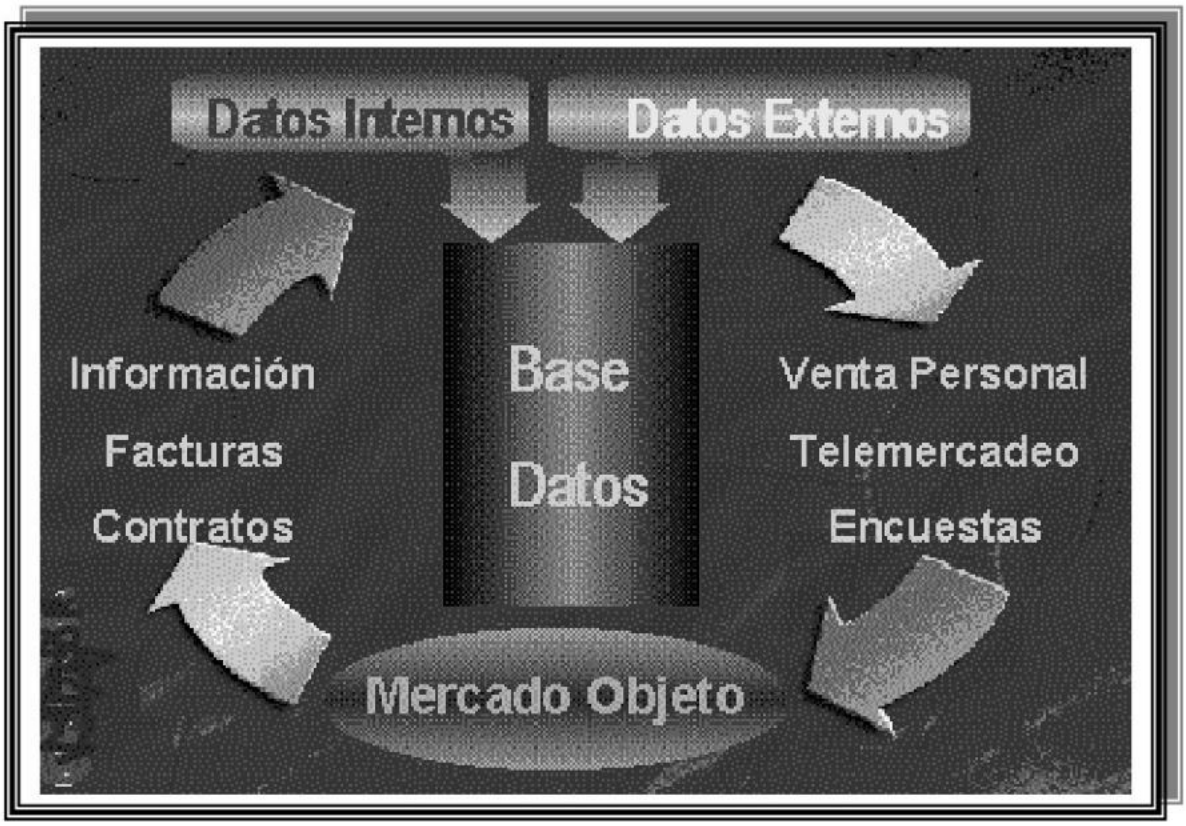

TEMA - VI:

PROGRAMACIÓN
DE
BASES DE DATOS

BD – (1º DAM)

ÍNDICE

1. PROCEDIMIENTOS ALMACENADOS Y FUNCIONES

Un procedimiento almacenado es un conjunto de comandos SQL que pueden almacenarse en el servidor. Una vez que se hace, los clientes no necesitan relanzar los comandos individuales pero pueden en su lugar referirse al procedimiento almacenado.

Algunas situaciones en que los procedimientos almacenados pueden ser particularmente útiles:

- Cuando múltiples aplicaciones cliente se escriben en distintos lenguajes o funcionan en distintas plataformas, pero necesitan realizar la misma operación en la base de datos.
- Cuando la seguridad es muy importante. Los bancos, por ejemplo, usan procedimientos almacenados para todas las operaciones comunes. Esto proporciona un entorno seguro y consistente, y los procedimientos pueden asegurar que cada operación se loguea apropiadamente. En tal entorno, las aplicaciones y los usuarios no obtendrían ningún acceso directo a las tablas de la base de datos, sólo pueden ejecutar algunos procedimientos almacenados.

Los procedimientos almacenados pueden mejorar el rendimiento ya que se necesita enviar menos información entre el servidor y el cliente. El intercambio que hay es que aumenta la carga del servidor de la base de datos ya que la mayoría del trabajo se realiza en la parte del servidor y no en el cliente. Considere esto si muchas máquinas cliente (como servidores Web) se sirven a sólo uno o pocos servidores de bases de datos.

Los procedimientos almacenados le permiten tener bibliotecas o funciones en el servidor de base de datos. Esta característica es compartida por los lenguajes de programación modernos que permiten este diseño interno, por ejemplo, usando clases. Usando estas características del lenguaje de programación cliente es beneficioso para el programador incluso fuera del entorno de la base de datos.

Los procedimientos almacenados y rutinas se crean con comandos CREATE PROCEDURE y CREATE FUNCTION . Una rutina es un procedimiento o una función. Un procedimiento se invoca usando un comando CALL , y sólo puede pasar valores usando variables de salida. 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. Las rutinas almacenadas pueden llamar otras rutinas almacenadas.

MySQL soporta la extensión muy útil que permite el uso de comandos regulares SELECT (esto es, sin usar cursores o variables locales) dentro de los procedimientos almacenados. El conjunto de resultados de estas consultas se envía directamente al cliente. Comandos SELECT múltiples generan varios conjuntos de resultados, así que el cliente debe usar una biblioteca cliente de MySQL que soporte conjuntos de resultados múltiples.

1.1. CREATE PROCEDURE y CREATE FUNCTION

```
CREATE PROCEDURE sp_name ([parameter[,...]])
    [characteristic ...] routine_body

CREATE FUNCTION sp_name ([parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body

parameter:
    [ IN | OUT | INOUT ] param_name type

type:
    Any valid MySQL data type

characteristic:
    LANGUAGE SQL
    | [NOT] DETERMINISTIC
    | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
    | SQL SECURITY { DEFINER | INVOKER }
    | COMMENT 'string'

routine_body:
    procedimientos almacenados o comandos SQL válidos
```

La cláusula RETURNS puede especificarse sólo con FUNCTION, donde es obligatorio. Se usa para indicar el tipo de retorno de la función, y el cuerpo de la función debe contener un comando RETURN value.

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 parámetro IN por defecto. Para especificar otro tipo de parámetro, use la palabra clave OUT o INOUT antes del nombre del parámetro. Especificando IN, OUT, o INOUT sólo es valido para una PROCEDURE.

MySQL almacena la variable de sistema sql_mode que está en efecto cuando se crea la rutina, y siempre ejecuta la rutina con esta inicialización.

La cláusula COMMENT es una extensión de MySQL, y puede usarse para describir el procedimiento almacenado. Esta información se muestra con los comandos SHOW CREATE PROCEDURE y SHOW CREATE FUNCTION .

El siguiente es un ejemplo de un procedimiento almacenado que use un parámetro OUT . El ejemplo usa el cliente s mysql y el comando delimiter para cambiar el delimitador del comando de ; a // mientras se define el procedimiento . Esto permite pasar el delimitador ; usado en el cuerpo del **procedimiento** a través del servidor en lugar de ser interpretado por el mismo s mysql.

```
mysql> delimiter //

mysql> CREATE PROCEDURE simpleproc (OUT param1 INT)
    -> BEGIN
    -> SELECT COUNT(*) INTO param1 FROM t;
    -> END
    -> //

mysql> delimiter ;

mysql> CALL simpleproc(@a);

mysql> SELECT @a;
+----+
| @a |
+----+
| 3 |
```

Al usar el comando delimiter, debe evitar el uso de la antibarra ('\') ya que es el carácter de escape de MySQL.

El siguiente es un ejemplo de **función** con un parámetro que retorna el resultado de una operación.

```
mysql> delimiter //
mysql> CREATE FUNCTION Saludo (s CHAR(20)) RETURNS CHAR(50)
    -> RETURN CONCAT('Hola, ',s,'!');
    -> //
mysql> delimiter ;
mysql> SELECT Saludo('Don Pepito');
+------------------+
| Hola, Don Pepito! |
+------------------+
```

EJEMPLO de Parametros:

```
DROP PROCEDURE Parame;
DELIMITER $
CREATE PROCEDURE Parame(IN E INT, OUT S INT, INOUT ES INT)
  BEGIN
    SELECT E, '-Antes';
    SET E=1;
    SELECT E, '-Despues';
    SELECT S,'-Antes';
    SET S=2;
    SELECT S, '-Despues';
    SELECT ES,'-Antes';
    SET ES=3;
    SELECT ES, '-Despues';
  END; $
DELIMITER ;
SET @A=10;
SET @B=20;
SET @C=30;
CALL Parame(@A,@B,@C);
SELECT @A,@B,@C;
```

EJEMPLO-1:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS `consultas`.`P` $$
CREATE PROCEDURE `consultas`.`P` ()
BEGIN
DECLARE A INT;
  SELECT COUNT(*) INTO A
    FROM Ped;
  IF A>2 THEN
    DELETE FROM Ped
       WHERE NP='P1';
  END IF;
END $$
DELIMITER ;
CALL P;
```

EJEMPLO-2:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS `consultas`.`TALLA` $$
CREATE PROCEDURE `consultas`.`TALLA` ()
BEGIN
   DECLARE MA,NUM INT;
   SELECT MAX(talla) INTO MA
    FROM Art;
  SELECT COUNT(*) INTO NUM
    FROM Art
    WHERE talla=MA;
  IF NUM=1 THEN
    UPDATE Art
      SET talla=5
      WHERE talla=MA;
  END IF;
END $$
DELIMITER ;
```

1.2. ALTER PROCEDURE y ALTER FUNCTION

```
ALTER {PROCEDURE | FUNCTION} sp_name [characteristic ...]
characteristic:
```

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

Este comando puede usarse para cambiar las características de un procedimiento o función almacenada.

1.3. DROP PROCEDURE y DROP FUNCTION

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

Este comando se usa para borrar un procedimiento o función almacenado. Esto es, la rutina especificada se borra del servidor.

1.4. SHOW CREATE PROCEDURE y SHOW CREATE FUNCTION

```
SHOW CREATE {PROCEDURE | FUNCTION} sp_name
```

Este comando es una extensión de MySQL . Similar a SHOW CREATE TABLE, retorna la cadena exacta que puede usarse para recrear la rutina nombrada.

```
mysql> SHOW CREATE FUNCTION test.hello\G
*************************** 1. row ***************************
Function: hello
sql_mode:
Create Function: CREATE FUNCTION `test`.`hello`(s CHAR(20)) RETURNS
CHAR(50)
RETURN CONCAT('Hello, ',s,'!')
```

1.5. SHOW PROCEDURE STATUS y SHOW FUNCTION STATUS

```
SHOW {PROCEDURE | FUNCTION} STATUS [LIKE 'pattern']
```

Este comando es una extensión de MySQL . Retorna características de rutinas, como el nombre de la base de datos, nombre, tipo, creador y fechas de creación y modificación. Si no se especifica un patrón, le lista la información para todos los procedimientos almacenados, en función del comando que use.

```
mysql> SHOW FUNCTION STATUS LIKE 'hello'\G
Db: test
Name: hello
Type: FUNCTION
Definer: testuser@localhost
Modified: 2004-08-03 15:29:37
Created: 2004-08-03 15:29:37
Security_type: DEFINER
Comment:
```

También puede obtener información de rutinas almacenadas de la tabla ROUTINES en INFORMATION_SCHEMA.

1.6. SENTENCIAS

1.6.1. La sentencia CALL

```
CALL sp_name([parameter[,...]])
```

El comando CALL invoca un procedimiento definido préviamente con CREATE PROCEDURE.

CALL puede pasar valores al llamador usando parámetros declarados como OUT o INOUT . También "retorna" el número de registros afectados, que con un programa cliente puede obtenerse a nivel SQL llamando la función ROW_COUNT() y desde C llamando la función de la API C mysql_affected_rows() .

1.6.2. Sentencia compuesta BEGIN ... END

```
[begin_label:] BEGIN
[statement_list]
END [end_label]
```

Los procedimientos almacenados pueden contener varios comandos, usando un comando compuesto BEGIN ... END .

1.6.3. Declarar variables locales con DECLARE

```
DECLARE var_name[,...] type [DEFAULT value]
```

Este comando se usa para declarar variables locales. Para proporcionar un valor por defecto para la variable, incluya una cláusula DEFAULT . El valor puede especificarse como expresión, no necesita ser una constante. Si la cláusula DEFAULT no está presente, el valor inicial es NULL. La visibilidad de una variable local es dentro del bloque BEGIN ... END donde está declarado.

1.6.4. Sentencia SET para variables

```
SET var_name = expr [, var_name = expr] ...
```

SET [@]variable = (SELECT ..)

1.6.5. La sentencia SELECT ... INTO

```
SELECT col_name[,...] INTO var_name[,...] table_expr
```

Esta sintaxis SELECT almacena columnas seleccionadas directamente en variables. Por lo tanto, sólo un registro puede retornarse.

```
SELECT id,data INTO x,y FROM test.t1 LIMIT 1;
```

1.6.6. Sentencia IF

```
IF search_condition THEN statement_list
[ELSEIF search_condition THEN statement_list] ...
[ELSE statement_list]
END IF
```

IF Valor [NOT] IN (SELECT A FROM T)

1.6.7. La sentencia CASE

```
CASE case_value
WHEN when_value THEN statement_list
[WHEN when_value THEN statement_list] ...
[ELSE statement_list]
END CASE
```

O:

```
CASE
WHEN search_condition THEN statement_list
[WHEN search_condition THEN statement_list] ...
[ELSE statement_list]
END CASE
```

El comando CASE para procedimientos almacenados implementa un constructor condicional complejo. Si una *search_condition* se evalúa a cierto, el comando SQL correspondiente se ejecuta. Si no coincide ninguna condición de búsqueda, el comando en la cláusula ELSE se ejecuta.

Nota: La sitaxis de un *comando* CASE mostrado aquí para uso dentro de procedimientos almacenados difiere ligeramente de la *expresión* CASE SQL. El comando CASE no puede tener una cláusula ELSE NULL y termina con END CASE en lugar de END.

1.6.8. Sentencia LOOP

```
[begin_label:] LOOP
      statement_list
END LOOP [end_label]
```

LOOP implementa un constructor de bucle simple que permite ejecución repetida de comandos particulares. El comando dentro del bucle se repite hasta que acaba el bucle, usualmente con un comando LEAVE .

Un comando LOOP puede etiquetarse. *end_label* no puede darse hasta que esté presente *begin_label* , y si ambos lo están, deben ser el mismo.

1.6.9. Sentencia LEAVE

```
LEAVE label
```

Este comando se usa para abandonar cualquier control de flujo etiquetado. Puede usarse con BEGIN ... END o bucles.

1.6.10. La setencia ITERATE

```
ITERATE label
```

ITERATE sólo puede aparecer en comandos LOOP, REPEAT, y WHILE . ITERATE significa "vuelve a hacer el bucle."

Por ejemplo:

```
CREATE PROCEDURE doiterate(p1 INT)
BEGIN
   label1: LOOP
      SET p1 = p1 + 1;
      IF p1 < 10 THEN ITERATE label1; END IF;
      LEAVE label1;
   END LOOP label1;
   SET @x = p1;
END
```

1.6.11. Sentencia REPEAT

```
[begin_label:] REPEAT
  statement_list
UNTIL search_condition
END REPEAT [end_label]
```

El comando/s dentro de un comando REPEAT se repite hasta que la condición *search_condition* es cierta. Un comando REPEAT puede etiquetarse. *end_label* no puede darse a no ser que *begin_label* esté presente, y si lo están, deben ser el mismo.

Por ejemplo:

```
mysql> delimiter //
mysql> CREATE PROCEDURE dorepeat(p1 INT)
            -> BEGIN
            -> SET @x = 0;
            -> REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;
            -> END
            -> //
Query OK, 0 rows affected (0.00 sec)

mysql> CALL dorepeat(1000)//
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x//
    +------+
    | @x |
    +------+
    | 1001 |
    +------+
    1 row in set (0.00 sec)
```

1.6.12. Sentencia WHILE

```
[begin_label:] WHILE search_condition DO
  statement_list
END WHILE [end_label]
```

El comado/s dentro de un comando WHILE se repite mientras la condición *search_condition* es cierta. Un comando WHILE puede etiquetarse. *end_label* no puede darse a no ser que *begin_label* también esté presente, y si lo están, deben ser el mismo.

Por ejemplo:

```
CREATE PROCEDURE dowhile()
BEGIN
   DECLARE v1 INT DEFAULT 5;
   WHILE v1 > 0 DO
      ...
      SET v1 = v1 - 1;
   END WHILE;
END
```

1.7. CURSORES

Se soportan cursores simples dentro de procedimientos y funciones almacenadas. La sintaxis es la de SQL empotrado. Los cursores no son sensibles, son de sólo lectura, y no permiten *scrolling*. No sensible significa que el servidor puede o no hacer una copia de su tabla de resultados.

1.7.1. Declarar cursores

```
DECLARE cursor_name CURSOR FOR select_statement
```

Este comando declara un cursor. Pueden definirse varios cursores en una rutina, pero cada cursor en un bloque debe tener un nombre único. El comando SELECT no puede tener una cláusula INTO

1.7.2. Sentencia OPEN del cursor

```
OPEN cursor_name
```

Este comando abre un cursor declarado préviamente.

1.7.3. Sentencia de cursor FETCH

```
FETCH cursor_name INTO var_name [, var_name] ...
```

Este comando trata el siguiente registro (si existe) usando el cursor abierto especificado, y avanza el puntero del cursro.

1.7.4. Sentencia de cursor CLOSE

```
CLOSE cursor_name
```

Este comando cierra un cursor abierto préviamente. Si no se cierra explícitamente, un cursor se cierra al final del comando compuesto en que se declara.

EJEMPLO-0:

```
DROP PROCEDURE Cursor2;
DROP TABLE TABLA_AUX;
CREATE TABLE TABLA_AUX
   (Nombrebis VARCHAR(15),
    Paibis VARCHAR(15));
DELIMITER //

CREATE PROCEDURE Cursor2 ()
BEGIN
  DECLARE Paisbis VARCHAR(30) DEFAULT "";
  DECLARE Nombrebis VARCHAR(30) DEFAULT "";
  DECLARE Done BOOLEAN DEFAULT FALSE;
  DECLARE Cur1 CURSOR FOR SELECT Nomp, Ciudadp FROM PRO;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET Done=TRUE;

  OPEN CUR1;
  FETCH CUR1 INTO Nombrebis,Paisbis;
  WHILE NOT Done DO
    INSERT INTO TABLA_AUX VALUES (Nombrebis,Paisbis);
    FETCH CUR1 INTO Nombrebis,Paisbis;
  END WHILE;
  CLOSE CUR1;
END
//

DELIMITER ;

CALL CURSOR2();
SELECT * FROM TABLA_AUX;
```

EJEMPLO-1:

```
CREATE PROCEDURE curdemo()
  BEGIN
   DECLARE done INT DEFAULT 0;
   DECLARE a CHAR(16);
   DECLARE b,c INT;
   DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
   DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
   DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
   OPEN cur1;
     OPEN cur2;
     FETCH cur1 INTO a, b;
     FETCH cur2 INTO c;
   WHILE done=0 DO
     IF NOT done THEN
       IF b < c THEN INSERT INTO test.t3 VALUES (a,b);
         ELSE INSERT INTO test.t3 VALUES (a,c);
       END IF;
     END IF;
     FETCH cur1 INTO a, b;
     FETCH cur2 INTO c;
   END WHILE;
   CLOSE cur1;
   CLOSE cur2;
  END
```

EJEMPLO-2:

```
CREATE PROCEDURE `consultas`.`TTT` ()
  BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE a,b,c INT;
    DECLARE cur1 CURSOR FOR SELECT id,D FROM test.t1;
    DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
    DROP TABLE IF EXISTS Combi;
    CREATE TABLE Combi (A INT, B INT);
    CREATE TABLE Com (A INT, B INT);
    CREATE TABLE T1  (ID INT, D INT);
    CREATE TABLE T2  (I INT);
    INSERT INTO T1 VALUES(1,2),(3,4),(5,6);
    INSERT INTO T2 VALUES(2),(3),(8);
    OPEN cur1;
    OPEN cur2;
    REPEAT
      FETCH cur1 INTO a, b;
      FETCH cur2 INTO c;
      IF NOT done THEN
          IF b < c THEN INSERT INTO Combi VALUES (a,b);
                ELSE INSERT INTO Com VALUES (a,c);
          END IF;
      END IF;
  UNTIL done END REPEAT;
  CLOSE cur1;
  CLOSE cur2;
END $$
```

2. EJERCICIOS

Los efectos de todas y cada una de las operaciones que se hagan en la BD deben ser anuladas (ROLLBACK), una vez comprobado su funcionamiento.

2.1. RESUELTOS

4. Crear una función que calcule el volumen de una esfera cuyo radio de tipo FLOAT se pasará como parámetro. Realiza una consulta después para calcular el volumen de una esfera de radio 5

```
CREATE FUNCTION volumen(radio FLOAT)
RETURNS FLOAT DETERMINISTIC
BEGIN
DECLARE volume FLOAT;
SET volume = (4/3)*PI()*POW(radius,3);
RETURN volume;
END $$
```

5. Escribe un procedimiento que reciba una palabra y la devuelva escrita del revés. Utiliza para ello un único parámetro de entrada y salida.

```
CREATE DEFINER=`admin`@`%` PROCEDURE `reves`(INOUT cadena varchar(25))
BEGIN
declare cad varchar(25) ;
declare myc char(1);
declare i int default 1;
declare lon int default CHAR_LENGTH(cadena);
set cad=SUBSTRING(cadena,lon, 1);
        WHILE i <= lon DO
    SET myc = SUBSTRING(cadena,lon-i, 1);
                SET cad=concat (cad,myc);
    SET i = i + 1;
  END WHILE;
set cadena=cad;
```

```
set @a='HOLA'
call revés(@a);
select @a
```

NOTA: un argumento OUT o INOUT debe ser una variable.

6. Crear un PA que cambie el mail de un cliente, tabla costumer, por otro que se pasará como parámetro, el PA recibirá dos parámetros, el identificador del cliente y el nuevo mail. Ejecutar el PA

```
CREATE DEFINER=`admin`@`%` PROCEDURE `cambio_mail`(identi smallint(5), mail
varchar(50) )
BEGIN
update customer set email=mail where customer_id=identi;
END
```

```
call cambio_mail(2,'patricia@gmail.com')
```

13. Crea un procedimiento que tenga un parámetro de entrada que será el nombre de la categoría y un parámetro de salida que contendrá el número de películas para esa categoría.

```
CREATE DEFINER=`admin`@`%` PROCEDURE `new_procedure`(IN categoria
varchar(45), OUT total INT)
BEGIN
select count(*) into total from film
where film_id in (select film_id from film_category where category_id in (select category_id
from category where
name= categoria));
END
```

```
Call ('Animation', @total)
Select @total as total_animacion;
```

14. Crea un procedimiento que reciba una cadena que puede contener letras y números y sustituya los número por *. Por ejemplo si hemos introducido la cadena 12abcd23rts, devolverá **abcd**rst
Consulta el manual de referencia sobre funciones
http://dev.mysql.com/doc/refman/5.0/es/functions.html

```
CREATE DEFINER=`admin`@`%` PROCEDURE `sin_numero`(INOUT str varchar(50))
BEGIN
DECLARE i INT DEFAULT 1;
  DECLARE myc CHAR(1);
  DECLARE outstr VARCHAR(50) DEFAULT str;
  WHILE i <= CHAR_LENGTH(str) DO
    SET myc = SUBSTRING(str, i, 1);
                IF myc IN ('1', '2', '3', '4', '5', '6', '7', '8', '9', '0') THEN
      SET outstr = INSERT(outstr,i,1,'*');
                    END IF;
    SET i = i + 1;
  END WHILE;
  SET str = outstr;
END
```

```
Set @str='12holaue998kte';
Call (@str)
Select @str
```

15. Desarrollar una función que devuelva el número de años completos que hay entre dos fechas que se pasan como parámetros. Utiliza la función DATEDIFF . Para visualizar el formato de la fecha con la que trabaja mysql en la sesión que estás utilizando visualiza la fecha actual utilizando la función current_date() .

```
CREATE DEFINER=`admin`@`%` FUNCTION `años_completos`(fecha1 date, fecha2
date) RETURNS int(11)
BEGIN
declare dias, comple integer;
select datediff(fecha1,fecha2)into dias;
set dias=truncate(dias/365,0);

RETURN dias;
END
```

16. Escribir una función que, haciendo uso de la función anterior, devuelva los trienios que hay entre dos fechas.

```
CREATE DEFINER=`admin`@`%` FUNCTION `trienios`(fecha1 date, fecha2 date)
RETURNS int(11)
BEGIN
declare trienios int;
select truncate(años_completos(fecha1,fecha2)/3,0) into trienios;
RETURN trienios;
END
```

17. Codificar un procedimiento que reciba una lista de hasta tres números y visualice su suma.

Nota, PL-SQL permite utilizar parámetros opcionales, mysql no, debes pasar tantos parámetros a la función o procedimiento como hayas definido en la función. Si luego no quieres pasar un parámetro en mysql, utiliza null a la hora de llamar a la función o procedimiento.

```
CREATE DEFINER=`admin`@`%` FUNCTION `suma`(n1 int, n2 int, n3 int) RETURNS
int(11)
BEGIN
if n2 is null
then
        if n3 is null
                then
                return n1;
        end if;
return n1+n3;
end if;
if n3 is null then
return n1+n2;
else
return n1+n2+n3;
end if;

END
```

18. Escribir un procedimiento que permita borrar un actor cuyo identificador se pasará como parámetro. Si el actor cuyo número se ha pasado como parámetro no existe, aparecerá un mensaje diciendo "Ese actor no existe". Comprueba el funcionamiento del procedimiento. ¿Qué ocurre cuando tratas de borrar un actor que ya existe?¿Porqué?

```
CREATE DEFINER=`admin`@`%` PROCEDURE `borrar_actor`(num smallint(5))
BEGIN
declare nume smallint(5);
select actor_id into nume from actor where actor_id=num;
if nume is  null then
select concat('El actor no existe', num);
else
delete from actor where actor_id=num;
```

26. Crea un procedimiento de modo que permita actualizar el campo special_features de la tabla film para una determinada película. Tendrá como parámetro el identificador del film y el parametro para special_features, defínelo como varchar(25).
Prueba el procedimiento con los datos (1, 'Special') ¿Qué ocurre?¿Porqué? Anota el número del error obtenido y realiza el tratamiento para este error.
Prueba ahora el procedimiento para los siguientes valores
(1, 'Trailers')
(1, 'Trailers222')

```
CREATE PROCEDURE `sakila`.`actualiza_pelicula` (id smallint(5), especial varchar(25))
BEGIN
declare valor_erroneo condition for 1265;
declare exit handler for valor_erroneo select 'no es válida esa opcion de special_features';
update film set special_features=especial where film_id=id;
END
```

27. Crea un procedimiento para dar de alta un nuevo actor, de modo que si la clave para ese nuevo actor ya existe, se pondrá como clave el valor inmediatamente superior al valor mayor de las claves. Por ejemplo si la clave mayor es 300, se pondrá como clave 301. Realiza el procedimiento utilizando el handler para el error 1062 se produce cuando se está duplicando una clave primaria al hacer una inserción.
Prueba el procedimiento con los siguientes valores

(193, 'Maria', 'Arnes')
(305, 'Julio', 'Arranz')

```
CREATE PROCEDURE `sakila`.`insertar_actor1` (id smallint(5), nombre varchar(25), apellido varchar(25))
BEGIN
declare iden smallint(5);
declare clave_duplicada condition for 1062;
declare exit handler for clave_duplicada
begin
select max(actor_id) into iden from actor;
insert into actor(actor_id, first_name, last_name) values (iden+1, nombre, apellido);
end;
insert into actor(actor_id, first_name, last_name) values (id, nombre, apellido);


END
```

28. En la BD World, crear un procedimiento para actualizar la población de un determinado país. Se pasarán dos parámetros, la nueva población de tipo float y el nombre del país. Realiza el procedimiento primero sin hacer el tratamiento de errores y pruébalo con los siguientes valores (Angola, 1234567891234) ¿Qué ocurre y porqué?
No permite porque la población introducida supera el valor permitido que es int(11)
Realiza ahora el tratamiento de los errores de modo que si se introduce un valor para la población mayor del permitido, se actualizará la población de ese país aumentándola un 10%. Si se introduce un país que no existe, se acabará el procedimiento con un mensaje indicando que el país no existe.

```
CREATE DEFINER=`admin`@`%` PROCEDURE `actualizar_poblacion`( pobla float, nombre char(52) )
BEGIN
declare nom char(52);
```

30. Crear un procedimiento que actualice el costo de reemplazo de las películas de una determinada categoría. Tendrá tres parámetros de entrada, el nombre de la categoría, un importe y un porcentaje. La subida será el porcentaje o el importe que se indica en el parámetro (el que sea superior.)

```
CREATE DEFINER=`admin`@`%` PROCEDURE `costo_actual`( categoria varchar(50),
importe decimal(5,2), porcentaje float)
BEGIN
declare done int default 0;
declare peli smallint(5);
declare costo decimal(5,2);
declare peliculas cursor for select distinct film_id from film_category where category_id
= (select category_id from category where name=categoria);
declare continue handler for not found set done=1;


open peliculas;
repeat
fetch peliculas into  peli;
select replacement_cost into costo from film where film_id=peli;
if (importe<costo*(1+porcentaje)) then
update film set replacement_cost=costo*(1+porcentaje) where film_id=peli;
else
update film set replacement_cost=importe where film_id=peli;
end if;
until done end repeat;
close peliculas;
END
```

1. Visualiza los triggers de la BD sakila de dos formas diferentes.

```
SHOW TRIGGERS
select * from information_schema.triggers where trigger_Schema='sakila'
```

2. Visualiza los triggers de la BD world.

```
select * from information_schema.triggers where trigger_Schema='world'
```

3. Con la información que muestra la tabla TRIGGERS de information_schema, ¿Qué tipo de evento y cuando disparan los triggers costumer_create_date, ins_film, upd_film, del_film?

sakila	customer_create_date	customer	0		SET NEW.create_date = NOW()	ROW	BEFORE

4. Visualiza los triggers de la tabla film. ¿Cuántos tiene?

```
select * from information_schema.triggers where trigger_Schema='sakila'     and
event_object_table='film'
```

2.2. PROPUESTOS

2.2.1. PROCEDIMIENTOS y FUNCIONES

1. Escribir la función ***Mayor(N1,N2,N3)*** a la cual se le pasan tres números, y devuelve el mayor de ellos.
2. Codifica una función a la cual se le pasa una **nota numérica y devuelve la correspondiente interpretación literal**:

   1-4.99 Suspenso 5-6 Suficiente 6-7 Bien 7-8 Notable 9-10 Sobresaliente Resto->ERROR.
3. Escribe una función que **sume tres números** pasados como parámetros, pero teniendo en cuenta que el valor de cualquiera de los parámetros puede ser NULL. No se debe usar función adicional.
4. Codificar la función ***Multi(N1,N2)*** que recibe dos números y devuelve el resultado de la multiplicación de ambos sin usar el * (cuidado con los valores negativos).
5. Codificar el procedimiento ***Divide(DDO,DSOR,CTE,RESTO)*** que recibe dos números y devuelve cociente (CTE) y el resto de dividir DDO entre DSOR, sin usar / ni operadores o funciones extras (cuidado con los valores negativos).
6. Escribe un procedimiento que **reciba una palabra y la devuelva escrita del revés**.
7. En los **ejercicios del TEMA-II** (BD Metro, …), conseguir con **funciones** o **procedimientos**:
   - Las especificaciones que no se pueden contemplar a nivel de diseño.
   - El control de la ***exclusividad*** entre tablas.
   - El control de la ***cardinalidad mínima*** entre tablas

BD: *Obras*

8. Crea un procedimiento para **aumentar el precio de la hora de las máquinas** (prciohora) en un porcentaje, dicho porcentaje se le pasa al procedimiento como parámetro de entrada.
9. Crea un procedimiento para **cambiar la categoría de los conductores** (categ). El procedimiento debe tener dos parámetros de entrada, el primero indica si es subida o bajada de categoría (1 subida , 2 bajada), y el segundo parámetro indica el valor de la subida o bajada de categoría.
10. Crea un procedimiento **que calcule y devuelva cuántos conductores han realizado más de N trabajos**. El procedimiento tendrá un solo parámetro (N) donde se le pasa el número de trabajos y devuelve el número de conductores que lo cumplen.
11. REALIZAR TODOS Y CADA UNO DE LOS EJERCICIOS ANTERIORES HACIENDO USO DE FUNCIONES EN LUGAR DE PROCEDIMIENTOS.
12. Crea un procedimiento que dé de **baja las dos máquinas más caras.**
13. Crea un procedimiento que **suba de categoría a todos los conductores que han realizado más trabajos que el conductor *"C05"***. La subida será de 2 puntos para los conductores de Rivas y de 1 punto para los de Arganda, para el resto 0.

BD: *Gestión de HOSPITALES*

14. Desarrollar un procedimiento que **visualice** el *apellido* y la *fecha de alta* de todos los empleados ordenados por apellido.
15. Sobre la tabla EMP30 insertar un nuevo 'SALESMAN'. Su número de empleado será consecutivo al mayor de los existentes. Su departamento será el mismo que el de su jefe 'CLARK'.

2.2.2. CURSORES

1. (BD Hospitales): Codificar un programa que devuelva, a través de la tabla DosMG (la cual se crea antes de lanzar el procedimiento y se muestra su contenido después de ejecutar el procedimiento) **los dos empleados que ganan menos de cada oficio**.

2. Sacar el resultado de una consulta con un formato:

3. A partir de la tabla PROVEEDORES generar la tabla LISTADO:

PROVEEDORES			
CODP	**NombreP**	**PoblacionP**	**CiudadP**
P1	Juan	Sanse	MADRID
P2	Sergio	Camas	SEVILLA
P3	Antonio	Alcobendas	MADRID
P4	Fernando	Toro	ZAMORA
P5	Pedro	Trabazos	ZAMORA

LISTADO
Proveedores de MADRID
--- Antonio
--- Juan
Proveedores de SEVILLA
--- Sergio
Proveedores de ZAMORA
--- Fernando
--- Pedro

4. Escriba un bloque que recupere todos los proveedores por países. El resultado debe almacenarse en una nueva tabla *Tabla_Aux* que permita almacenar datos del tipo:
   - Proveedor: ONCE – País: España
   - Utiliza un cursor para recuperar cada país de la tabla *Tabla_Proveedores* y pasar dicho país a un cursor que obtenga el nombre de los proveedores en él. Una vez que se tiene el nombre del proveedor y su país, debe añadirse a la nueva tabla en el formato especificado.

   NOTA: En primer lugar, debe crear las tablas *Tabla_Proveedores* y *Tabla_Aux* e insertar en las mismas datos de prueba.