

#### 4.2.1 SINTAXIS Y EJEMPLOS DE RUTINAS ALMACENADAS

Plantilla general para la construcción de guiones en MySQL Workbench.

```
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 }
```

- **sp\_name:** es el nombre de la rutina almacenada.
- **E parameter:** son los parámetros que en general se caracterizan por un tipo y un nombre. El tipo puede ser de entrada (**IN**), de salida (**OUT**) o de entrada/salida (**INOUT**).
- **E routine\_body:** es el cuerpo de la rutina, formado generalmente por sentencias SQL. En caso de haber más de una, deben ir dentro de un bloque delimitado por las sentencias **BEGIN** y **END**, como veremos en los siguientes ejemplos.
- **E Deterministic:** indica si es determinista, es decir, si siempre produce el mismo resultado.
- **E Contains SQL/no SQL:** especifica si contiene sentencias SQL o no.

Ejemplo:

```
USE world;
DELIMITER $$
DROP PROCEDURE IF EXISTS hola_mundo$$
CREATE PROCEDURE world.hola_mundo()
BEGIN
SELECT 'hola mundo';
END$$
```

PROCEDIMIENTO CON EL COMANDO source:

```
mysql> source hola_mundo.slq
Query OK, 0 rows affected, 1 warning (0.01 sec)
Query OK, 0 rows affected (0.00sec)
```

Una vez creado estamos en condiciones de ejecutarlo llamándolo con el comando CALL:

```
mysql> call hola_mundo() $$
+-----+
| Hola mundo |
+-----+
| Hola mundo |
+-----+
```

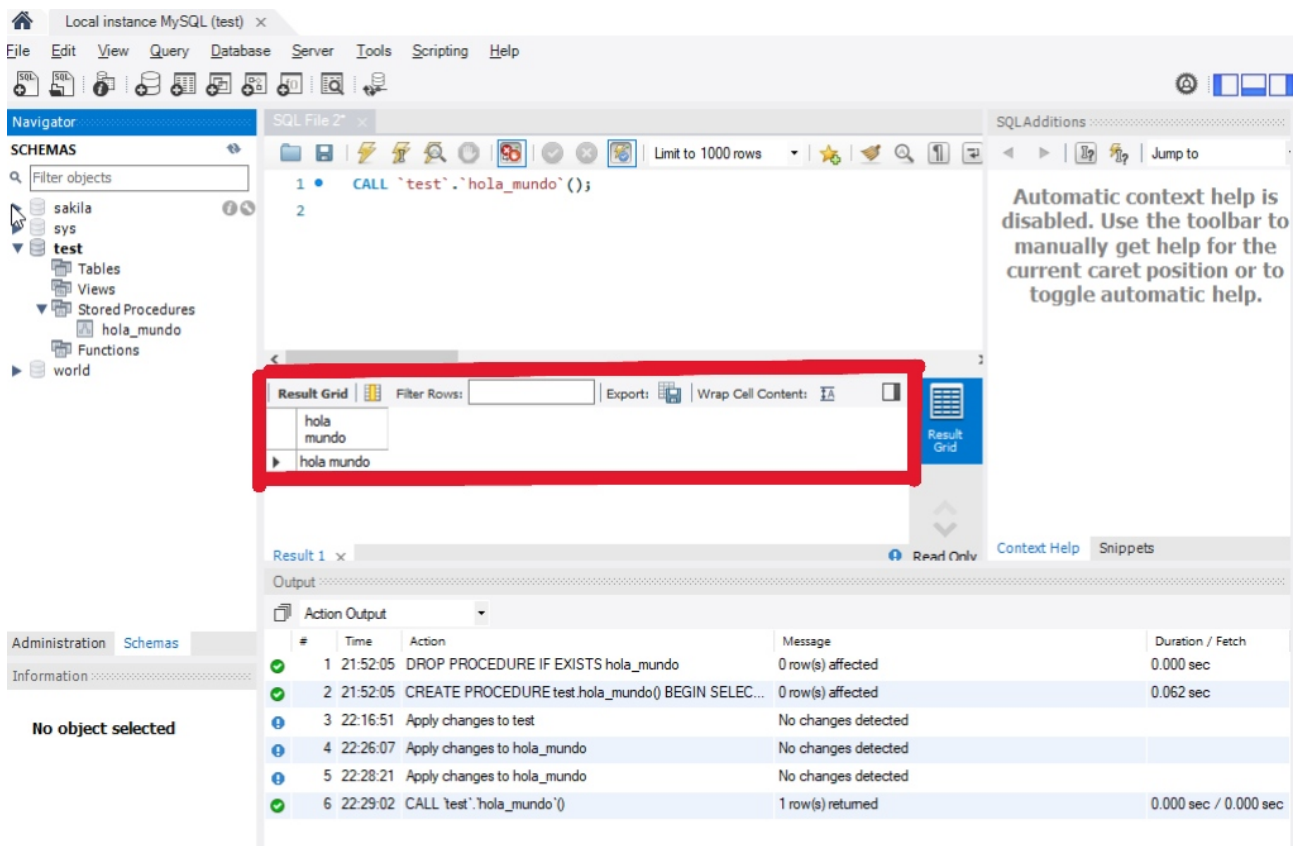
**1 row in set (0.01 sec)**  
**Query OK, 0 rows affected (0.01 sec)**

Si queremos comprobar que el procedimiento existe en la base de datos test usaremos el comando SHOW CREATE

PROCEDURE hola\_mundo.

Para realizar lo mismo usando el Workbench usaríamos un script tab en el que incluiríamos el código del procedimiento, quedaría algo así:

```
Rountine DDL
DELIMITER $$
CREATE DEFINER='root'@'localhost' PROCEDURE 'hola_mundo'()
BEGIN
    SELECT 'hola mundo';
END
```



The screenshot shows the MySQL Workbench interface. The SQL editor contains the following script:

```
1 CALL `test`.`hola_mundo`();
2
```

The output pane shows the results of the script execution:

#	Time	Action	Message	Duration / Fetch
1	21:52:05	DROP PROCEDURE IF EXISTS hola_mundo	0 row(s) affected	0.000 sec
2	21:52:05	CREATE PROCEDURE test.hola_mundo() BEGIN SELEC...	0 row(s) affected	0.062 sec
3	22:16:51	Apply changes to test	No changes detected	
4	22:26:07	Apply changes to hola_mundo	No changes detected	
5	22:28:21	Apply changes to hola_mundo	No changes detected	
6	22:29:02	CALL `test`.`hola_mundo`()	1 row(s) returned	0.000 sec / 0.000 sec

Para ver el estado de nuestras tablas, insertaremos lo siguiente:

**SHOW GRANTS FOR 'root'@'localhost';**

## Ejemplo 4.4

```
USE world;
DELIMITER $$
CREATE FUNCTION estado(in_estado CHAR(1))
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
    DECLARE estado_desc VARCHAR(20);

    IF in_estado = 'P' THEN
        SET estado_desc = 'caducado';
    ELSEIF in_estado = '0' THEN
        SET estado_desc = 'activo';
    ELSEIF in_estado = 'N' THEN
        SET estado_desc = 'nuevo';
    ELSE
        SET estado_desc = 'desconocido'; -- Manejar valores no esperados
    END IF;

    RETURN estado_desc;
END$$
DELIMITER ;
```

Para consultar la tabla:

```
SHOW FUNCTION STATUS WHERE Db = 'world';
```

En este ejemplo la función recibe un valor de estado como entrada y comprueba su valor. Según cual sea se asignará con el comando SET el valor abreviado a la variable estado que es devuelta.

### Propósito

Sirve para **listar las funciones almacenadas** en una base de datos específica (en este caso, la base de datos `world`). Esto te permite obtener información sobre las funciones creadas en dicha base de datos.

### Información que muestra

El comando devuelve una tabla con varias columnas que describen las funciones existentes. Las columnas más relevantes son:

1. **Db**: El nombre de la base de datos donde se encuentra la función.
2. **Name**: El nombre de la función.
3. **Type**: Indica si es una función o un procedimiento.
4. **Definer**: El usuario que creó la función.
5. **Created**: La fecha en que se creó la función.
6. **Modified**: La fecha de la última modificación de la función.
7. **Security\_type**: Define cómo se ejecuta la función (por ejemplo, si usa permisos del invocador o del definidor).

## Uso práctico

Este comando es útil para:

1. **Verificar funciones existentes** en una base de datos antes de crear una nueva.
2. **Identificar posibles duplicados** que puedan causar errores.
3. **Auditar funciones almacenadas**, verificando quién las creó y cuándo.

## Ejemplo de salida

Si ejecutas el comando en una base de datos llamada `world` que contiene una función llamada `estado`, podrías ver algo como esto:

Db	Name	Type	Definer	Created	Modified	Security_type
world	estado	FUNCTION	root@localhost	2025-01-01 12:00:00	2025-01-01 12:00:00	DEFINER

Esto indica que ya existe una función llamada `estado`, y podrías decidir eliminarla o cambiar el nombre antes de crear una nueva.

#### Ejemplo 4.5

```
USE world;
DROP FUNCTION IF EXISTS esimpar;
DELIMITER $$

CREATE FUNCTION esimpar(numero INT)
RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE impar INT;

    IF MOD(numero, 2) = 0 THEN
        SET impar = 0; -- 0 para falso en MySQL
    ELSE
        SET impar = 1; -- 1 para verdadero en MySQL
    END IF;

    RETURN impar;
END$$

DELIMITER ;
```

#### Propósito del código

La función `esimpar` determina si un número entero dado es impar o no. Devuelve:

- 1 (o verdadero) si el número es impar.
- 0 (o falso) si el número es par.

#### Explicación

1. **Entrada:** Un número entero (`numero`).
2. **Proceso:**
  - Se utiliza la función `MOD(numero, 2)` para calcular el resto de la división del número entre 2:
    - Si el resto es 0, el número es par, y la función devuelve 0.
    - Si el resto no es 0, el número es impar, y la función devuelve 1.
3. **Salida:** Un valor entero (1 o 0) que indica si el número es impar o no.

#### Uso práctico

La función se puede utilizar en consultas SQL para verificar si un número en una columna es impar:

```
SELECT numero, esimpar(numero) AS es_impar
FROM tabla_numeros;
```

Este ejemplo muestra cada número en la columna `numero` junto con un indicador (`es_impar`) que vale 1 si el número es impar y 0 si es par.

#### Ejemplo 4.7

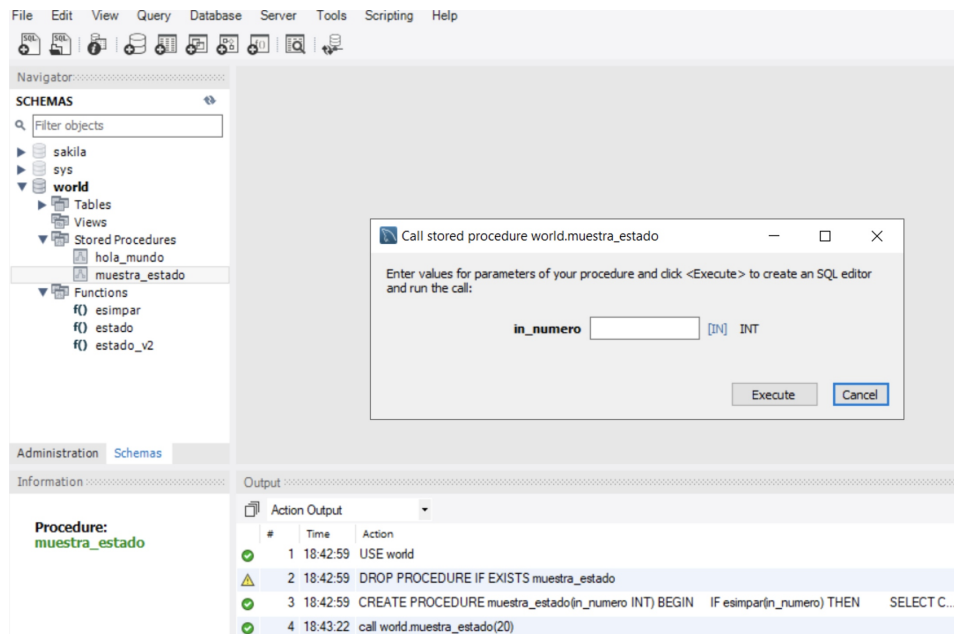
En este caso recibimos un número como entrada devolviendo TRUE si es par y FALSE en caso de que sea impar.

```
USE world;
DELIMITER $$

DROP PROCEDURE IF EXISTS muestra_estado$$

CREATE PROCEDURE muestra_estado(in_numero INT)
BEGIN
  IF esimpar(in_numero) THEN
    SELECT CONCAT(in_numero, ' es impar');
  ELSE
    SELECT CONCAT(in_numero, ' es par');
  END IF;
END$$

DELIMITER ;
```



The screenshot shows a database management tool interface. On the left, the 'Navigator' pane displays the 'world' database schema, including tables, views, stored procedures, and functions. The 'Stored Procedures' section is expanded, showing 'muestra\_estado'. In the center, a dialog box titled 'Call stored procedure world.muestra\_estado' prompts the user to enter values for parameters. The parameter 'in\_numero' is shown with a text input field and a data type of 'INT'. Below the input field are 'Execute' and 'Cancel' buttons. On the right, the 'Output' pane displays a log of actions performed, including 'USE world', 'DROP PROCEDURE IF EXISTS muestra\_estado', 'CREATE PROCEDURE muestra\_estado', and 'call world.muestra\_estado(20)'.

#### Ejemplo 4.8

##### PARAMETROS Y VARIABLES.

Igual que en otros lenguajes de programación los procedimientos y funciones usan variables y parámetros que determinan la salida del algoritmo. Veámoslo en el siguiente ejemplo más completo que el anterior:

```
USE world;
DELIMITER $$

DROP PROCEDURE IF EXISTS procl$$

CREATE PROCEDURE procl(IN parametrol INTEGER)
BEGIN
    DECLARE variable1 INTEGER DEFAULT 0;
    DECLARE variable2 INTEGER DEFAULT 0;

    IF parametrol = 17 THEN
        SET variable1 = parametrol;
    ELSE
        SET variable2 = 30;
    END IF;

    INSERT INTO t(variable1, variable2) VALUES (variable1, variable2);
END$$

DELIMITER ;
```

Encontramos dos nuevas cláusulas para el manejo de variables: DECLARE: crea una nueva variable con su nombre y tipo.

Los tipos son los usuales de MySQL como char, varchar, int, float, etc... Esta cláusula puede incluir una opción para indicar valores por defecto.

Si no se indica, dichos valores serán NULL.

Por ejemplo: DECLARE a, b INT DEFAULT 5; Crea dos variables enteras con valor 5 por defecto. SET: permite asignar valores a las variables usando el operador de igualdad.

En el ejemplo se recibe una variable entera de entrada llamada parametrol. 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 a la variable variable1 y sino la variable variable2 se le asigna el valor 30.

**Obviamente el ejemplo es absurdo y solo tiene propósitos didácticos.**

Tipos de parámetros. Existen 3 tipos.

**IN:** Es el tipo por defecto y sirve para incluir parámetros de entrada que usara el procedimiento. En este caso no se mantienen las modificaciones.

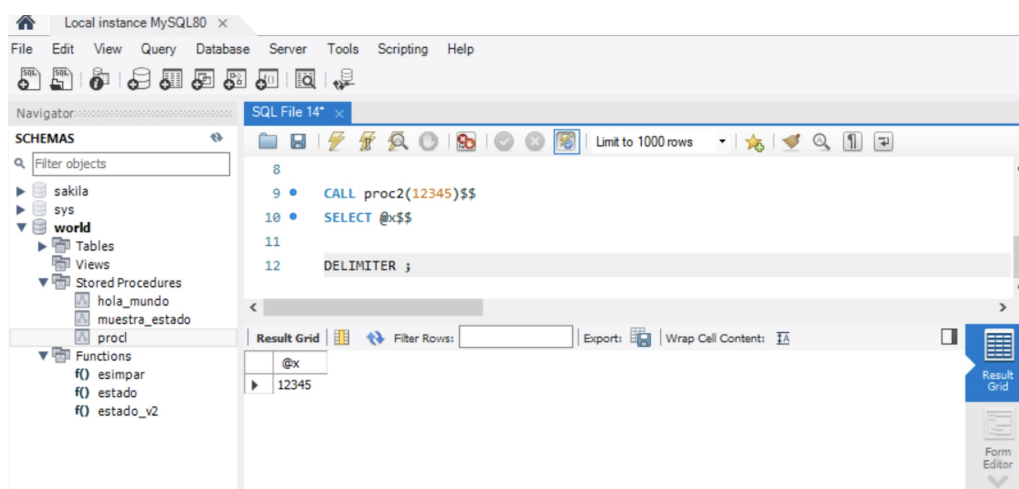
```
USE world;
DELIMITER $$

CREATE PROCEDURE proc2(IN p INT)
BEGIN
    SET @x = p;
END$$

CALL proc2(12345)$$
SELECT @x$$

DELIMITER ;
```

En este ejemplo de procedimiento se establece el valor de una variable de sesión (precedida por @) al valor de entrada p.





**OUT:** Parámetros de salida. El procedimiento puede asignar valores a dichos parámetros que son devueltos en la llamada.

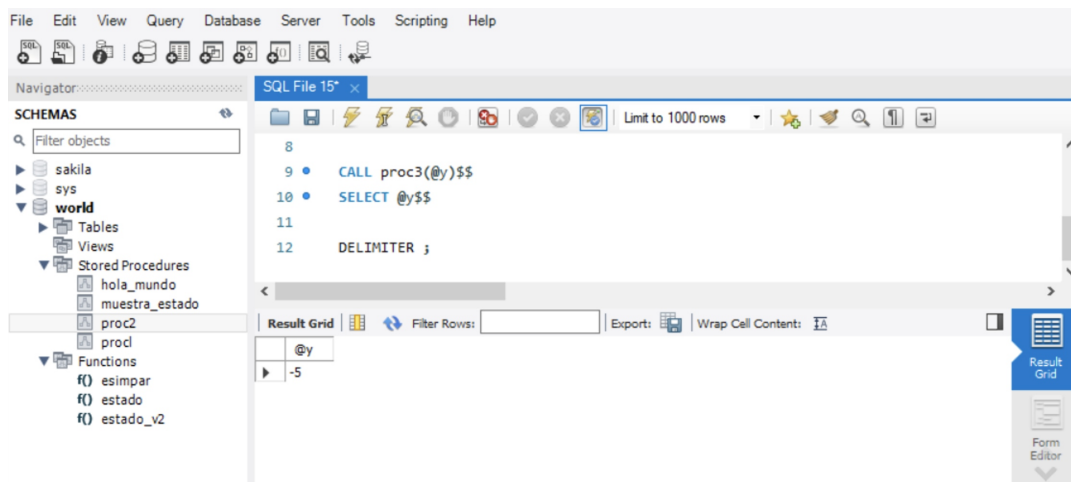
```
USE world;
DELIMITER $$

CREATE PROCEDURE proc3(OUT p INT)
BEGIN
    SET p = -5;
END$$

CALL proc3(@y)$$
SELECT @y$$

DELIMITER ;
```

En este caso hemos creado una nueva variable @y al llamar al procedimiento cuyo valor se actualiza dentro del mismo por ser ésta de tipo OUT.



**INOUT:** Permite pasar valores al proa que serán modificados y devueltos en la llamada.

```
USE world;
DELIMITER $$

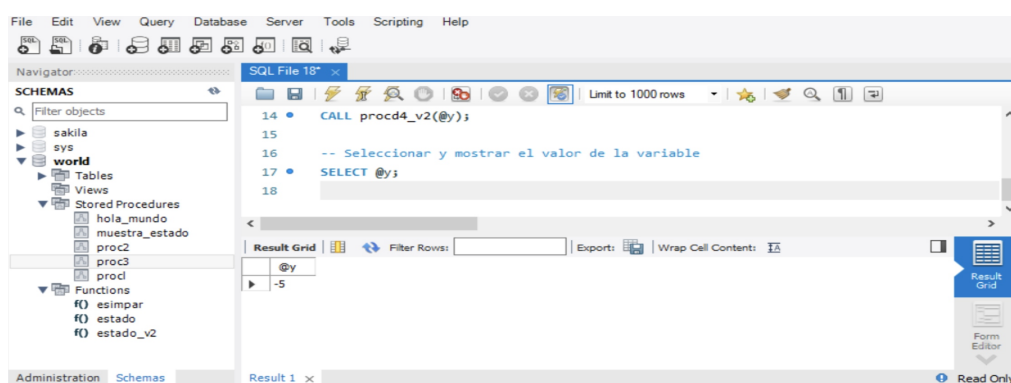
CREATE PROCEDURE procd4_v2(INOUT p INT)
BEGIN
    SET p = p - 5;
END$$

DELIMITER ;

-- Establecer la variable antes de llamar al procedimiento
SET @y = 0;

-- Llamar al procedimiento
CALL procd4_v2(@y);

-- Seleccionar y mostrar el valor de la variable
SELECT @y;
```



#### Propósito del código

El procedimiento `procd4` realiza una operación matemática simple sobre una variable pasada como parámetro y modifica su valor. En este caso, **reduce el valor de la variable en 5 unidades**.

- **Entrada/Salida (INOUT):** El parámetro `p` se pasa al procedimiento y es modificado dentro de él.
- **Salida:** El nuevo valor de la variable se refleja fuera del procedimiento gracias al uso de `INOUT`.

#### Flujo del código

- **Crear el procedimiento:** Define el procedimiento `procd4`, que toma un parámetro de entrada/salida (`p`) y reduce su valor en 5.
- **Inicializar la variable:** Establece la variable `@y` con un valor inicial (en este caso, 0).
- **Llamar al procedimiento:** Usa `CALL procd4(@y)` para modificar el valor de `@y` dentro del procedimiento.
- **Consultar el resultado:** Usa `SELECT @y` para verificar el nuevo valor de la variable.

#### Ejemplo de ejecución

1. Antes de llamar al procedimiento:

```
SET @y = 10;
CALL procd4(@y);
SELECT @y;
Resultado: @y = 5
```

2. Si el valor inicial de `@y` es 0:

```
SET @y = 0;
CALL procd4(@y);
SELECT @y;
Resultado: @y = -5
```

## INSTRUCCIONES CONDICIONALES

En muchas ocasiones, el valor de una o más variables o parámetros determina el proceso a seguir. Cuando esto ocurre, debemos utilizar instrucciones condicionales. Existen diferentes tipos de estructuras condicionales:

- **IF simple:** Se utiliza cuando solo hay una condición a evaluar.
- **IF THEN ELSE:** Apropiado cuando existen dos posibilidades; permite definir una acción alternativa.
- **CASE:** Se emplea cuando tenemos un conjunto de condiciones distintas a evaluar, permitiendo múltiples alternativas.

Estas estructuras nos ayudan a dirigir el flujo del programa de acuerdo con las condiciones evaluadas.

Como hemos visto en el ejemplo 4.2, podemos incluir instrucciones condicionales usando **IF** o, de manera más completa, **IF - THEN - ELSE** para manejar múltiples condiciones. La sintaxis general para esta estructura es:

```
USE world;  
IF expr1 THEN  
  -- acciones a realizar si expr1 es verdadera  
ELSEIF expr2 THEN  
  -- acciones a realizar si expr2 es verdadera  
ELSE  
  -- acciones a realizar si ninguna de las condiciones anteriores se cumple  
END IF
```

#### Ejemplo 4.13

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

```
USE world;  
DELIMITER $$  
  
CREATE PROCEDURE proc7(IN par1 INT)  
BEGIN  
    DECLARE var1 INT;  
    SET var1 = par1 + 1;  
  
    IF par1 = 0 THEN  
        UPDATE t SET s1 = s1 + 2;  
    END IF;  
  
    INSERT INTO t VALUES (var1); -- Asumiendo que se desea insertar el valor de var1  
END$$  
  
DELIMITER ;
```

Cuando el valor de la variable 1 es 0 entonces hacemos una inserción, en caso de que sea 0 el parámetro de entrada actualizamos sumando 1 al valor actual y sino sumamos 2.

## CASE

Cuando hay muchas condiciones a evaluar, el uso de estructuras condicionales sucesivas como IF-THEN-ELSE puede generar confusión en el código. Para estos casos, es más apropiado utilizar la instrucción CASE, que simplifica la lectura y la escritura de múltiples ramificaciones condicionales.

Sintaxis general:

```
CASE expression  
WHEN value1 THEN statements  
[WHEN value2 THEN statements ...]  
[ELSE statements]  
END CASE;
```

Donde `expression` es una expresión cuyo valor se compara con uno de los valores posibles (val1, val2, etc.). Si ninguno de los valores coincide, se ejecutan las instrucciones especificadas en la cláusula ELSE.

## Ejemplo 4.14

```
USE world;
DELIMITER $$

CREATE PROCEDURE proc8(IN parameter1 INT)
BEGIN
    DECLARE variable1 INT;
    SET variable1 = parameter1 + 1;

    CASE variable1
        WHEN 0 THEN INSERT INTO t VALUES (17);
        WHEN 1 THEN INSERT INTO t VALUES (18);
        ELSE INSERT INTO t VALUES (19);
    END CASE;
END$$

DELIMITER ;
```

### Propósito del código

El propósito del código es crear un procedimiento almacenado llamado `proc8` en la base de datos `world`. Este procedimiento:

- Toma un **parámetro de entrada** (`parameter1`), que es un número entero (INT).
- Declara una **variable interna** (`variable1`) para realizar operaciones dentro del procedimiento.
- Realiza una **operación matemática**: Incrementa el valor de `parameter1` en 1 y lo almacena en `variable1`.
- Evalúa el valor de `variable1` con una estructura **CASE**:
  - Si `variable1` es igual a 0, inserta el valor 17 en la tabla `t`.
  - Si `variable1` es igual a 1, inserta el valor 18 en la tabla `t`.
  - Para cualquier otro valor, inserta 19 en la tabla `t`.

En resumen, este procedimiento **modifica el valor de `parameter1`, evalúa el resultado y realiza una inserción condicional en la tabla `t` basada en dicho valor.**

### Detalles de ejecución

- **Entrada:**
  - El procedimiento acepta un parámetro de tipo entero (`parameter1`).
- **Proceso:**
  - Incrementa `parameter1` en 1 (`SET variable1 = parameter1 + 1`).
  - Utiliza una estructura CASE para decidir qué valor insertar en la tabla `t`.
- **Salida:**
  - Inserta un valor fijo (17, 18 o 19) en la tabla `t` dependiendo del resultado de la evaluación.

### Ejemplo de uso

Supongamos que existe una tabla `t` definida como:

```
CREATE TABLE t (value INT);
```

Si ejecutamos el procedimiento:

#### 1. Llamada con `parameter1 = -1`:

sql

Copiar código

```
CALL proc8(-1);
```

Proceso:

- `variable1 = -1 + 1 = 0`
- Inserta 17 en la tabla `t`. Resultado en la tabla `t`:

```
plaintext
Copiar código
value
-----
17
```

## 2. Llamada con `parameter1 = 0`:

```
sql
Copiar código
CALL proc8(0);
Proceso:
```

- $\text{variable1} = 0 + 1 = 1$
- Inserta 18 en la tabla `t`. Resultado en la tabla `t`:

```
plaintext
Copiar código
value
-----
17
18
```

## 3. Llamada con `parameter1 = 2`:

```
sql
Copiar código
CALL proc8(2);
Proceso:
```

- $\text{variable1} = 2 + 1 = 3$
- Inserta 19 en la tabla `t`. Resultado en la tabla `t`:

```
plaintext
Copiar código
value
-----
17
18
19
```

### Limitaciones y observaciones

1. **Dependencia de la tabla `t`:** El procedimiento asume que la tabla `t` existe y tiene una columna llamada `value` de tipo entero (INT).
2. **Inserciones fijas:** Los valores insertados (17, 18, 19) son fijos y no dependen directamente del valor de `parameter1`, sino de su evaluación en el CASE.
3. **Seguridad de datos:** Si no se controla adecuadamente, este procedimiento puede generar múltiples inserciones en la tabla `t` sin validación previa.

#### 4.2.4 INSTRUCCIONES REPETITIVAS O LOOPS

Los loops permiten iterar un conjunto de instrucciones un número determinado de veces. MySQL provee tres tipos de estructuras de loop: Simple loop, Repeat until y While loop.

**Simple Loop.** Su sintaxis básica es:

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

Donde la palabra opcional *etiqueta* permite etiquetar el loop para referirse a él dentro del bloque.

**Ejemplo de un Bucle Infinito:** Este ejemplo muestra un bucle infinito que no se recomienda probar:

```
Infinite_loop: LOOP
    SELECT 'Esto no acaba nunca!!!';
END LOOP Infinite_loop;
```

**Uso de LEAVE para Salir de un Loop:** En el siguiente ejemplo, etiquetamos el loop con el nombre `loop_label`. Este loop se ejecuta mientras no se cumpla una condición específica. En caso de que se cumpla, la orden LEAVE termina el loop etiquetado como `loop_label`.

```
loop_label: LOOP
    -- Instrucciones aquí
    IF condición THEN
        LEAVE loop_label;
    END IF;
    -- Más instrucciones
END LOOP loop_label;
```

Este ejemplo incrementa y muestra un contador, comenzando desde un valor inicial establecido (asegúrate de que `@counter` está inicializado antes de comenzar el loop).

El bucle se repite hasta que el valor del contador alcanza o supera 5.

```
counter_example: REPEAT
    SET @counter = @counter + 1;
    SELECT @counter;
UNTIL @counter >= 5
END REPEAT counter_example;
```

#### While Loop

El `while` loop ejecuta un conjunto de instrucciones repetidamente mientras una condición dada es verdadera. Es ideal para situaciones donde se necesita repetir una acción pero sólo mientras se cumpla una condición específica.

**Sintaxis General:**

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

**etiqueta:** Opcional, utilizada para identificar el bucle, especialmente útil para las sentencias LEAVE.



**expresión:** La condición que se evaluará antes de cada iteración del bucle. Si la condición es verdadera, se ejecutan las instrucciones dentro del bucle.

**instrucciones:** Las instrucciones SQL que se ejecutarán en cada iteración mientras la condición sea verdadera.

### Repeat Until Loop

El Repeat until loop ejecuta un bloque de instrucciones repetidamente hasta que una condición específica se cumple. Es útil cuando necesitas que el cuerpo del bucle se ejecute al menos una vez antes de evaluar la condición.

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

**etiqueta:** Una etiqueta opcional que puede ser utilizada para identificar el loop y para posibles sentencias LEAVE.

**instrucciones:** Las instrucciones SQL que se ejecutarán en cada iteración del bucle.

**expresión:** La condición que se evaluará después de cada iteración. El loop continúa hasta que esta expresión resulte verdadera.

**Ejemplo de Uso:** Este ejemplo muestra cómo utilizar un While loop para identificar números impares dentro de un rango determinado:

```
USE world;
DELIMITER $$

CREATE PROCEDURE proc10()
BEGIN
  DECLARE i INT DEFAULT 1; -- Inicialización correcta dentro de un bloque BEGIN...END

  loop1: WHILE i <= 10 DO
    IF MOD(i, 2) <> 0 THEN
      SELECT CONCAT(i, ' es impar');
    END IF;
    SET i = i + 1;
  END WHILE loop1;
END$$

DELIMITER ;
```

#### 4.2.5 SQL en rutinas: Cursores.

Hasta ahora, todos los ejemplos que hemos visto se han centrado en instrucciones o expresiones referidas a cálculos matemáticos o manipulaciones de cadenas sencillas, sin implicar el uso de datos de una base de datos.

Sin embargo, normalmente el uso de procedimientos almacenados implica manipular datos en tablas de bases de datos, lo que requiere el uso de instrucciones SQL. En esta sección, veremos ejemplos diversos de procedimientos que acceden a bases de datos, haciendo uso de las instrucciones explicadas en apartados anteriores.

En general, podemos utilizar cualquier instrucción de SQL, incluyendo aquellas pertenecientes al DDL (Data Definition Language), DML (Data Manipulation Language) y DCL (Data Control Language).

#### Ejemplo 4.18

Para comenzar, veamos un ejemplo en el que usamos sentencias SQL de definición (DROP y CREATE) junto con sentencias de manipulación (INSERT, UPDATE y DELETE):

```
USE world;
DELIMITER $$

CREATE PROCEDURE simple_sql()
BEGIN
    DECLARE i INT DEFAULT 1;

    /* Instrucción DDL para eliminar y crear una tabla */
    DROP TABLE IF EXISTS test_table;
    CREATE TABLE test_table (
        id INT PRIMARY KEY,
        some_data VARCHAR(30)
    ) ENGINE=InnoDB;

    /* INSERT usando una variable de procedimiento */
    WHILE i <= 10 DO
        INSERT INTO test_table (id, some_data) VALUES (i, CONCAT('record ', i));
        SET i = i + 1;
    END WHILE;

    /* Ejemplo de actualización usando variables de procedimiento */
    SET i = 5;
    UPDATE test_table SET some_data = CONCAT('I updated row ', i) WHERE id = i;

    /* DELETE with a procedure variable */
    DELETE FROM test_table WHERE id > i;
END$$

DELIMITER ;
```

#### Ejemplo 4.19

En el siguiente ejemplo usamos la propiedad de las sentencias select de enviar valores a variables usando INTO:

```
USE world;  
DELIMITER $$  
  
CREATE PROCEDURE motorblog.obtener_datos_noticia(id_noticia INT)  
BEGIN  
  DECLARE vtitulo VARCHAR(200);  
  DECLARE vcontenido TEXT;  
  DECLARE vfecha DATE;  
  
  -- Recuperar los datos de la tabla 'noticias' basado en 'id_noticia'  
  SELECT titulo, contenido, fecha  
  INTO vtitulo, vcontenido, vfecha  
  FROM noticias  
  WHERE id = id_noticia;  
  
  -- Procesar los datos obtenidos, por ejemplo, mostrándolos  
  SELECT vtitulo, vcontenido, vfecha;  
END$$  
  
DELIMITER ;
```

En el anterior ejemplo observamos como la sentencia SELECT asigna los valores de la fila seleccionada para asignarlos a su vez a nuevas variables internas del procedimiento. No obstante muchas veces queremos recuperar más de una fila para manipular sus datos, en estos casos no sirve la sentencia anterior y requerimos el uso de cursores. Conceptualmente un cursor se asocia con un conjunto de filas o una consulta sobre una tabla de una base de datos.

Un cursor se define del siguiente modo: DECLARE cursor\_name CURSOR FOR SELECT\_statement; Y debe hacerse después de declarar todas las variables necesarias para el procedimiento.

Ejemplo 4.21 Un ejemplo con variables sería:

```
USE world;
DELIMITER $$

CREATE PROCEDURE cursor_demo(IN id_noticia INT)
BEGIN
    DECLARE vid INT;
    DECLARE vtitulo VARCHAR(30);
    DECLARE done INT DEFAULT FALSE;
    DECLARE cl CURSOR FOR
        SELECT id, titulo
        FROM noticias
        WHERE id = id_noticia;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    OPEN cl;

    read_loop: LOOP
        FETCH cl INTO vid, vtitulo;
        IF done THEN
            LEAVE read_loop;
        END IF;
        -- Aquí puedes procesar cada fila, por ejemplo, mostrando los valores
        -- Esta línea solo es un placeholder para tus operaciones:
        SELECT vid, vtitulo;
    END LOOP;

    CLOSE cl;
END$$

DELIMITER ;
```

### Comandos Relacionados con Cursores

Para manipular los cursores en SQL, disponemos de una serie de comandos esenciales:

- **OPEN:** Inicializa el conjunto de resultados asociados con el cursor.

```
OPEN cursor_name;
```

- **FETCH:** Extrae la siguiente fila de valores del conjunto de resultados del cursor, moviendo su puntero interno una posición hacia adelante. Este comando se utiliza para cargar los valores de la fila en variables específicas.

```
FETCH cursor_name INTO variable_list;
```

- **CLOSE:** Cierra el cursor, liberando la memoria que ocupa y haciendo imposible el acceso a cualquiera de sus datos.

```
CLOSE cursor_name;
```

Estos comandos son fundamentales para la manipulación efectiva de cursores en bases de datos SQL, permitiendo el manejo controlado de filas de datos desde una consulta.

## Ejemplo 2.24

```

USE world;
DELIMITER $$

CREATE PROCEDURE cursor_demo3()
BEGIN
  DECLARE tmp VARCHAR(200);
  DECLARE lrf BOOL DEFAULT FALSE; -- Inicialización correcta de la variable booleana
  DECLARE nn INT DEFAULT 0;      -- Inicialización correcta de la variable contador
  DECLARE cursor2 CURSOR FOR SELECT titulo FROM noticias;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET lrf = TRUE; -- Establece lrf a TRUE
  cuando no hay más filas

  OPEN cursor2;

  cursor_loop: LOOP
    FETCH cursor2 INTO tmp;
    SET nn = nn + 1;
    IF lrf THEN
      LEAVE cursor_loop;
    END IF;
  END LOOP cursor_loop;

  CLOSE cursor2;
  SELECT nn; -- Muestra el número de filas procesadas
END$$

DELIMITER ;

```

En este caso hemos declarado dos nuevas variables, *lrf* (last row fetched o última fila extraída) que es una variable booleana con posibles valores 0 y 1 indicando si hemos llegado o no a la última fila del cursor, por su parte *nn* almacena el número de noticias o registros contenidos en el cursor. Gracias a la sentencia *LEAVE*, podemos terminar el bucle cuando *lrf* adquiere el valor 1 o lo que es lo mismo, se alcanza el final del cursor.

Para ilustrar lo visto hasta ahora veremos un ejemplo más elaborado en el que se obtienen y muestran el número de noticias publicadas de cada autor para lo cual se precisan dos cursores:

```

USE world;
DELIMITER $$
CREATE PROCEDURE noticias_autor()
READS SQL DATA
BEGIN
  DECLARE v_autor INT;
  DECLARE na_count INT;
  DECLARE fin BOOL DEFAULT FALSE;
  DECLARE autor_cursor CURSOR FOR SELECT id_autor FROM autor;
  DECLARE noticia_cursor CURSOR FOR SELECT autor FROM noticias WHERE autor = v_autor;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin = TRUE;

  OPEN autor_cursor;
  autor_loop: LOOP
    FETCH autor_cursor INTO v_autor;

```

```

IF fin THEN
    LEAVE autor_loop;
END IF;

SET na_count = 0;
OPEN noticia_cursor;
noticias_loop: LOOP
    FETCH noticia_cursor INTO v_autor;
    IF fin THEN
        LEAVE noticias_loop;
    END IF;
    SET na_count = na_count + 1;
END LOOP noticias_loop;
CLOSE noticia_cursor;

SELECT CONCAT('El autor ', v_autor, ' tiene ', na_count, ' noticias');
SET fin = FALSE;
END LOOP autor_loop;
CLOSE autor_cursor;
END; $$

```

#### 4.2.6 Gestión de rutinas almacenadas

Las rutinas se manipulan con los comandos CREATE (ya visto), DROP y SHOW.

**Eliminación de rutinas:** Para eliminar procedimientos o funciones, usamos el comando SQL DROP con la siguiente sintaxis:

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

**Consulta de rutinas:** Podemos obtener información más o menos detallada de nuestras rutinas usando los comandos:

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

En el segundo caso, podemos hacer un filtro por patrones. Estos comandos, y en general todos los de tipo SHOW, se nutren del diccionario de datos gracias a la tabla INFORMATION\_SCHEMA.ROUTINES, que también podemos consultar con instrucciones SELECT.

#### 4.2.7 Manejo de errores

Cuando un programa almacenado encuentra una condición de error, la ejecución se detiene y se devuelve un error a la aplicación que llama. Ese es el comportamiento predeterminado. ¿Qué pasa si necesitamos otro tipo de comportamiento? ¿Qué pasa si, por ejemplo, queremos que la trampa de error, log, o informar al respecto y luego continuar con la ejecución de nuestra aplicación? Para ese tipo de control tenemos que definir controladores de excepciones en nuestros programas (iguales que los vistos en la parte de cursores).

Procedimiento sin manejo de errores	Procedimiento con manejo de errores
<b>CREATE PROCEDURE insertar_noticia(   titulo VARCHAR(200), contenido TEXT, fecha   DATE</b>	<b>DELIMITER \$\$ CREATE PROCEDURE insertar_autor(   pautor INT, plogin VARCHAR(45), OUT estado</b>

<b>)</b> <b>MODIFIES SQL DATA</b> <b>BEGIN</b> <b>INSERT INTO noticias (titulo, contenido, fecha)</b> <b>VALUES (titulo, contenido, fecha);</b> <b>END;</b>	<b>VARCHAR(45)</b> <b>)</b> <b>MODIFIES SQL DATA</b> <b>BEGIN</b> <b>DECLARE CONTINUE HANDLER FOR</b> <b>SQLSTATE '23000' SET estado = 'Duplicate Entry';</b> <b>SET estado = 'OK';</b> <b>INSERT INTO autores (id_autor, login) VALUES</b> <b>(pautor, plogin);</b> <b>END; \$\$</b> <b>DELIMITER ;</b>
--	--

Pero si queremos hacer algo con el error debemos usar la variable out\_status. En el siguiente ejemplo llamamos al procedimiento dentro de otro procedimiento condicionando la salida al valor de la variable estado de tipo out:

<b>USE world;</b> <b>DELIMITER \$\$</b>  <b>CREATE PROCEDURE insertar_comentario(</b> <b>pautor INT, pfecha DATE, pcontenido VARCHAR(30), plogin VARCHAR(45)</b> <b>)</b> <b>MODIFIES SQL DATA</b> <b>BEGIN</b> <b>DECLARE estado VARCHAR(20);</b>  <b>CALL insertar_autor(pautor, plogin, estado);</b> <b>IF estado = 'Duplicate Entry' THEN</b> <b>SELECT CONCAT('Warning: autor repetido ', pautor, ' login ', plogin) AS warning;</b> <b>END IF;</b> <b>INSERT INTO comentario (autor, contenido, fecha) VALUES (pautor, pcontenido, pfecha);</b> <b>END; \$\$</b>  <b>DELIMITER ;</b>
---

#### 4.2.7.1 Sintaxis del manejador.

- **EXIT** Cuando se encuentra un error, el bloque que se está ejecutando actualmente se termina. Si este bloque es el bloque principal, el procedimiento termina y el control se devuelve al procedimiento o programa externo que invocó el procedimiento. Si el bloque está encerrado en un bloque externo dentro del mismo programa almacenado, el control se devuelve a ese bloque exterior.
- **CONTINUE** Para el caso de CONTINUE, la ejecución continúa en la declaración siguiente a la que ocasionó el error. En cualquier caso, las declaraciones definidas dentro del manejador (el controlador de las acciones) se ejecutan antes de que se lleve a cabo la EXIT o CONTINUE.

Ejemplo 4.31 Ejemplo con EXIT.

```
USE world;
DELIMITER $$

CREATE PROCEDURE insertar_autor(
    pautor INT,
    plogin VARCHAR(45)
)
MODIFIES SQL DATA
BEGIN
    DECLARE duplicate_key INT DEFAULT 0;

    -- Definición del bloque interno
    BEGIN
        DECLARE EXIT HANDLER FOR SQLSTATE '23000' /* Clave repetida */
        SET duplicate_key = 1;

        INSERT INTO autores (id_autor, login) VALUES (pautor, plogin);
    END;

    -- Verificación del resultado y respuesta acorde
    IF duplicate_key = 1 THEN
        SELECT CONCAT('Error en la inserción: clave duplicada') AS Resultado;
    ELSE
        SELECT CONCAT('Autor ', plogin, ' creado') AS Resultado;
    END IF;
END; $$

DELIMITER ;
```



Ejemplo 4.32 con CONTINUE.

```
USE world;
DELIMITER $$

CREATE PROCEDURE insertar_autor(
    pautor INT,
    plogin VARCHAR(45)
)
MODIFIES SQL DATA
BEGIN
    DECLARE duplicate_key INT DEFAULT 0;
    DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET duplicate_key = 1; -- Cambiado a
manejar SQLSTATE en lugar de un número de error

    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$$

DELIMITER ;
```

### 4.3 Triggers

Un trigger, o disparador, es un tipo especial de rutina almacenada que se activa automáticamente cuando ocurren eventos específicos en una tabla, como INSERT, DELETE o UPDATE. Los disparadores permiten implementar funcionalidades automáticas que responden a cualquier cambio en una tabla, asegurando que ciertas reglas o lógicas de negocio se mantengan sin intervención manual directa.

#### Ejemplo 4.33

El disparador suma las cantidades insertadas cada vez que se introduce un nuevo movimiento en la variable de usuario @sum:

```
USE world;
DELIMITER $$

CREATE TRIGGER insertar_movimiento
BEFORE INSERT ON movimiento
FOR EACH ROW
BEGIN
    SET @sum = COALESCE(@sum, 0) + NEW.cantidad;
END$$

DELIMITER ;
```

#### 4.4.1 Gestión de disparadores

##### Crear un disparador:

```
CREATE TRIGGER nombre_disp
momento_disp evento_disp ON nombre_tabla
FOR EACH ROW
sentencia_disp
```

- **momento\_disp:** Define el momento en que el disparador entra en acción. Puede ser BEFORE (antes) o AFTER (después), para indicar que el disparador se ejecute antes o después de la sentencia que lo activa.
- **evento\_disp:** Indica la clase de sentencia que activa al disparador. Puede ser INSERT, UPDATE, o DELETE. Por ejemplo, un disparador BEFORE INSERT podría utilizarse para validar los valores a insertar. No puede haber dos disparadores en una misma tabla que correspondan al mismo momento y evento.

**Eliminación de disparadores:** Para eliminar un disparador, se emplea la sentencia DROP TRIGGER. El nombre del disparador debe incluir el nombre de la tabla:

```
DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name
```

**Consulta de disparadores:** Podemos obtener información de los disparadores creados con SHOW TRIGGERS:

```
SHOW TRIGGERS [(FROM | IN) db_name] [LIKE 'pattern' | WHERE expr]
```

Este comando permite mostrar disparadores de una base de datos, filtrándolos con un patrón o cláusula **WHERE**. Cuando se crean disparadores, se crea un nuevo registro en la tabla **INFORMATION\_SCHEMA . TRIGGERS** que podemos visualizar con el siguiente comando:

```
SELECT trigger_name, action_statement FROM information_schema.triggers
```

#### 4.2.4 Uso de los disparadores

##### Control de sesiones.

En ocasiones, puede ser útil almacenar ciertos valores en variables de sesión creadas por el usuario que permitan visualizar un resumen de las actividades realizadas durante esa sesión. Este es el caso del ejemplo proporcionado.

En dicho ejemplo, antes de insertar uno o varios movimientos, se acumula la cantidad total de todos ellos en la variable de usuario **@sum**. Para utilizar esto, se debe establecer el valor de la variable acumulador a cero, ejecutar una o varias sentencias **INSERT** y luego verificar el valor que presenta la variable:

```
mysql> SET @sum = 0;
mysql> INSERT INTO movimiento VALUES (137, 14.98), (141, 1937.50), (97,
-100.00);
mysql> SELECT @sum AS "Total insertado";
+-----+
| Total insertado |
+-----+
| 1852.48         |
+-----+
```

En este caso, el valor de **@sum** después de ejecutar la sentencia **INSERT** es 1852.48, que es el resultado de sumar 14.98, 1937.50 y restar 100.

##### Control de Valores de Entrada

Una posible aplicación de los disparadores es el control de los valores insertados o actualizados en las tablas.

En el siguiente ejemplo, se crea un disparador en la tabla **movimiento** para la acción **UPDATE** que verifica los valores utilizados para actualizar cada columna, asegurándose de que estos valores estén dentro de un rango permitido de 0 a 100. Esto debe hacerse en un disparador **BEFORE** porque los valores deben ser verificados antes de actualizar el registro:

```
USE world;
DELIMITER $$

CREATE TRIGGER comprobacion_saldo
BEFORE UPDATE ON movimiento
FOR EACH ROW
BEGIN
  IF NEW.cantidad < 0 THEN
    SET NEW.cantidad = 0;
  ELSEIF NEW.cantidad > 100 THEN
    SET NEW.cantidad = 100;
```

```
END IF;  
END; $$  
  
DELIMITER ;
```

#### 4.4 VISTAS

##### Resumen y Corrección sobre el Uso de Vistas en Bases de Datos

Las vistas en bases de datos son esencialmente consultas guardadas que representan un subconjunto de datos. Son especialmente útiles en entornos de producción empresarial para filtrar información de bases de datos complejas con alta normalización y numerosas claves foráneas. Además, las vistas ofrecen beneficios significativos en términos de seguridad y control de acceso, permitiendo al Administrador de Bases de Datos (DBA) proteger los datos y gestionar cómo los usuarios finales acceden a la información sin interactuar directamente con las tablas.

Las vistas pueden incorporar múltiples tablas, otras vistas, subconsultas y operaciones de **JOIN**. Si bien en ciertos contextos es posible actualizar y eliminar datos a través de vistas, existen restricciones significativas para mantener la integridad y seguridad de los datos:

- No se deben utilizar tablas temporales.
- Se prohíbe el uso de cláusulas **GROUP BY**, **HAVING**.
- Las uniones externas y las consultas correlacionadas están restringidas.
- Solo se permite la actualización o inserción en reuniones **INNER** cuando afecten a campos de una sola tabla implicada.

Las vistas reflejan instantáneamente los cambios en las tablas subyacentes. Son actualizables si cumplen ciertos criterios, como una relación uno-a-uno con los registros de la tabla subyacente, y no contienen columnas derivadas ni duplicadas, excepto en actualizaciones específicas.

Además, las vistas pueden emplear la cláusula **WITH CHECK OPTION** para asegurar que solo se inserten o actualicen registros que cumplan las condiciones especificadas en la cláusula **WHERE** de la definición de la vista. Esta funcionalidad añade una capa adicional de control al modificar datos a través de una vista, garantizando que solo se realicen cambios validados y acordes con las políticas de seguridad de la base de datos.

Estas características hacen de las vistas una herramienta poderosa y flexible para gestionar el acceso a los datos y realizar consultas eficientes en sistemas de bases de datos complejos.

#### 4.4.1 Gestión de vistas

Los comandos que nos permiten trabajar con vistas en MySQL incluyen CREATE VIEW, ALTER VIEW, DROP VIEW, y SHOW CREATE VIEW.

##### Creación: CREATE VIEW

```
CREATE [OR REPLACE]
[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
[DEFINER = {user | CURRENT USER}] [SQL SECURITY {DEFINER | INVOKER}]
VIEW view_name [(column_list)] AS select_statement
[WITH [CASCADED | LOCAL] CHECK OPTION];
```

Esta sentencia crea una nueva vista o reemplaza una existente con la cláusula OR REPLACE. La select\_statement proporciona la definición de la vista y puede dirigirse a tablas o a otras vistas.

- **ALGORITHM:** Indica el algoritmo utilizado. Opciones incluyen:
  - MERGE: Integra la consulta de la vista con la consulta del solicitante.
  - TEMPTABLE: Crea una tabla temporal, haciendo la vista no actualizable pero liberando bloqueos más rápidamente.
  - UNDEFINED: Permite que MySQL elija el algoritmo automáticamente.

Ejemplo con algoritmo MERGE:

```
DELIMITER $$
CREATE ALGORITHM = MERGE VIEW vdatos_jugador AS
SELECT * FROM jugador WHERE id_jugador = identificador_jugador$$
```

Este ejemplo proporciona a cada jugador acceso solo a sus propios datos.

- **DEFINER:** Determina la cuenta asociada a la vista.
- **SQL\_SECURITY:** Define los permisos de ejecución de la vista, ya sea como el creador (DEFINER) o el invocador (INVOKER).
- **WITH CHECK OPTION:** Restringe las modificaciones a las filas afectadas por la consulta que define la vista. LOCAL restringe la comprobación a la vista actual; CASCADED la extiende a cualquier vista que incluya la vista actual.

##### Modificación: ALTER VIEW

```
ALTER VIEW view_name [(column_list)]
AS select_statement
[WITH [CASCADED | LOCAL] CHECK OPTION];
```

Permite modificar una vista existente, aplicando cambios estructurales o de lógica a la consulta.

##### Eliminación: DROP VIEW

```
DROP VIEW [IF EXISTS] view_name [, view_name2, ...] [RESTRICT | CASCADE];
```

Elimina una o más vistas. IF EXISTS evita errores si la vista no existe. RESTRICT y CASCADE gestionan cómo se tratan las dependencias.

## Consulta de Vistas

Para obtener información sobre la definición de una vista, se puede utilizar:

```
SHOW CREATE VIEW view_name;
```

O acceder directamente a la tabla VIEWS del INFORMATION\_SCHEMA:

```
SELECT VIEW_DEFINITION FROM INFORMATION_SCHEMA.VIEWS  
WHERE TABLE_SCHEMA = 'test';
```

Este comando proporciona detalles sobre las vistas en la base de datos especificada.

## 4.5 Eventos

Los eventos en MySQL son tareas automatizadas que se ejecutan según un horario predefinido, similar a `cron` en Linux o el Programador de Tareas en Windows. Funcionan como "triggers temporales", activándose en momentos específicos en lugar de hacerlo en respuesta a cambios en la base de datos.

### Habilitación del Programador de Eventos:

La variable global `event_scheduler` controla si el programador de eventos está activo:

- **ON:** Activa el programador.
- **OFF:** Desactiva el programador.
- **DISABLED:** Impide que el programador sea activado en tiempo de ejecución.

Para verificar si el programador está activo, se puede usar:

```
SHOW PROCESSLIST;
```

Para activar el programador:

```
SET GLOBAL event_scheduler = ON;
```

### Gestión de Eventos:

- **Crear Evento:**

```
CREATE EVENT [IF NOT EXISTS] event_name
ON SCHEDULE schedule
[ON COMPLETION [NOT] PRESERVE]
[ENABLE | DISABLE | DISABLE ON SLAVE]
[COMMENT 'comment']
DO event_body;
```

**ON SCHEDULE:** Define cuándo y cómo se ejecuta el evento, pudiendo ser en un momento específico, repetidamente o entre fechas determinadas.

**DEFINER:** Especifica el usuario bajo cuyo permiso se ejecutará el evento.

**event\_body:** Las instrucciones SQL que se ejecutarán cuando el evento sea activado.

- **Modificar Evento:**

```
ALTER EVENT event_name
[ON SCHEDULE schedule]
[ON COMPLETION [NOT] PRESERVE]
[RENAME TO new_event_name]
[ENABLE | DISABLE | DISABLE ON SLAVE]
[COMMENT 'comment']
[DO event_body];
```

### Consultar Eventos:

```
SHOW EVENTS [(FROM | IN) schema_name] [LIKE 'pattern' | WHERE expr];
```

**Eliminar Evento:**

**DROP EVENT [IF EXISTS] event\_name;**



Plantilla mejorada:

```
-- Creación de un procedimiento almacenado
CREATE PROCEDURE sp_name ([parameter[, ...]])
[characteristic[, ...]]
BEGIN
    -- routine_body: Coloca aquí las sentencias SQL
END;

-- Creación de una función
CREATE FUNCTION sp_name ([parameter[, ...]])
RETURNS type
[characteristic[, ...]]
BEGIN
    -- routine_body: Coloca aquí las sentencias SQL
    RETURN value; -- Asegúrate de incluir una sentencia RETURN adecuada
END;

-- Detalles de los parámetros
parameter:
[ IN | OUT | INOUT ] param_name type

-- Tipo de datos
type:
-- Cualquier tipo de dato válido de MySQL

-- Características de la rutina
characteristic:
LANGUAGE SQL
| [NOT] DETERMINISTIC
| { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
| SQL SECURITY { DEFINER | INVOKER }
```