si se invoca al procedimiento 2 veces consecutivas vemos el mensaje generado por el manejador informando de que el uso de la clave (id_autor) está repetida. EJEMPLO 7.3.1

EJEMPLO USO MANEJADOR DE ERRORES (ejemplo con controlador EXIT)

```
DELIMITER $$
CREATE PROCEDURE insertar_autor(pautor INT, plogin VARCHAR(45))
MODIFIES SQL DATA
BEGIN
DECLARE duplicate_key INT DEFAULT 0;
BEGIN
DECLARE EXIT HANDLER FOR 1062  /* clave repetida */
SET duplicate_key=1;
INSERT INTO autores(id_autor, login) VALUES (pautor, plogin);
END;

IF duplicate_key=1 THEN
SELECT CONCAT('error en la inserción clave duplicada',plogin);
ELSE
SELECT CONCAT('Autor ',plogin,'creado');
END IF;
END; $$
```

La llamada sería `CALL insertar_autor(8,'autor1');`

EJEMPLO USO MANEJADOR DE ERRORES (ejemplo con controlador CONTINUE)

```
DELIMITER $$
CREATE PROCEDURE insertar_autor(pautor INT, plogin VARCHAR(45))
MODIFIES SQL DATA
BEGIN
DECLARE duplicate_key INT DEFAULT 0;
BEGIN
DECLARE CONTINUE HANDLER FOR 1062; /* clave repetida */
SET duplicate_key=1;
INSERT INTO autores(id_autor, login) VALUES (pautor, plogin);

IF duplicate_key=1 THEN
SELECT CONCAT('error en la inserción de ', plogin, 'clave duplicada') as "Resultado";
ELSE
SELECT CONCAT('Autor ',plogin,'creado') as "Resultado";
END IF;
END; $$
```

La llamada sería `CALL insertar_autor(8,'autor1');`