INDICE

1. Introducción

NOTA: Para probar los ejemplos siguientes utilizaremos la base de datos "*Kadoo*".

La aparición de los programas almacenados (procedimientos, funciones y triggers o disparadores) en MySQL han supuesto una enorme revolución en este gestor de bases de datos que ya disfrutaba de una gran popularidad. Se ha producido el salto para que MySQL pueda ser utilizado como SGBD empresarial. A ello hay que añadir que la sintaxis de los programas es sencilla, lo que facilita su escritura.

Un **programa almacenado** es un conjunto de instrucciones almacenadas dentro del servidor de bases de datos que se ejecutan en él y que está identificado por un nombre.

Tipos de programas almacenados:

- **Procedimientos**[1]**:** El más común de los programas almacenados. Resuelven un determinado problema cuando son invocados. Pueden aceptar parámetros de entrada y devolver parámetros de salida.
- **Funciones**[2]: Similares a los procedimientos, salvo que sólo devuelven un valor como parámetro de salida. La ventaja que presentan las funciones es que pueden ser utilizadas dentro de instrucciones SQL y por tanto aumentan considerablemente las capacidades de este lenguaje.

1 son rutinas o subprogramas compuestos por un conjunto nombrado de sentencias SQL agrupados lógicamente para realizar una tarea específica, que se guardan en la base de datos y que se ejecutan como una unidad cuando son invocados por su nombre.

2 conjunto de instrucciones SQL que después de ejecutarse devuelven un valor.

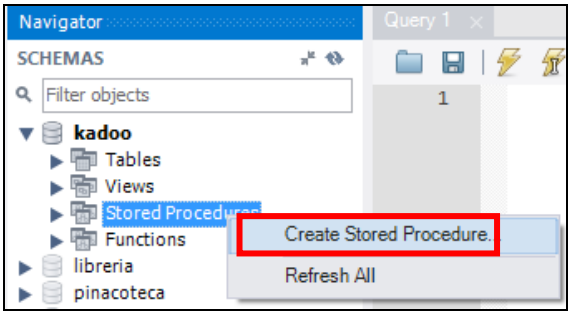
- **Triggers, desencadenadores o disparadores[3]:** Son programas que se activan ("disparan") ante un determinado suceso ocurrido dentro de la base de datos.

2. Ventajas de su utilización

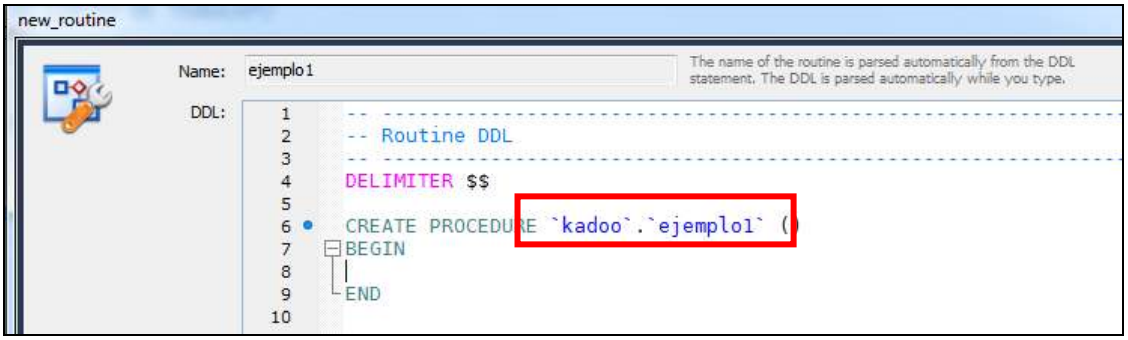
- ✓ **Mayor seguridad y robustez en la base de datos.** Al permitir que los usuarios puedan ejecutar diferentes programas (para los que estén autorizados), se está limitando e impidiendo el acceso directo a las tablas donde están almacenados los datos evitando la manipulación directa de éstas por parte de los usuarios y por tanto eliminando la posibilidad de pérdida accidental de los datos. Los programas serán los que accederán a las tablas.
- ✓ **Mejor mantenimiento de las aplicaciones que acceden a los programas almacenados** y, por tanto, disminución de la posibilidad de aparición de errores. En lugar de que cada aplicación cliente disponga de sus propios programas para realizar operaciones de inserción de consulta o actualización, los programas almacenados permiten centralizar los métodos de acceso y actualización anteriores presentando una interfaz común para todos los programas.
- ✓ **Mayor portabilidad de las aplicaciones** (relacionada con el punto anterior), puesto que la lógica de la aplicación ya queda implementada en los programas almacenados, permitiendo al programador centrarse sólo en su interfaz.
- ✓ **No se necesita de ningún tipo de conector o driver**. Bastará agrupar estas últimas bajo un programa almacenado que se llamará en el momento en el que se necesite.
- ✓ **Reducción del tráfico de red**. El cliente llama a un procedimiento del servidor enviándole unos datos. Este los recibe y tras procesarlos devuelve unos resultados. Por la red no viajan nada más que los datos. En contrapartida señalar que se produce una carga más elevada en el servidor, mucho más que si las aplicaciones se ejecutaran en los clientes, pero hoy en día no supone mucho inconveniente con la tecnología actual de servidor de que se dispone.

3. Edición de programas

Para gestionar (crear, modificar o eliminar) los programas almacenados seguiremos utilizando la herramienta MySQL Workbench. Vamos a crear nuestro primer programa utilizando esta herramienta.

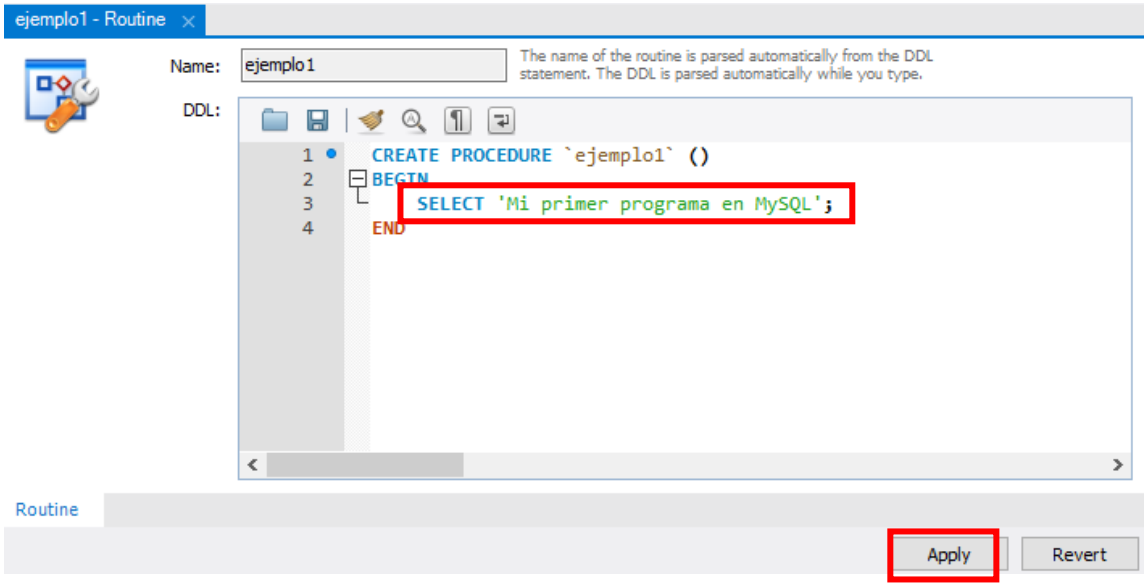


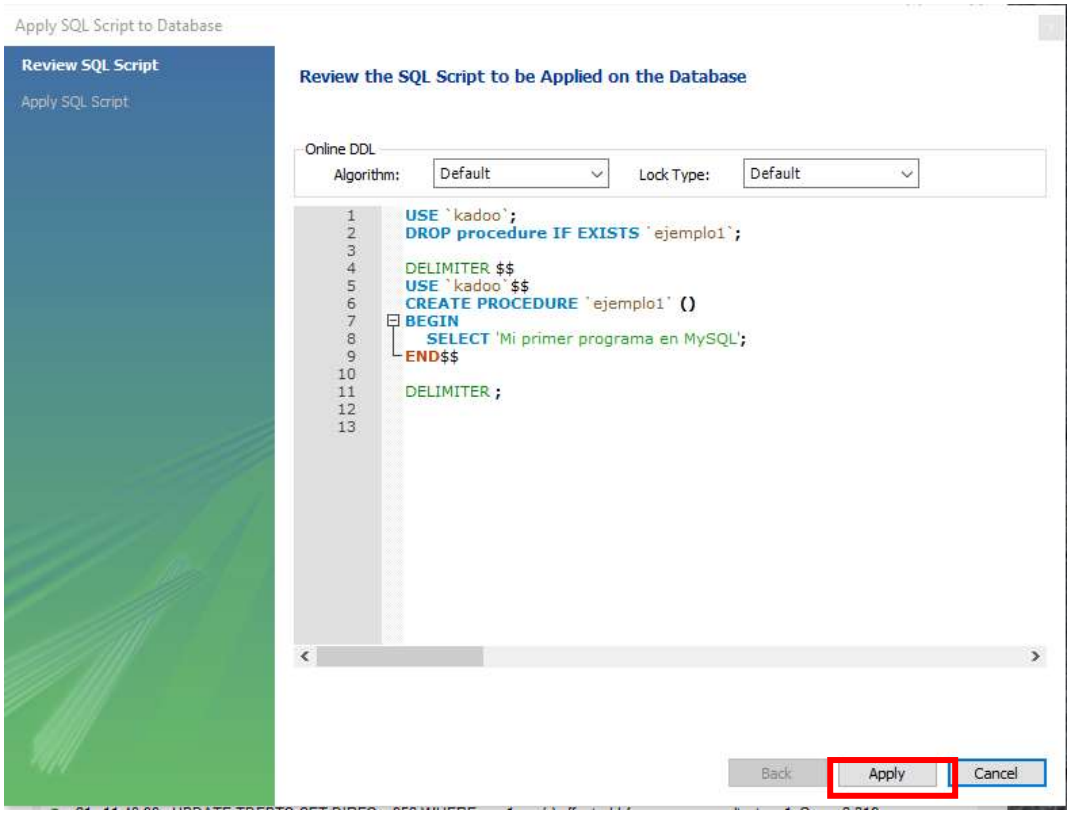
Poner un nombre al procedimiento, en nuestro caso, '***ejemplo1***'

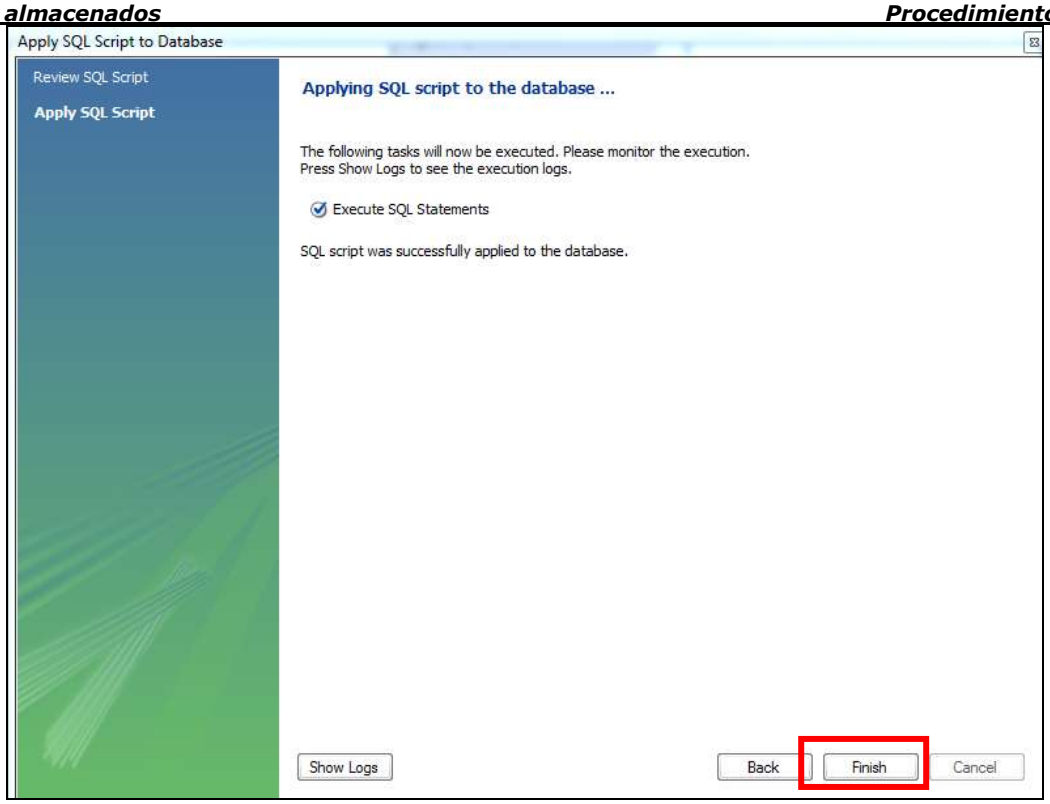


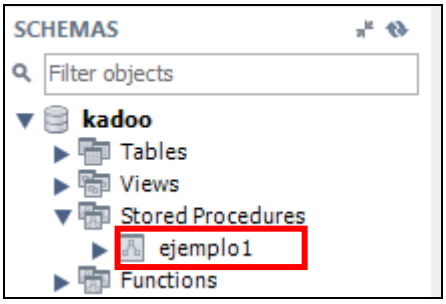
3 rutina asociada con una tabla, que se activa o ejecuta automáticamente cuando se produce algún evento sobre la tabla.

Escribir la instrucción `'select 'Mi primer programa en MySQL'`, entre las etiquetas BEGIN … END

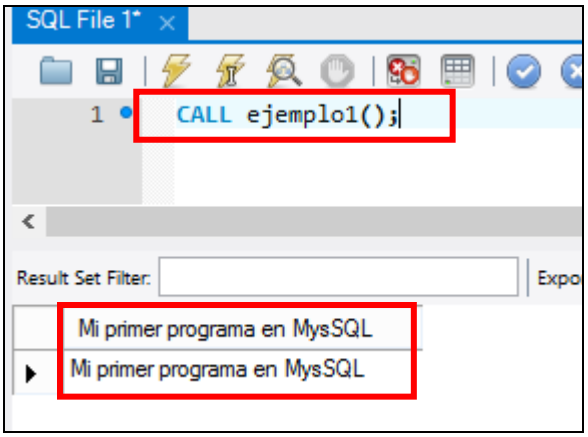








Para ejecutar o invocar el procedimiento utilizamos el comando CALL.



4. Fundamentos básicos del lenguaje

4.1. Variables, tipos de datos, comentarios y literales

Las **variables** son elementos de datos con un nombre cuyo valor puede ir cambiando a lo largo de la ejecución del programa.

SINTAXIS:

```
DECLARE nombre_variable1 [,nombre_variable2...] tipo [DEFAULT valor];
```

La declaración de variables se realiza antes del comienzo de las instrucciones. De la sintaxis anterior se deduce que:

- ✓ Se pueden declarar varias variables del mismo tipo seguidas y separadas por comas.
- ✓ Al mismo tiempo que se declaran las variables se pueden definir asignándoles un valor inicial mediante la cláusula **DEFAULT**. Si no se les asigna ninguno valor, las variables quedan definidas al valor NULL.
- ✓ En el momento de la declaración hay que indicar el tipo de datos de la variable. Pueden utilizarse cualquiera de los tipos de datos que se emplean en la creación de tablas.

TIPOS DE DATOS NUMÉRICOS	
INT, INTEGER	Entero. Los valores pueden ir desde -2147483648 a 2147483647 para enteros con signo o desde 0 a 4294967295 para enteros sin signo.
TINYINT	El más pequeño de los enteros. Rango entre -128 y -127 con signo, o de 0 a 255 sin signo.
SMALLINT	Entero entre -32768 y 32767 (valores con signo) o entre 0 y 65535 (valores con signo).
MEDIUMINT	Entero de valores con signo que van de -8388608 a 8388607 o para valores sin signo de 0 a 16777215.
BIGINT	Entero grande. Con signo puede tomar valores desde -9223372036854775808 a 9223372036854775807 y sin signo de 0 a 18446744073709551615.
FLOAT	Real de precisión simple. Permite almacenar números de -1.7E38 a 1.7E38 con signo o de 0 a 3.4E38 para valores sin signo.
DOUBLE	Real de precisión doble. Puede llegar a alcanzar valores de 0 a 1.7E308 para números sin signo.
DECIMAL(precisión, escala) NUMERIC(precisión, escala)	Equivalen al tipo DOUBLE pero se diferencian en que ocupan bastante mayor espacio (por almacenar valores exactos y no aproximados). Si el número de decimales es importante (cantidades monetarias) es mejor utilizar el tipo NUMERIC. Precisión indica el número de dígitos totales, escala es el número de decimales a la derecha de la coma del total de dígitos que viene expresado en la precisión.

TIPOS DE DATOS DE TEXTO	
CHAR(longitud)	Cadenas de texto de longitud fija hasta un máximo de longitud de 255 caracteres. Si el valor a almacenar es más corto que la longitud de la variable el resto de caracteres se rellenan a blancos.
VARCHAR(longitud)	Cadenas de texto de longitud variable hasta un máximo de 64 KB. A diferencia del tipo CHAR, si el valor a almacenar es más corto, el tamaño real de la variable es el número de caracteres que ocupa el valor pues no rellena a blancos. Como almacena la longitud junto con los caracteres, su utilización en los programas hace que le ejecución de éstos sea un poco más lenta que si se utiliza el tipo CHAR.
ENUM	Almacena un valor concreto de un conjunto posible de valores.
SET	Similar a ENUM pero permite guardar más de un valor.
TEXT	Texto de hasta 64 KB. de tamaño.
LONGTEXT	Texto de hasta 4 GB. de tamaño.

TIPOS DE DATOS DE FECHA Y HORA	
DATE	Fechas con el formato AAAA-MM-DD entre 1000-01-01 y 9999-12-31.
DATETIME	Fecha y hora con el formato AAAA-MM-DD hh:mm:ss. Para la parte de la hora el rango debe estar entre 00:00:00 y 23:59:59.

OTROS TIPOS DE DATOS	
BLOB	Hasta 64KB. de datos binarios.
LONGBLOB	Hasta 4GB. de datos binarios.

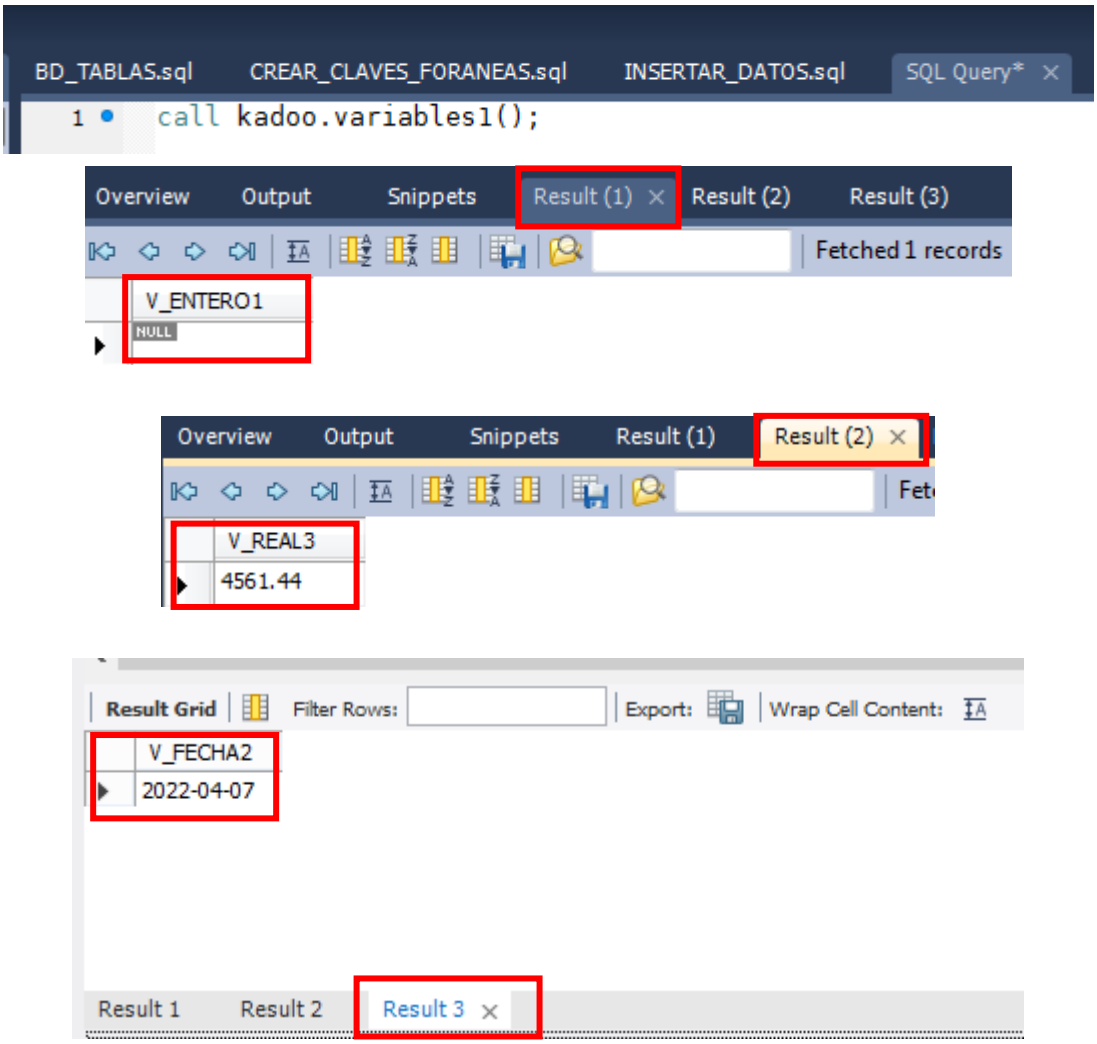
A continuación un ejemplo de declaración de variables, el valor de algunas de ellas se visualiza en las líneas 23, 24 y 25:

variables1

Limit to 1000 rows

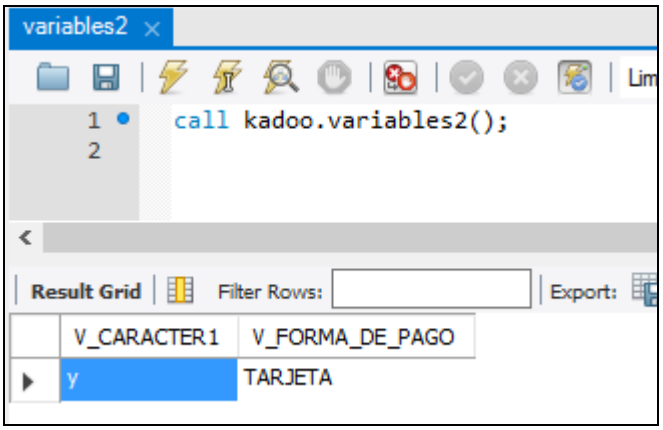
```
1  -- ----------------------------------------------------
2  -- Routine DDL
3  -- Note: comments before and after the routine body will
4  -- ----------------------------------------------------
5  DELIMITER $$
6
7  CREATE  PROCEDURE `variables1`()
8  BEGIN
9      DECLARE V_ENTERO1,V_ENTERO2,V_ENTERO3 INT;
10     DECLARE V_ENTERO4 INT DEFAULT -40000;
11     DECLARE V_ENTERO5 INT DEFAULT 400000000;
12
13     DECLARE V_REAL1 FLOAT DEFAULT 344.67;
14     DECLARE V_REAL2 FLOAT DEFAULT 1.5E14;
15     DECLARE V_REAL3 NUMERIC(7,2) DEFAULT 4561.44;
16
17     DECLARE V_CARACTER1 CHAR(1) DEFAULT 'Y';
18     DECLARE V_CARACTER2 VARCHAR(20);
19
20     DECLARE V_FECHA1 DATE DEFAULT '1966-11-03';
21     DECLARE V_FECHA2 DATE DEFAULT CURRENT_DATE;
22
23 SELECT V_ENTERO1;
24 SELECT V_REAL3;
25 SELECT V_FECHA2;
26
27
28
29
30
31
32 END
```

EJECUTAMOS



Otro ejemplo de declaración de variables. Utilizamos la sentencia de asignación **SET** para asignar valores a variables, como se verá más adelante:

```
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `variables2`()
BEGIN

-- Declaración de Variables
    DECLARE V_CARACTER1 CHAR(1);
    DECLARE V_FORMA_DE_PAGO ENUM('METALICO','TARJETA','TRANSFERENCIA');

-- Asignación de valores a las variables
    set v_caracter1= 'y';
    SET V_FORMA_DE_PAGO = 'TARJETA';

/* Mostrar el valor de las variables*/
    SELECT V_CARACTER1, V_FORMA_DE_PAGO;


END
```



De las imágenes anteriores se observa:

- Aparecen dos tipos de **COMENTARIOS**:
  - **Comentarios de una sola línea**, precedidos de **--**
  - **Comentarios de varias líneas**, entre **/* */**, aunque en este caso solo se extiende el comentario a lo largo de una sola línea.
- Los LITERALES de texto y fecha van encerradas entre ‘ ’.

Las reglas para nombrar variables son bastante flexibles pues a diferencia de otros lenguajes se permite:

1. Nombres largos (más de 255 caracteres).
2. Caracteres especiales.
3. Pueden comenzar con caracteres numéricos.
4. Deben de ser escalares, es decir, de un solo valor, a diferencia de otros lenguajes que permiten definir variables basadas en tipos de datos compuestos como son los registros, arrays…

Para asignar valores a variables se utiliza la siguiente sintaxis:

```
SET nombre_variable1 = expresión1 [,nombre_variable2 = expresión2 ...]
```

En este caso y a diferencia de otros lenguajes es necesario especificar la sentencia **SET** para asignar valores a las variables. Se puede en una sola instrucción realizar varias asignaciones:

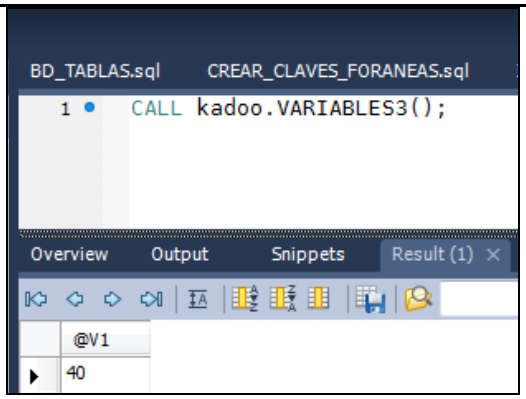
```
1  -- -----------------------------------------------------------------------
2  -- Routine DDL
3  -- -----------------------------------------------------------------------
4  DELIMITER $$
5
6  CREATE DEFINER=`root`@`localhost` PROCEDURE `asignal`()
7  BEGIN
8  DECLARE V_ENTERO1, V_ENTERO2, V_ENTERO3 INT;
9  DECLARE V_ENTERO4 INT DEFAULT -40000;
10 DECLARE V_ENTERO5 INT UNSIGNED DEFAULT 4000000000;
11 DECLARE V_REAL1 FLOAT DEFAULT 344.67;
12 DECLARE V_REAL2 FLOAT DEFAULT 1.5E14;
13 DECLARE V_REAL3 NUMERIC(7,2) DEFAULT 4561.44;
14 DECLARE V_CARACTER1 CHAR(1) DEFAULT 'Y';
15 DECLARE V_CARACTER2 VARCHAR(20);
16 DECLARE V_FECHA1 DATE DEFAULT '1966-11-03';
17 DECLARE V_FECHA2 DATE DEFAULT CURRENT_DATE;
18
19 SET V_CARACTER1 = 'N';
20 SET V_ENTERO1 = V_ENTERO4 + 10000;
21 SET V_REAL2 = V_ENTERO4 + V_REAL3;
22 SET V_FECHA1 = '2013/05/08', V_FECHA2 = V_FECHA2 + 1;
23
24 SELECT V_CARACTER1;
25 SELECT V_ENTERO1;
26 SELECT V_REAL2;
27 SELECT V_FECHA1;
28 SELECT V_FECHA2;
29
30 END
```

4.2. Variables de usuario

Un tipo especial de variables son las **variables de usuario o sesión**, ya que pueden ser manipuladas dentro y fuera de los programas almacenados. Son una característica de MySQL desde la versión 3. Ejemplo de utilización:

VARIABLES3 — The name of the routine is parsed automatically from the DDL statement. The DDL is parsed automatically while you type.

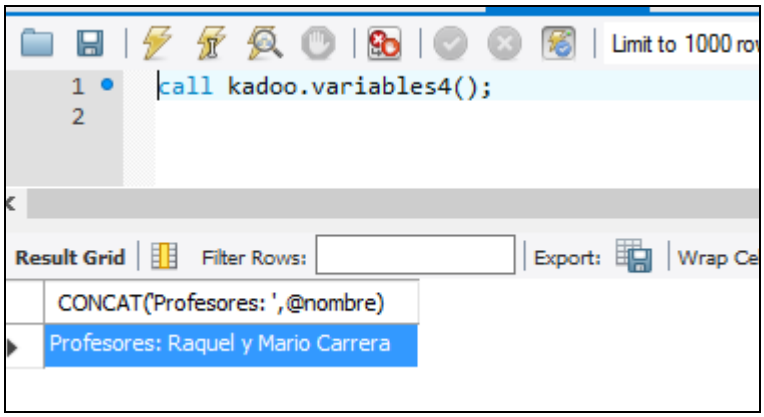
```
1  -- -----------------------------------------------------------------------
2  -- Routine DDL
3  -- -----------------------------------------------------------------------
4  DELIMITER $$
5
6  CREATE DEFINER=`root`@`localhost` PROCEDURE `VARIABLES3`()
7  BEGIN
8
9      SET @V1 = 20;
10     SET @V1  = @V1 * 2;
11
12     SELECT @V1;
13 END
14
```



Su **alcance es a nivel de sesión** y por tanto **son accesibles desde cualquier programa que se ejecuta durante esa sesión**, lo que las **asemeja** al concepto de **variables globales** como se muestra en el siguiente ejemplo:

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `variables4`()
BEGIN

    DECLARE  NUMERO INT DEFAULT 0; -- Variable local
    SET @nombre = 'Raquel'; -- Variable de usuario ó global
    CALL variables5();

    SELECT CONCAT('Profesores: ',@nombre) as Profesores;


END
```

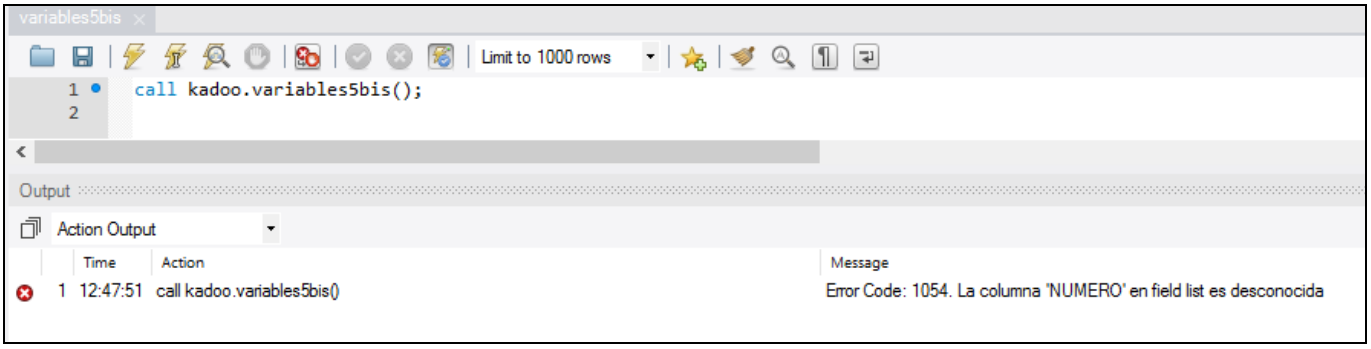
```
-- --------------------------------------------------------------------------------
-- Routine DDL
-- Note: comments before and after the routine body will not be stored by the server
-- --------------------------------------------------------------------------------
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `variables5`()
BEGIN

    SET @nombre = CONCAT(@nombre, ' y Mario Carrera');

END
```



En este ejemplo, hemos añadimos a la función CONCAT, la variable local NUMERO definida en el procedimiento 'variables4 ()'.

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `variables5bis`()
BEGIN
SET @nombre = CONCAT(@nombre, ' y Mario Carrera',NUMERO);
END
```

Ejecutamos el procedimiento 'variables5bis()', observamos que produce un error, la variable 'NUMERO' es local y sólo es visible dentro del procedimiento donde está definida, es decir, desde el procedimiento 'variables4()'



4.3. Parámetros

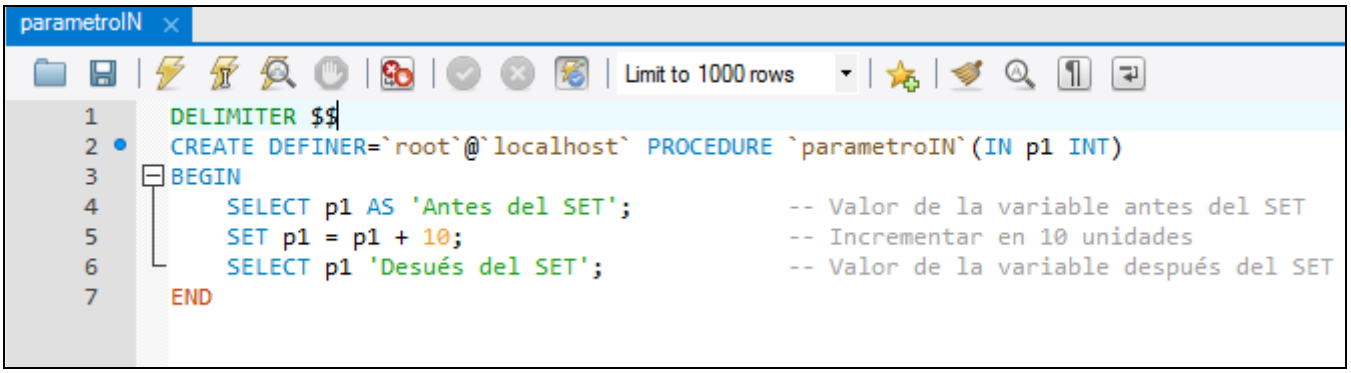
Los **parámetros o argumentos** son variables que reciben y envían los programas a los que se invocan.

Se definen en la cláusula **CREATE** de creación de los procedimientos de la siguiente forma:

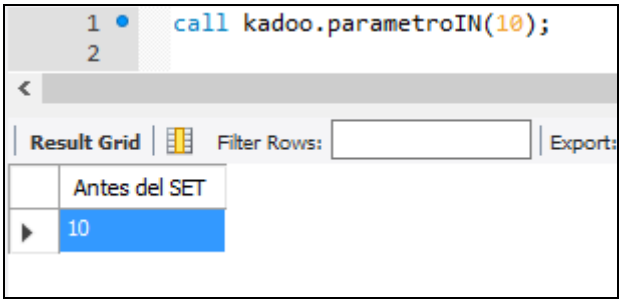
```
CREATE PROCEDURE([[IN |OUT |INOUT] nombre_parámetro tipo de datos...])
```

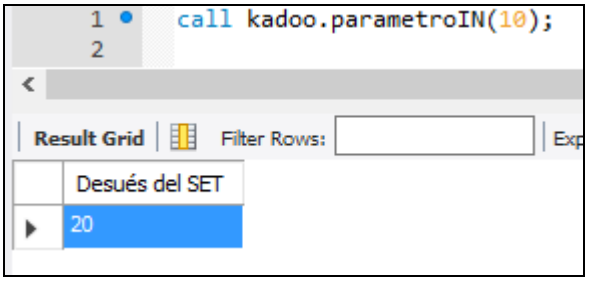
Los tipos de parámetros, según el modo en que se pasan al procedimiento llamado son:

1. **IN**: Opción por defecto. En otros **lenguajes representa el modo de paso de parámetros por valor**, es decir, el procedimiento trabaja con una "**copia**" del parámetro que recibe y por tanto no modifica el valor del parámetro que se le pasa al programa almacenado.



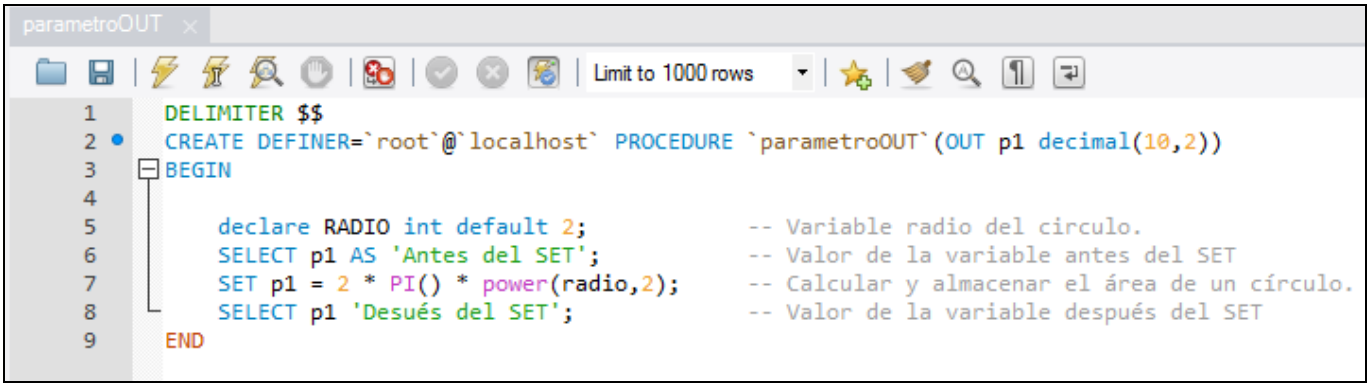
Ejecutar



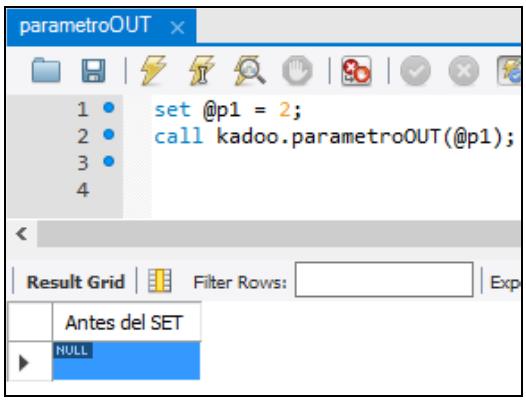


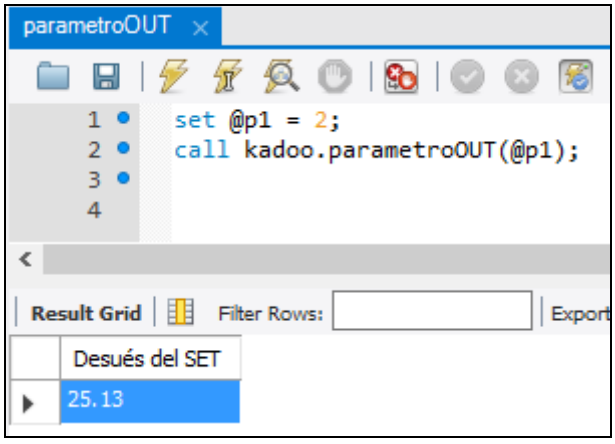
2. OUT: Es una forma de paso de **parámetros por variable**, es decir, las modificaciones del parámetro dentro del programa almacenado modifican directamente el parámetro pasado como argumento. Hasta que no se le asigne un valor determinado dentro del programa, su valor dentro de él será NULL. Se suelen utilizar como flags o indicadores de cómo ha ido la ejecución de un programa, así lo veremos en el apartado de tratamiento de errores.

Ejemplo:



Ejecutar



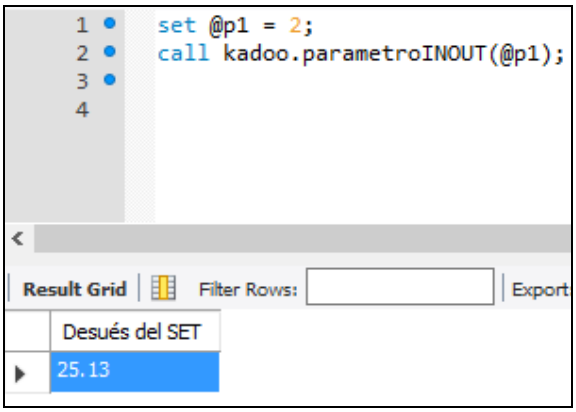


3. INOUT: Otra forma de paso de **parámetros por variable** pero con la característica de que se le puede pasar un valor inicial que el programa llamado tendrá en cuenta (y no lo considerará NULL como en caso del parámetro OUT).

Ejemplo:

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `parametroINOUT`(INOUT p1 decimal(6,2))
BEGIN

    SELECT p1 AS 'Antes del SET';          -- Valor de la variable antes del SET
    SET p1 = 2 * PI() * power(p1,2);       -- Calcular y almacenar el área de un círculo.
    SELECT p1 'Desués del SET';            -- Valor de la variable después del SET
END
```

Ejecutamos



4.4. Principales Operadores

Similares a los de otros lenguajes de programación. **Se utilizan en la mayoría de los casos junto con la sentencia SET para asignar valores a variables, formando parte de expresiones de comparación y en bucles.**

OPERADORES DE COMPARACIÓN.		
Comparan dos valores y devuelven como resultado CIERTO (1), FALSO (0) O NULO (NULL)		
OPERADOR	**SIGNIFICADO**	**EJEMPLO Y RESULTADO**
>	Mayor que	3 > 2 → Cierto
<	Menor que	4 < 6 → Cierto
<=	Menor o igual que	4 <= 3 → Falso
>=	Mayor o igual que	4>=3 → Cierto
=	Igual a	4 = 4 → Cierto
< > !=	Distinto de	4 <> 4 → Falso
<=>	Comparación de valores nulos. Devuelve cierto si ambos valores son nulos	NULL <=> NULL → Cierto
BETWEEN	Comprendido entre dos valores	45 BETWEEN 25 AND 50 → Cierto
IS NULL	Si valor nulo	3 IS NULL → Falso
IS NOT NULL	Si valor no nulo	3 IS NOT NULL → Cierto
NOT BETWEEN	No comprendido entre dos valores	45 NOT BETWEEN 25 AND 50 → Falso
IN	Pertenencia al conjunto o lista	45 IN (44, 45, 46) → Cierto
NOT IN	No pertenencia al conjunto o lista	45 NOT IN (44, 45, 46) → Falso
LIKE	Coincidencia con patrón de búsqueda	"ALBERTO CARRERA" LIKE "%CARRERA" → Cierto

OPERADORES MATEMÁTICOS	
+	Suma
-	Resta
*	Multiplicación
/	División
DIV	División entera
%	Resto

```
DELIMITER $$

DROP PROCEDURE IF EXISTS operadores1 $$
CREATE PROCEDURE operadores1 ()
BEGIN
   DECLARE a INT DEFAULT 2 ;
   DECLARE b INT DEFAULT 256 ;
   DECLARE c FLOAT DEFAULT 440.56 ;
   --
   SET c = c + b / a ;
   --
   SELECT c;        --   568.56
   SELECT c/2*3;    --    852.83999633789
   SELECT b%a;      -- 0
END $$
DELIMITER ;
```

Procedimiento 11 operadores1

TABLA DE VERDAD DEL OPERADOR LÓGICO AND			
AND (&&)	**CIERTO**	**FALSO**	**NULL**
CIERTO	CIERTO	FALSO	NULL
FALSO	FALSO	FALSO	NULL
NULL	NULL	NULL	NULL

TABLA DE VERDAD DEL OPERADOR LÓGICO OR			
OR (||)	**CIERTO**	**FALSO**	**NULL**
CIERTO	CIERTO	CIERTO	CIERTO
FALSO	CIERTO	FALSO	NULL
NULL	CIERTO	NULL	NULL

TABLA DE VERDAD DEL OPERADOR LÓGICO NOT			
NOT (!)	**CIERTO / TRUE (1)**	**FALSO / FALSE (0)**	**NULL**
	FALSO (0)	CIERTO (1)	NULL

4.5. Expresiones

Se trata de una combinación de literales, variables y operadores que se evalúan para devolver un valor.

4.6. Funciones incorporadas

En los programas almacenados pueden seguirse utilizando la mayoría de las funciones incluidas en MySQL y que se utilizan para formar las sentencias SQL, excepto las que trabajan con grupos de datos (cláusula GROUP BY) puesto que las variables en los programas almacenados son escalares y almacenan un solo valor. Por eso funciones como SUM, COUNT, MIN, MAX y AVG pueden emplearse en programas almacenados siempre y cuando devuelvan una fila y no varias (como consecuencia p.ej. en este último caso de utilizar la cláusula GROUP BY).

FUNCIONES MATEMÁTICAS		
FUNCIÓN	**DEVUELVE**	**EJEMPLO Y RESULTADO**
ABS (num.)	Valor absoluto de num.	SELECT ABS(-3) → 3
SIGN (num.)	-1, 0 o 1 en función del valor de num.	SELECT SIGN(2), SIGN(-2), SIGN(0) → 1, -1, 0
MOD(num1, num2)	Resto de la división de num1 por num2	SELECT MOD (5,2) → 1
FLOOR (num.)	Mayor valor entero inferior a num.	SELECT FLOOR(23.9) → 23
CEILING (num.)	Menor valor entero superior a num.	SELECT CEILING(23.9) → 24
ROUND (num.)	Redondeo entero más próximo	SELECT ROUND(23.5), ROUND(23.4); → 24 23
ROUND(num., d)	Redondeo a decimales más próximo	SELECT ROUND(23.545,2), ROUND(23.44,1) → 23,55 23,4
TRUNCATE (num., d)	Num. truncado a decimales	SELECT TRUNCATE (22.89, 1), TRUNCATE (15326,-3) → 22,8 5000
POW(num1, num2)	Num1 elevado a la num2 potencia	SELECT POW(2,5) → 32
SQRT (num.)	Raíz cuadrada de num.	SELECT SQRT(36) → 6

FUNCIONES DE CADENA		
FUNCIÓN	**DEVUELVE**	**EJEMPLO Y RESULTADO**
LIKE(plantilla)	Resultado de comparar una cadena con una plantilla	SELECT 'ALBERTO' LIKE 'ALBER%' → 1 (cierto)
NOT LIKE (plantilla)	Lo contrario a la fila anterior	SELECT 'ALBERTO' NOT LIKE 'ABIERTO' → 1
_ (subrayado)	Se trata de un comodín que reemplaza un carácter en una cadena	SELECT 'ALBERTO' LIKE 'ALBERT_' → 1
%	Como el caso anterior pero para uno o más caracteres	SELECT 'ALBERTO' LIKE 'ALBER%' → 1 (cierto)
\	Como en otros lenguajes se trata del carácter de escape, si precede al comodín elimina su función y lo trata como un carácter más	SELECT '30%' LIKE '30\%' → 1

FUNCIONES DE CADENA (*Continuación*)		
BINARY	Por defecto, en las comparaciones entre cadenas no se distingue mayúsculas de minúsculas salvo que se indique esta opción	SELECT 'ALBERTO' LIKE BINARY 'Alberto' → 0 (falso)
STRCMP(cad1, cad2)	-1 si cad1 < cad2, 0 si cad1=cad2 o 1 si cad1 > cad2	SELECT STRCMP('ALBERTO', 'ABIERTO') → 1
UPPER(cad)	La cadena cad en mayúsculas	SELECT UPPER('Alberto') → ´ALBERTO'
LOWER(cad)	La cadena cad en minúsculas	SELECT LOWER('Alberto') → 'alberto'

FUNCIONES DE FECHA		
FUNCIÓN	**DEVUELVE**	**EJEMPLO Y RESULTADO**
NOW()	Fecha y hora según el formato 'aaaa-mm-dd hh:mm:ss'	SELECT NOW() → 2006-08-01 00:40:25
DAYOFWEEK(fecha)	Cifra que representa el día de la semana (1 – domingo, 2 –lunes…)	SELECT DAYOFWEEK('1966-11-03') → 5 |
WEEKDAY(fecha)	Ídem de DAYOFWEEK pero con otros valores: 0 – lunes, 1 – martes…	SELECT WEEKDAY('1966-11-03') → 3
DAYOFMONTH(fecha)	Día del mes (entre 1 y 31)	SELECT DAYOFMONTH('1966-11-03') → 3
DAYOFYEAR(fecha)	Día del año (entre 1 y 366)	SELECT DAYOFYEAR('1966-11-03') → 307
MONTH(fecha);	Mes del año (entre 1 y 12)	SELECT MONTH('1966-11-03') → 11
DAYNAME(fecha)	Nombre del día de la fecha	SELECT DAYNAME('1966-11-03') → 'Thursday'
MONTHNAME(fecha)	Nombre del mes	SELECT MONTHNAME('1966-11-03') → 'November'
QUARTER(fecha)	Trimestre del año (entre 1 y 4)	SELECT QUARTER('1966-11-03') → 4
WEEK(fecha [,inicio])	Semana del año (entre 1 y 52). Inicio especifica el comienzo de la semana. Si no se especifica vale 0 (domingo). Para empezar el lunes utilizar el 1	SELECT WEEK('2006-12-20',1) → 51
YEAR(fecha)	Año (entre 1000 y 9999)	SELECT YEAR('2006-12-20') → 2006
HOUR(fecha)	La hora	SELECT HOUR(NOW()) →1
MINUTE(fecha)	Los minutos	SELECT MINUTE(NOW()) → 5
SECOND(fecha)	Los segundos	SELECT SECOND(NOW()) → 58

FUNCIONES DE FECHA *(Continuación)*		
TO_DAYS(fecha)	Número de días transcurridos desde el año 0 hasta la fecha	SELECT TO_DAYS('2006-08-01') → 732889
DATE_ADD(fecha, INTERVAL valor tipo de intervalo	La fecha sumado el intervalo especificado	SELECT DATE_ADD('2006-08-01', INTERVAL 1 MONTH) → '2006-09-01'
DATEDIFF(fecha1, fecha2)	El número de días transcurridos entre fecha1 y fecha2	SELECT DATEDIFF('2006-08-01', '2006-07-26') → 6
CURDATE() CURRENT_DATE()	Fecha actual según el formato 'aaaa-mm-dd'	SELECT CURRENT_DATE() → '2006-08-01'
CURTIME() CURRENT_TIME()	Fecha actual según el formato 'hh:mm:ss'	SELECT CURRENT_TIME() → '01:12:43'
DATE_FORMAT (fecha, formato)	Devuelve la fecha en el formato especificado. Consultar el manual para las posibilidades de la opción formato.	SELECT DATE_FORMAT(NOW(), 'Hoy es %d de %M de %Y') → 'Hoy es 01 de August de 2006'

FUNCIONES DE CONTROL		
FUNCIÓN	**DESCRIPCIÓN**	**EJEMPLO Y RESULTADO**
IF(expr1, expr2, expr3)	Si la expresión expr1 es cierta, devuelve expr2, sino expr3	SET @A=20; SET @B=15; SELECT IF(@A<@B, @A+@B, @A - @B); → 5
IFNULL(expr1, expr2)	Si la expresión expr1 es NULL devuelve expr2, sino expr1	SET @A=20; SELECT IFNULL(@A, 0); → 20
NULLIF(expr1, expr2)	Si la expresión expr1 es igual a expr2, devuelve NULL sino expr1	SET @A=20; SET @B=15; SELECT NULLIF(@B, @A); → 15
CASE valor WHEN comp1 THEN res1 [WHEN comp2 THEN res2] [ELSE reselse] END	Compara el valor con cada una de las expresiones comp. Si se verifica la igualdad entonces devuelve el valor res asociado, en cualquier otro caso devuelve reselse	SELECT CASE WEEKDAY(NOW()) WHEN 5 THEN 'Fin de semana' WHEN 6 THEN 'Fin de semana' ELSE 'No es fin de semana' END;

FUNCIONES DE AGREGACIÓN		
FUNCIÓN	**DEVUELVE**	**EJEMPLO Y RESULTADO**
AVG(columna)	Media de los valores de la columna especificada	SELECT AVG(salario) FROM empleados → 302.9412
COUNT (columna | *)	Número de valores no nulos de la columna (si esta se especifica como argumento). Utilizando el carácter * devuelve el número total de valores incluyendo los nulos	SELECT COUNT(comision) FROM empleados → 14 SELECT COUNT(*) FROM empleados → 34 (luego 20 trabajadores no tienen comisión)
MIN(columna)	Valor mínimo de la columna	SELECT MIN(salario) FROM empleados → 100
MAX(columna)	Valor máximo de la columna	SELECT MAX(salario) FROM empleados → 720
SUM(columna)	Suma de valores contenidos en la columna	SELECT SUM(salario) FROM empleados → 10300

OTRAS FUNCIONES		
FUNCIÓN	**DESCRIPCIÓN**	**EJEMPLO Y RESULTADO**
CAST (expresión AS tipo) CONVERT (expresión, tipo)	Convierte la expresión al tipo indicado	SELECT CONVERT(20060802, DATE) → '2006-08-02'
LAST_INSERT_ID()	Devuelve el valor creado por una columna de tipo AUTO_INCREMENT en la última inserción	SELECT LAST_INSERT_ID() → 0
VERSION()	Devuelve la versión del servidor MySQL	SELECT VERSION() → '5.0.20a-nt'
CONNECTION_ID()	Devuelve el identificador de conexión	SELECT CONNECTION_ID() → 4
DATABASE()	Devuelve la base de datos actual	SELECT DATABASE() → 'test'
USER()	Devuelve el usuario actual	SELECT USER() → root@localhost

5. Bloques de instrucciones.

Hasta ahora hemos estado trabajando con procedimientos con un solo bloque de instrucciones, comenzando con la sentencia **BEGIN** y terminando con la sentencia **END**:

```
CREATE {PROCEDURE | FUNCTION |TRIGGER} nombre_del_programa
  BEGIN
        Instrucciones
  END;
```

Este es el caso más sencillo pues muchos programas almacenados MySQL contienen varios bloques agrupando un conjunto de instrucciones y comenzando cada uno de ellos con la instrucción BEGIN y finalizando con la END. Con la estructura de bloques se consigue reunir las instrucciones en agrupaciones lógicas que realizan una determinada función, como por ejemplo los bloques de los manejadores de errores que se tratan en este curso. También se consigue delimitar el ámbito de las variables, declarándolas y definiéndolas dentro de un bloque interno, de este modo no son visibles fuera de él. En cambio una variable externa al bloque interno será accesible también desde dentro de este último. En el caso en que la variable externa al bloque y la interna tuvieran el mismo nombre, dentro del bloque interno se estará referenciando a la variable interna.

Un bloque no solo agrupa instrucciones sino también otros elementos como:

1. Declaración de variables y condiciones.
2. Declaración de cursores.
3. Declaración de manejadores de error.
4. Código de programa.

En el momento de declararse, para evitar mensajes de error, debe seguirse el orden anterior, comenzando en primer lugar por la declaración de variables.

Como ocurre con otros lenguajes de programación, los bloques pueden etiquetarse. Esta forma de actuar garantiza una fácil lectura del procedimiento y, por tanto, de su mantenimiento y permite abandonar el bloque antes de que este concluya si así fuera necesario (sentencia LEAVE).

```
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `bloque1`()
BEGIN

    DECLARE v_externa VARCHAR(45) DEFAULT 'Variable visible en todo el procedimiento ';
    BEGIN
            DECLARE v_interna VARCHAR(100);
            SET v_interna = 'Variable SÓLO accesible entre las líneas 11-16 ';
            SELECT v_externa;
            SELECT v_interna;
    END;
    SELECT v_interna;

END
```

```
-- ----------------------------------------------------------------------------
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `bloque2`()
BEGIN

    DECLARE v VARCHAR(65) DEFAULT 'Variable visible en todo el procedimiento ';
    BEGIN

            SET v = 'Cambio del valor de la variable "v" en el bloque interno ';

    END;
    SELECT v; -- 'Cambió del valor de la variable en el bloque interno '

END
```

```
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `bloque3`()
BEGIN

    DECLARE v INT DEFAULT 500;
    BEGIN
        DECLARE v INT;
        SET v = 200;
        SELECT v as 'Valor de "v" en el Bloque Interno'; -- 200

    END;
    SELECT v as 'Valor de "v" en el Bloque Externo'; -- 500

END
```

```
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `bloque4`()
BEGIN
    bloque_externo: BEGIN
        DECLARE v INT DEFAULT 100;
        bloque_interno: BEGIN
            IF V < 500 THEN
                LEAVE bloque_interno;
            END IF;
            SELECT 'No llega a ejecutarse esta instrucción';
        END; -- Fin bloque interno
    SELECT 'Fin del bloque externo';

    END; -- Fin bloque externo;

END
```

5.1. Control de Flujo. IF

Igual en su funcionamiento que otros lenguajes de programación. Ejecuta la acción cuya expresión (condición) es cierta (de no ser cierta ninguna entonces ejecuta la acción asociada a la sentencia ELSE).

Sintaxis:

```
IF condición1 THEN instrucciones
        [ELSEIF condición2 THEN instrucciones ....]
        [ELSE instrucciones]
END IF;
```

```
DELIMITER $$

CREATE PROCEDURE `condicionalIF` ()
BEGIN

    DECLARE v_edad TINYINT UNSIGNED DEFAULT 47;
    IF v_edad <=12 THEN
        SELECT 'NIÑ@';
    ELSEIF v_edad <=30 THEN
        SELECT 'JOVEN';
    ELSE
        SELECT 'ADULTO';
    END IF;

END
```

Funciones de Control de Flujo

FUNCION	SINTAXIS	DESCRIPCION
IF	`IF(expr1,expr2,expr3)`	Elección en función de una expresión booleana
IFNULL	`IFNULL(expr1,expr2)`	Elección en función de si el valor de una expresión es NULL
NULLIF	`NULLIF(expr1,expr2)`	Devuelve NULL en función del valor de una expresión

- **IF** *Si la expr1 es TRUE (expr1 <> 0 and expr1 <> NULL) entonces IF() devuelve expr2, en caso contrario, devolverá expr3. IF() devuelve un valor numérico o una cadena, dependiendo del contexto en el que se use.*
- **IFNULL** *Si expr1 no es NULL, IFNULL() devuelve expr1, en caso contrario, devuelve expr2. IFNULL() devuelve un valor numérico o una cadena, dependiendo del contexto en el que se use.*
- **NULLIF** *Devuelve NULL sí expr1 = expr2, si no devuelve expr1.*

5.2. Control de Flujo. CASE

Similar a la sentencia condicional IF anterior. Todo lo que se puede expresar con la sentencia IF se puede expresar con la sentencia CASE. 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 difícil de leer.

Existen 2 formas posibles si lo que se evalúa es una expresión o una condición:

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

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `condicionalCASEExpresiones`()
BEGIN
    DECLARE v_forma_pago ENUM ('metalico','tarjeta','transferencia');
    SET v_forma_pago = 'metalico';
    CASE v_forma_pago
        WHEN 'metalico' THEN
            SELECT 'Forma de pago elegida: METALICO';
        WHEN 'tarjeta' THEN
            SELECT 'Forma de pago elegida: TARJETA';
        ELSE
            SELECT 'Forma de pago elegida: TRANSFERENCIA';
    END CASE;
END
```

La otra sintaxis es similar a la anterior pero con condiciones en lugar de expresiones:

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

```
-- ----------------------------------------------------------------
DELIMITER $$

CREATE PROCEDURE `condicionalCASECondiciones` ()
BEGIN

DECLARE v_edad TINYINT UNSIGNED DEFAULT 7;
CASE
    WHEN v_edad <=12 THEN
        SELECT 'NIÑ@';
    WHEN v_edad <=30 THEN
        SELECT 'JOVEN';
    ELSE -- Equivale a DEFAULT (en otros lenguaje)
        SELECT 'ADULTO';
    END CASE;

END
```

5.3. Bucle LOOP

Sintaxis

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

Todas las instrucciones comprendidas entre las palabras reservadas LOOP Y END LOOP (bucle), se ejecutan un número de veces hasta que la ejecución del bucle se encuentra con la instrucción:

```
LEAVE etiqueta
```

En ese momento se abandona el bucle.

```
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `bucle1`()
BEGIN

-- Uso de LOOP
    DECLARE i TINYINT UNSIGNED;
    SET i = 0;
    mibucle:LOOP
        SET i = i + 1;
        SELECT CONCAT('Valor de i = ',i ) as i;
        IF i = 4 THEN
            LEAVE mibucle;
        END IF;
    END LOOP mibucle;
END
```

La sentencia:

```
ITERATE etiqueta
```

Se utiliza para forzar que la ejecución del bucle termine en el momento donde se encuentra con esta instrucción y continúe por el principio del bucle. De esta manera en el ejemplo que viene a continuación se ejecutará 3 veces la instrucción de la línea 13 (para los valores de i del 1 al 4 excepto el 3):

```
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `bucle2`()
BEGIN

-- Uso de LOOP con LEAVE e ITERATE
    DECLARE i TINYINT UNSIGNED;
    SET i = 0;
    mibucle:LOOP
        SET i = i + 1;
        IF i = 3 THEN
            ITERATE mibucle;
        END IF;
        SELECT CONCAT('Valor de i = ',i ) as i;
        IF i = 4 THEN
            LEAVE mibucle;
        END IF;
    END LOOP mibucle;

END
```

5.4. Bucle REPEAT ... UNTIL

Sintaxis:

```
[etiqueta:] REPEAT
       instrucciones
       UNTIL expresión
 END REPEAT [etiqueta]
```

Las instrucciones se ejecutarán hasta que sea cierta la expresión. Por lo menos el conjunto de instrucciones se ejecuta una vez pues la evaluación de la expresión se hace posterior a la ejecución de las instrucciones. El mismo procedimiento bucle1 anterior pero utilizando esta otra instrucción quedaría:

```
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `bucle3`()
BEGIN

-- Uso de REPEAT...UNTIL
    DECLARE i TINYINT UNSIGNED;
    SET i = 0;
    mibucle:REPEAT
        SET i = i + 1;
        /*IF i = 3 THEN
            ITERATE mibucle;
        END IF;*/
        SELECT CONCAT('Valor de i = ',i ) as i;
    UNTIL i = 4
    END REPEAT mibucle;


END
```

Equivale a la sentencia iterativa vista anteriormente:

```
etiqueta: LOOP
        instrucciones
        IF expresión THEN LEAVE etiqueta; END IF;
END LOOP etiqueta;
```

Podría utilizarse la sentencia ITERATE pero puede llevar a situaciones contradictorias si llegara a ejecutar otra vez el conjunto de instrucciones aun habiendo hecha cierta la expresión de finalización del bucle.

La sentencia LEAVE también puede utilizarse.

5.5. Bucle WHILE

```
[etiqueta:] WHILE expresión DO
        instrucciones
END WHILE [etiqueta]
```

```
DELIMITER $$

CREATE PROCEDURE `bucle4` ()
BEGIN

-- Uso de WHILE...DO
    DECLARE i TINYINT UNSIGNED;
    SET i = 0;
    mibucle: WHILE i < 4 DO
        SET i = i + 1;
        SELECT CONCAT('Valor de i = ',i ) as i;

    END WHILE mibucle;

END
```

Equivale a la instrucción iterativa anterior:

```
etiqueta: LOOP
    IF !expresión THEN LEAVE etiqueta; END IF;
    resto de instrucciones;
END LOOP etiqueta;
```

5.6. Bucles anidados

Consiste en utilizar comandos de repetición dentro de otros. Ejemplo de utilización:

```
1  DELIMITER $$
2
3  DROP PROCEDURE IF EXISTS bucle5 $$
4  CREATE PROCEDURE bucle5 ()
5  BEGIN
6     DECLARE i,j TINYINT UNSIGNED DEFAULT 1;
7
8    bucle_externo: LOOP
9       SET j=1;
10      bucle_interno: LOOP
11         SELECT  CONCAT("Valor de i y j:  ",i, "-", j) AS i_j;
12         SET j=j+1;
13         IF j>2 THEN
14             LEAVE bucle_interno;
15         END IF;
16      END LOOP bucle_interno;
17      SET i=i+1;
18      IF i>2 THEN
19         LEAVE bucle_externo;
20      END IF;
21   END LOOP bucle_externo;
22
23 END $$
24
25 DELIMITER ;
```

Procedimiento 25 bucle5

6. Procedimientos

6.1. Creación de procedimientos. Diccionario de Datos

La sintaxis completa de creación de procedimientos es:

```
CREATE PROCEDURE nombre_procedimiento ([parametro1[,...]])
    [LANGUAGE SQL]
    [ [NOT] DETERMINISTIC]
    [ {CONTAINS SQL | MODIFIES SQL DATA | READS SQL DATA | NO SQL} ]
    [SQL SECURITY {DEFINER | INVOKER} ]
    [COMMENT comentario]
    bloque_de_instrucciones_del_procedimiento
```

Para crear un procedimiento o función el usuario debe disponer del privilegio CREATE ROUTINE y el de ALTER ROUTINE para modificarlo o borrarlo; este último privilegio se asigna automáticamente al creador del procedimiento o función. Para poder ejecutar un procedimiento o función debe disponerse del privilegio EXECUTE.

Aspectos a tener en cuenta:

1. El nombre del procedimiento cumple con las mismas normas referidas a nombrar variables. La lista de argumentos suministrados al procedimiento también ha sido estudiada.
2. La cláusula **LANGUAGE SQL** indica que el lenguaje utilizado en los procedimientos cumple el estándar SQL:PSM[4]. Es una cláusula innecesaria en este momento pues los procedimientos en MySQL solo soportan este estándar.
   Si MySQL en un futuro aceptara la escritura de procedimientos almacenados en otros lenguajes como C o Java entonces ya sería necesario indicar el lenguaje de programación utilizado en el procedimiento.
3. **[NOT] DETERMINISTIC**: Referido al comportamiento del procedimiento. Si un procedimiento es DETERMINISTIC, siempre ante una misma entrada devuelve una misma salida. Funciones numéricas como el valor absoluto, cuadrado, raíz cuadrada son DETERMINISTIC pues siempre devuelven el mismo resultado ante un mismo valor. Por otro lado, una función que devolviera el número de días transcurridos desde 1900 hasta la fecha sería NOT DETERMINISTIC pues cambia según el día que se ejecuta. Por defecto la opción es NOT DETERMINISTIC. Al igual que la anterior se puede prescindir de su utilización debido a que por el momento no es tenida en cuenta por el servidor. Más adelante podrá ser utilizada para la optimización de consultas.
4. **[{CONTAINS SQL | MODIFIES SQL DATA | READS SQL DATA | NO SQL}]:** Indica el tipo de acceso que realizará el procedimiento a la base de datos, si solo se va a leer datos se especificará la cláusula READ SQL DATA, si además de ello los modifica entonces la cláusula MODIFIES SQL DATA será la que se emplee. Si el procedimiento no realiza ningún tipo de acceso a la base de datos puede utilizarse la cláusula NO SQL. Por defecto se utiliza la opción CONTAINS SQL que indica que el procedimiento contiene consultas SQL. Estos parámetros se utilizan para mejorar el rendimiento.
5. **[SQL SECURITY {DEFINER | INVOKER}]:** Indica sí el procedimiento almacenado se ejecutará con los permisos del usuario que lo creó (DEFINER) o con los permisos del usuario que invoca al procedimiento (INVOKER). La opción por defecto es DEFINER. Hay que tener muy en cuenta que con la opción DEFINER un usuario que lance el procedimiento podrá acceder a los datos aunque no posea los privilegios sobre las tablas que almacenan dichos datos; se trata de un mecanismo de acceso a los procedimientos sin dar directamente acceso a los datos.

[4] **SQL/PSM (SQL Módulos Persistentes Almacenados)**

6. **[COMMENT comentario]:** Comentario sobre el procedimiento que puede ayudar al usuario a conocer/recordar el funcionamiento del procedimiento. Esta información puede consultarse como se ha visto anteriormente en el diccionario de datos. Puede prescindirse de dicha cláusula y realizar la escritura de los comentarios aclaratorios al principio del bloque de instrucciones del procedimiento utilizando los caracteres /* */ o --.

¿Dónde se almacenan los programas almacenados?

Para conocer toda la información sobre los procedimientos almacenados se procederá a consultar el diccionario de datos a través de la BD '***information_schema***' y la tabla '***routines***'. Se incluye también información para las funciones que se tratan más adelante.

```
1  select * from information_schema.routines
2  where routine_schema like 'kadoo';
3
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

SPECIFIC_NAME	ROUTINE_CATALOG	ROUTINE_SCHEMA	ROUTINE_NAME	ROUTINE_TYPE
bloque1	def	kadoo	bloque1	PROCEDURE
bloque2	def	kadoo	bloque2	PROCEDURE
bloque3	def	kadoo	bloque3	PROCEDURE
bloque4	def	kadoo	bloque4	PROCEDURE
bucle1	def	kadoo	bucle1	PROCEDURE
bucle2	def	kadoo	bucle2	PROCEDURE
bucle3	def	kadoo	bucle3	PROCEDURE
bucle4	def	kadoo	bucle4	PROCEDURE
condicionalCASECondiciones	def	kadoo	condicionalCASECondiciones	PROCEDURE
condicionalCASEExpresiones	def	kadoo	condicionalCASEExpresiones	PROCEDURE
condicionalIF	def	kadoo	condicionalIF	PROCEDURE

```
SHOW {PROCEDURE | FUNCTION} STATUS [LIKE patrón]
```

```
4  show procedure status where db like 'kadoo';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Db	Name	Type	Definer	Modified	Created	Security_type	Comment
kadoo	bloque1	PROCEDURE	root@localhost	2016-04-20 14:05:57	2016-04-20 14:05:57	DEFINER	
kadoo	bloque2	PROCEDURE	root@localhost	2016-04-20 14:10:34	2016-04-20 14:10:34	DEFINER	
kadoo	bloque3	PROCEDURE	root@localhost	2016-04-20 14:12:13	2016-04-20 14:12:13	DEFINER	
kadoo	bloque4	PROCEDURE	root@localhost	2016-04-20 14:15:34	2016-04-20 14:15:34	DEFINER	
kadoo	bucle1	PROCEDURE	root@localhost	2016-04-20 14:39:10	2016-04-20 14:39:10	DEFINER	
kadoo	bucle2	PROCEDURE	root@localhost	2016-04-20 14:40:33	2016-04-20 14:40:33	DEFINER	
kadoo	bucle3	PROCEDURE	root@localhost	2016-04-20 14:42:58	2016-04-20 14:42:58	DEFINER	
kadoo	bucle4	PROCEDURE	root@localhost	2016-04-20 14:43:24	2016-04-20 14:43:24	DEFINER	
kadoo	condicionalCASECondiciones	PROCEDURE	root@localhost	2016-04-20 14:38:11	2016-04-20 14:38:11	DEFINER	
kadoo	condicionalCASEExpresiones	PROCEDURE	root@localhost	2016-04-20 14:38:33	2016-04-20 14:38:33	DEFINER	
kadoo	condicionalIF	PROCEDURE	root@localhost	2016-04-20 14:25:23	2016-04-20 14:25:23	DEFINER	

Utilizando un patrón de búsqueda:

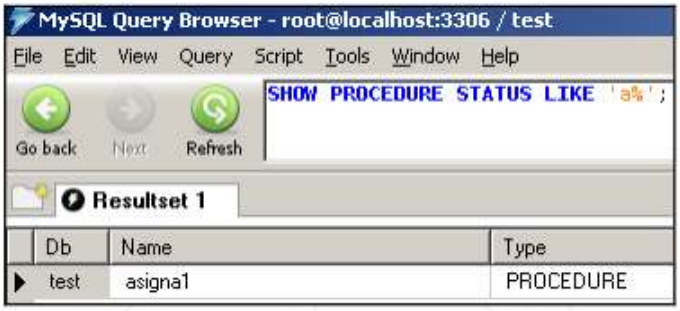


Ilustración 9. Información sobre determinados procedimientos

El comando:

```
SHOW CREATE [PROCEDURE | FUNCTION] nombre
```

Permite ver información de los procedimientos y funciones así como las líneas de código.

6.2. Modificación de procedimientos

Sintaxis:

```
ALTER PROCEDURE nombre_procedimiento
        {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA)
        | SQL SECURITY {DEFINER|INVOKER)
        |COMMENT comentario
```

Se debe poseer el privilegio de ALTER ROUTINE para poder modificar procedimientos y funciones. El uso de las distintas opciones se ha expuesto en el apartado de creación.

6.3. Borrado de procedimientos

Sintaxis:

```
DROP PROCEDURE [IF EXISTS] nombre_procedimiento
```

Al igual que para la modificación, se debe poseer el privilegio de ALTER ROUTINE para poder borrar procedimientos y funciones.

6.4. Utilización de instrucciones DDL y DML en procedimientos almacenados

Junto con la recuperación de datos (se expone en el siguiente punto), se trata de una de las más importantes ventajas del empleo de programas almacenados (en este caso procedimientos).

En el ejemplo siguiente, en las líneas 7 a 11 nos encontramos con instrucciones DDL (lenguaje definición de datos) cuya finalidad es la creación de una tabla. Por otro lado, la instrucción DML (lenguaje de manipulación de datos) de la línea 14 inserta un alumno de código i y nombre "alumno i" cada vez que se ejecuta dicha instrucción dentro del bucle (5 veces en total).

```
1  DELIMITER $$
2
3  DROP PROCEDURE IF EXISTS procedimiento1 $$
4  CREATE PROCEDURE procedimiento1 ()
5  BEGIN
6      DECLARE i TINYINT UNSIGNED DEFAULT 1;
7      DROP TABLE IF EXISTS alumnos ;
8      CREATE TABLE alumnos
9           (id         INT PRIMARY KEY,
10           alumno VARCHAR(30))
11      ENGINE=innodb;
12 --
13     WHILE (i<=5) DO
14          INSERT INTO alumnos VALUES(i,CONCAT("alumno ",i));
15          SET i=i+1;
16     END WHILE;
17
18 END $$
19
20 DELIMITER ;
```

Procedimiento 26 – "procedimiento1"

id	alumno
1	alumno 1
2	alumno 2
3	alumno 3
4	alumno 4
5	alumno 5

6.5. Utilización de instrucciones de consulta en procedimientos almacenados

A diferencia de otros gestores de bases de datos, en MySQL se puede (sólo en procedimientos) devolver como resultado de la ejecución del procedimiento un conjunto de filas. La funcionalidad de esta forma de recuperación se asemeja a la que se puede conseguir con el empleo de vistas.

En el siguiente procedimiento se recupera el conjunto de filas de los departamentos que pertenecen al centro que se le pasa como argumento:

```
1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS resultset1 $$
3 CREATE PROCEDURE resultset1 (IN p_numce INTEGER)
4 BEGIN
5      SELECT numde, nomde, presu
6         FROM departamentos
7      WHERE numce = p_numce;
8 END $$
9 DELIMITER ;
```

Procedimiento 27 resultset1

Ejemplo de ejecución:

```
mysql> CALL test.resultset1(20);
+-------+---------------------+----------+
| numde | nomde               | presu    |
+-------+---------------------+----------+
| 110   | DIRECCION COMERCIAL | 15000000 |
| 111   | SECTOR INDUSTRIAL   | 11000000 |
| 112   | SECTOR SERVICIOS    | 9000000  |
+-------+---------------------+----------+
3 rows in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
```

El inconveniente es que el programa que recibe el conjunto de filas resultantes no puede ser un procedimiento MySQL, sino otro programa escrito en otro lenguaje como Java o PHP. Para salvar este problema y poder enviar a un procedimiento MySQL un conjunto de registros se utilizan las tablas temporales:

```
1  DELIMITER $$
2  DROP PROCEDURE IF EXISTS resultset2 $$
3  CREATE PROCEDURE resultset2 (IN p_numce INTEGER)
4  BEGIN
5       DROP TEMPORARY TABLE IF EXISTS tmp_departamentos;
6       CREATE TEMPORARY TABLE tmp_departamentos AS
7            SELECT numde, nomde, presu
8              FROM departamentos
9            WHERE numce = p_numce;
10 END $$
11 DELIMITER ;
```

Procedimiento 28 resultset2

6.6. Almacenar en variables el valor de la fila de una tabla

No sólo las instrucciones SQL de definición y manipulación de datos (DDL y DML) pueden intercalarse en los procedimientos almacenados, la recuperación de los datos almacenados en la base de datos permitirá su posterior proceso o tratamiento.

El siguiente ejemplo almacena una fila entera de la tabla creada en el procedimiento anterior (tabla alumnos) en las variables declaradas en las líneas 6 y 7, mostrando su contenido en la línea 12. En la línea 8 se señalan las columnas a recuperar (en este caso las dos) y en la línea 9 se indica donde (variables) se guardarán los valores de dichas columnas según el orden en que aparecen (el valor de la columna id se guardará en la variable *v_id*, el de la columna alumno en la variable *v_alumno*).

```
1  DELIMITER $$
2
3  DROP PROCEDURE IF EXISTS procedimiento2 $$
4  CREATE PROCEDURE procedimiento2 ()
5  BEGIN
6      DECLARE v_id INT;
7      DECLARE v_alumno VARCHAR(30);
8      SELECT id, alumno
9        INTO v_id, v_alumno
10       FROM alumnos
11     WHERE id = 4;
12     SELECT v_id, v_alumno;
13 END $$
14
15 DELIMITER ;
```

Procedimiento 29 - "procedimiento2"

En el anterior ejemplo no ha habido ningún problema de ejecución pues cada una de las dos variables almacena solo un ítem (el de la correspondiente columna), pues la consulta solo devuelve una fila.

Pero ¿Qué ocurriría si la consulta devolviera más de una fila?

!	Description	ErrorNr.
!	Result consisted of more than one row	1172

Ilustración 13. Error: La consulta sólo debería devolver una fila

¿Qué ocurriría si no devolviera ningún valor la instrucción SELECT como es el caso del procedimiento4 que viene a continuación?:

```
1  DELIMITER $$
2
3  DROP PROCEDURE IF EXISTS procedimiento4 $$
4  CREATE PROCEDURE procedimiento4 ()
5  BEGIN
6      DECLARE v_id INT;
7      DECLARE v_alumno VARCHAR(30);
8      SELECT id, alumno
9        INTO v_id, v_alumno
10       FROM alumnos
11     WHERE id = 100;
12
13     SELECT v_id, v_alumno;
14 END $$
15
16 DELIMITER ;
```

Procedimiento 30 - "procedimiento4"

v_id	v_alumno
NULL	NULL

Ilustración 14. La consulta no devuelve filas de la tabla

¿Qué ocurriría si la consulta devuelve más de una fila? En ese momento entrarán en escena los CURSORES.