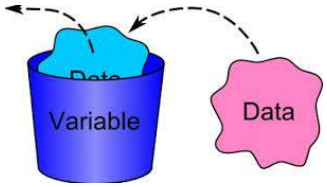
Unidad 6: Programación de bases de datos

Índice de contenidos

Realización de ejemplos y tareas sobre la *BD* "***conferencias2***"

1 Variables.



Como en cualquier otro lenguaje de programación, se tiene la posibilidad de utilizar variables. La diferencia con respeto a los lenguajes tradicionales es que **se pueden utilizar para guardar resultados de consultas o emplearlas como datos a enviar a sentencias como `INSERT`, `UPDATE`, …**

1.1 Variables de sesión (prefijo @).

Variables que **se pueden usar de forma global o bien en scripts *SQL***. Este tipo de variables se conocen como variables de sesión y deben cumplir las siguientes **características**:

- ✓ **Empiezan con el símbolo @**.
- ✓ Pueden incluir **cualquier carácter alfanumérico**, los **símbolos `.`, `_`, `$`** e incluso **espacios en blanco y otros símbolos** si se escribe el **nombre de la variable entre comillas**. Ej.: `@'mi var'`, `@"mi-var"`, …
- ✓ **No se distingue mayúsculas de minúsculas**.
- ✓ Su **nombre** puede tener una **longitud máxima** de **64 caracteres**.
- ✓ **No es necesario declararlas ni inicializarlas** (se pueden usar directamente). Si la variable no se ha inicializado, tendrá asignado el valor `NULL` por defecto. Ej.: `SELECT @prueba; -- Devolverá NULL`
- ✓ **No llevan un tipo de datos asociado** (tipado débil).
- ✓ Para **asignarles un valor** se emplean las sentencias **`SET`** (operadores de asignación `=` o `:=`) y **`SELECT`** (haciendo uso de la cláusula **`INTO`**, tener en cuenta que no se mostrará el resultado de la consulta y que si devuelve más de un valor se producirá un error o bien un aviso si no devuelve ningún valor).
- ✓ Son variables que **se pueden acceder desde cualquier parte** (procedimiento, función, *trigger*, evento, …), vienen a ser como variables globales en la programación tradicional y **tendrán un valor diferente en sesiones diferentes**, entendiéndose una sesión como una conexión al *SGBD*.

Ejemplo:

```
SET @nombre = "José"; -- @nombre := "José"
SELECT "Sánchez" INTO @apellidos;
SELECT CONCAT(@apellidos, ", ", @nombre);
```

Resultado #1 (1r × 1c)

CONCAT(@apellidos, ", ", @nombre)
Sánchez, José

Las **variables se pueden usar para guardar información de una sentencia `SELECT`**, pero es necesario **asegurarse de que únicamente devuelva una fila**. Se pueden utilizar varias variables a la vez dentro de una sentencia *SQL*.

Ejemplo: uso de la cláusula `INTO` dentro del `SELECT`.

```
SELECT nombreSala, capacidad INTO @nombre_sala, @capacidad
FROM sala
WHERE nombreSala = "Afrodita";
SELECT @nombre_sala, @capacidad;
```

Resultado #1 (1r × 2c)

@nombre_sala	@capacidad
Afrodita	80

En este ejemplo `INTO` utiliza dos variables. Cada **columna del `SELECT`** (y en el orden en el que se encuentran) se va a **corresponder con una variable del `INTO`** (en el mismo orden que el `SELECT`). El número de columnas y variables debe ser el mismo.

Ejemplo:

```
SET @min = 100, @max = 200;
SELECT *
FROM sala
WHERE capacidad BETWEEN @min AND @max;
```

sala (2r × 2c)

nombreSala	capacidad
Apolo	200
Hermes	150

Se puede hacer uso de las funciones proporcionadas por el *SGBD* para realizar cualquier modificación sobre los datos de las variables y después emplearlas en otras sentencias *SQL*.

Ejemplos:

```
SELECT nombre, apellido1, apellido2 INTO @nombre, @apellido1, @apellido2
FROM ponente
WHERE idPonente = "USA001";
SET @nombre_completo = CONCAT(@nombre, ", ", CONCAT_WS(" ", @apellido1, @apellido2));
SELECT @nombre_completo AS nombre_completo;
SELECT AVG(precio) INTO @precio_medio
FROM conferencia;
```

Resultado #1 (1r × 1c)

nombre_completo
Stephen, Gary

```
SET @nombre_sala = "Afrodita";
SELECT *
FROM conferencia
WHERE nombreSala = @nombre_sala COLLATE utf8mb4_spanish_ci;
```

```
SELECT *
FROM conferencia
WHERE precio > @precio_medio;
```

conferencia (2r × 6c)

idConferencia	tema	precio	fecha	turno	nombreSala
PO01314	Programación Orientada a Objetos	20,0	2013-10-02	M	Zeus
PWB1314	Programación Web	18,5	2013-10-02	T	Apolo

Se pueden emplear variables en cualquier sentencia *SQL*, no tiene por qué ser solo SELECT.

Ejemplo:

```
SELECT nombreSala INTO @sala_sin_conferencias
FROM sala
WHERE nombreSala NOT IN (SELECT nombreSala FROM conferencia)
LIMIT 1; -- Asegura que no se devuelve más de una fila
INSERT INTO conferencia VALUES("SIS1314", "Sistemas Informáticos", 10.0, DATE(NOW()),
                               "M", @sala_sin_conferencias);
SELECT * FROM conferencia WHERE nombreSala = @sala_sin_conferencias;
```

Las **variables definidas por el usuario se pueden usar en la mayoría de las declaraciones y cláusulas** que aceptan una expresión *SQL*. **Sin embargo, hay algunas excepciones**, como la cláusula **`LIMIT`**.

conferencia (1r × 6c)

idConferencia	tema	precio	fecha	turno	nombreSala
SIS1314	Sistemas Informáticos	10,0	2023-03-27	M	Hermes

Otra excepción, es que **no se pueden usar variables en los nombres de *BD*, nombres de tablas, nombres de columnas, etc.** Para usar una variable en estos casos, **se debe usar PREPARE en una cadena para construir una sentencia *SQL* dinámica** (se verán en el punto 3 de esta unidad).

Ejemplo:

```
SET @nombre_bd = 'mi_bd';
SET @sent_sql = CONCAT('CREATE DATABASE ', @nombre_bd, ' COLLATE utf8mb4_spanish_ci;');
PREPARE sentencia FROM @sent_sql; -- https://mariadb.com/kb/en/prepare-statement/
EXECUTE sentencia; -- https://mariadb.com/kb/en/execute-statement/
DEALLOCATE PREPARE sentencia; -- https://mariadb.com/kb/en/deallocate-drop-prepare/
```

1.2 Variables locales dentro de un procedimiento, función o *trigger*.

Las **variables de sesión pueden emplearse en cualquier lugar**, incluso dentro de procedimientos, funciones y *triggers*, aunque no es lo habitual.

Dentro de un procedimiento, función o *trigger*, … las variables se **definen** haciendo uso de la **sentencia DECLARE**:

```
DECLARE nom_var1 [, nom_var2 ...] [[ROW] TYPE OF]] tipo_dato [DEFAULT valor_por_defecto];
```

A diferencia de las variables de sesión:

```
DECLARE v_idPonente TYPE OF ponente.idPonente;
```

- ✓ El **nombre** de estas variables **no empieza por @**.
- ✓ Se debe **declarar el tipo asociado a la variable**, pero esto no impide que se le pueda asignar un dato que no se corresponda con el tipo, haciendo la conversión (siempre que se pueda).
- ✓ Si **no se inicializan**, su **valor por defecto** es **NULL** (si falta la cláusula DEFAULT, el valor inicial es NULL).

Ejemplo: `DECLARE salida VARCHAR(30) DEFAULT 'Hola mundo';`

En el ejemplo se está declarando una variable local de nombre *salida*, tipo de datos `VARCHAR(30)` y con valor por defecto '*Hola mundo*' (si no se utiliza DEFAULT, el valor por defecto sería NULL).

Para **asignar un valor** a una variable se utilizan **SET** o **INTO** de alguna de las siguientes **formas**:

- ✓ **`SET variable := valor;`**
- ✓ **`SET variable = valor;`**
- ✓ **`SELECT expresión INTO variable FROM tabla …;`**

A diferencia de las variables de sesión, **estas variables son locales** al procedimiento, función, *trigger*, … **donde estén definidas**. El **alcance de una variable local es el bloque dentro del cual se declara (`BEGIN ... END`)**.

Los **tipos de datos** que se pueden emplear son **los mismos que** se emplearon **para definir las columnas de las tablas** (ver unidad 3).

IMPORTANTE. Cuando se declaren variables locales, se debe tener en cuenta lo siguiente:

- ✓ **No declarar nombres de variables que coincidan con nombres de columnas** (darles un nombre diferente).
- ✓ Si se emplea la **variable** para guardar el resultado de una consulta o se va a emplear como dato para realizar un INSERT, UPDATE o DELETE, se debe **declararla con el mismo tipo y tamaño que la columna a la que va a hacer referencia en la sentencia *SQL***.

Ejemplo: si se quiere guardar el nombre de una sala con `SELECT … INTO …` en una variable, dicha variable debería estar declarada como `VARCHAR(50)` que es el tipo de dato y tamaño de la columna `nombreSala` de la tabla `sala`.

A continuación se van a poner ejemplos de uso de las variables locales en procedimientos y funciones.
No preocuparse si no se entiende ya que se verá en los próximos puntos de la unidad.

Ejemplo: uso de variables locales dentro de una función.

```
DELIMITER $$
CREATE OR REPLACE FUNCTION hola_mundo() RETURNS VARCHAR(30)
BEGIN
    DECLARE v_salida VARCHAR(30) DEFAULT 'Hola mundo';
    SET v_salida = 'Hola mundo con variables';
    RETURN v_salida;
END$$
DELIMITER ;
```

```
1  SELECT hola_mundo() AS mensaje;
```

Resultado #1 (1r × 1c)

mensaje
Hola mundo con variables

La variable `salida` es local y será destruida una vez finalice la ejecución de la función. El uso de `DEFAULT` en `DECLARE` asigna un valor por defecto al declarar la variable.

Ejemplo: uso de parámetros en funciones.

```
DROP FUNCTION IF EXISTS hola_mundo;
DELIMITER $$
CREATE OR REPLACE FUNCTION hola_mundo(p_entrada VARCHAR(20)) RETURNS VARCHAR(50)
BEGIN
    DECLARE v_salida VARCHAR(50);
    SET v_salida = CONCAT('Hola mundo ', p_entrada, '!!!');
    RETURN v_salida;
END$$
DELIMITER ;
```

```
1  SELECT hola_mundo("BD") AS mensaje;
```

Resultado #1 (1r × 1c)

mensaje
Hola mundo BD!!!

Ejemplo: función que acepta un dividendo y un divisor, y realiza la división entera (`dividendo DIV divisor`) sin usar el operador división `DIV` (la sentencia `WHILE` se verá más adelante en la unidad).

```
DELIMITER $$
CREATE OR REPLACE FUNCTION divide(p_dividendo INT, p_divisor INT) RETURNS INT
BEGIN
    DECLARE v_aux, v_contador, v_resto INT;
    SET v_contador = 0;
    SET v_aux = 0;
    WHILE (v_aux + p_divisor) <= p_dividendo DO
        SET v_aux = v_aux + p_divisor;
        SET v_contador = v_contador + 1;
    END WHILE;
    SET v_resto = p_dividendo - v_aux;
    RETURN v_contador;
END$$
DELIMITER ;
```

```
1  SELECT divide(20,2);
```

Resultado #1 (1r × 1c)

divide(20,2)
10

Ejemplo: uso de variables locales en un procedimiento dentro de una sentencia `SELECT`.

```
DELIMITER $$
CREATE OR REPLACE PROCEDURE salas_entre_1_var1_capacidad(p_var1 INT)
BEGIN
    DECLARE v_inicio INT UNSIGNED DEFAULT 1;
    SELECT p_var1, v_inicio;
    SELECT *
     FROM sala
     WHERE capacidad BETWEEN v_inicio AND p_var1;
END$$
DELIMITER ;

CALL salas_entre_1_var1_capacidad(180); -- Llamada al procedimiento
```

Resultado #1 (1r × 2c) | sala (2r × 2c)

var1	inicio
180	1

Resultado #1 (1r × 2c) | sala (2r × 2c)

nombreSala	capacidad
Afrodita	80
Hermes	150

1.3 Variables del *SGBD MariaDB* (prefijo @@).

El *SGBD MariaDB* mantiene muchas **variables del sistema** que pueden ser **de tipo GLOBAL o SESSION**.

Las **variables del sistema globales afectan al funcionamiento general del *SGBD***, mientras que las **variables de sesión afectan** al funcionamiento de las **conexiones de clientes individuales**.

Para **ver los valores usados por el servidor**, usar la **sentencia `SHOW VARIABLES` o `SELECT @@nombre_variable`**.
Ejemplos:

```
SHOW VARIABLES LIKE '%wait_timeout%'; -- SHOW VARIABLES; muestra todas (de sesión)
SELECT @@global.sql_mode, @@local.sql_mode, @@sql_mode; -- Las dos últimas de sesión
```

Se pueden configurar al iniciar el *SGBD* usando opciones en la línea de comando o en un archivo de configuración (`my.ini`). **La mayoría de ellas se pueden cambiar dinámicamente mientras el servidor se está ejecutando usando `SET GLOBAL` o `SET SESSION`**:

```
-- Establecer valores de variables globales
SET GLOBAL innodb_lock_wait_timeout = 100;
SET @@global.innodb_lock_wait_timeout = 100;
-- Establecer valores de variables de sesión
SET innodb_lock_wait_timeout = 100;
SET SESSION innodb_lock_wait_timeout = 100;
SET @@innodb_lock_wait_timeout = 100;
SET @@session.innodb_lock_wait_timeout = 100;
```

```
SET variable_assignment [, variable_assignment] ...
variable_assignment:
      user_var_name = expr
    | [GLOBAL | SESSION] system_var_name = expr
    | [@@global. | @@session. | @@]system_var_name = expr

user_var_name:= expr
```

2 Procedimientos.

Los procedimientos almacenados **son un conjunto de sentencias *SQL* más una serie de estructuras de control que proveen de cierta lógica al procedimiento** (permiten asociar un nombre a un conjunto de sentencias *SQL*). Estos procedimientos **están guardados en el *SGBD* y son accedidos a través de llamadas `CALL`.**

El **uso de procedimientos almacenados proporciona**:

- ✓ **Más rapidez**: las **sentencias *SQL* que conforman el procedimiento están pre-compiladas** y el *SGBD* **no tiene que verificar que estén bien escritas** (como tiene que hacer si se emplean sentencias *SQL* directamente).
- ✓ **Más simplicidad en la seguridad**: se puede especificar **quien tiene permiso para ejecutar el procedimiento**. Si se usan las sentencias *SQL* directamente se tiene que establecer la seguridad a nivel de tabla/columna por lo que es más complejo (quien tiene permiso para realizar qué operación y con qué columnas).
- ✓ **Más seguridad**: permiten **evitar acciones que comprometan la seguridad de la *BD***, como la inyección de código *SQL* en formularios.
- ✓ **Estándar**: existe un único lugar centralizado (el *SGBD*) donde se encuentran todos los procedimientos almacenados (asociados a una *BD*) y por lo tanto todos los **programadores harán uso de los procedimientos creados evitando duplicidades** de código.
- ✓ **Velocidad**: el código ya se encuentra todo en el lado del servidor, **no es necesario que los clientes envíen todas las sentencias, solamente los datos que van a necesitar los procedimientos** para su funcionamiento.

```
CREATE
    [OR REPLACE]
    [DEFINER = { user | CURRENT_USER | role | CURRENT_ROLE }]
    PROCEDURE [IF NOT EXISTS] sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

type:
    Any valid MariaDB data type

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

routine_body:
    Valid SQL procedure statement
```

Para crear procedimientos, ***MariaDB* ofrece la sentencia `CREATE PROCEDURE`**. Al crear un procedimiento, éste, es **asociado a la *BD* en uso** (activa), por ello es conveniente seleccionarla haciendo uso de la sentencia USE.

Un procedimiento almacenado **puede estar formado por sentencias de todo tipo salvo** las que se tratan en el siguiente **enlace** https://mariadb.com/kb/en/stored-routine-limitations/ (`ALTER VIEW`, `BEGIN`, `LOCK TABLES`, …).

La **sintaxis** básica para la creación de un procedimiento almacenado es la siguiente:

```
CREATE [OR REPLACE] PROCEDURE [IF NOT EXISTS] nombre_procedimiento ([parámetro[s]])
[características]
[BEGIN]
    definición      -- Sentencias SQL, estructuras de control, declaración
                    -- de variables locales, control de errores, etc.
[END]
```

Una **forma de indicar en qué *BD* debe crearse el procedimiento es anteponiendo al nombre del mismo, el nombre de la *BD* seguido por un punto**, de la forma:

```
CREATE [OR REPLACE] PROCEDURE nombre_bd.nombre_procedimiento([parámetro[s]])
...
```

Como **mínimo se debe indicar un nombre** para el procedimiento **y una definición**. El **conjunto de sentencias** va en la **definición**, si la forman **varias deben ir entre las etiquetas BEGIN y END**.

Un **procedimiento almacenado tiene**:

- Un **nombre**.
- Opcionalmente una **lista de parámetros**.
- Un **contenido** (sección también llamada **definición del procedimiento**) donde se especifica qué es lo que va a hacer y cómo. Puede estar compuesto por **sentencias *SQL*, estructuras de control, declaración de variables locales, control de errores, etc**.

Dentro de **características** es posible **incluir comentarios o definir si el procedimiento obtendrá los mismos resultados ante entradas iguales**, entre otras cosas.

```
CREATE [OR REPLACE] PROCEDURE [IF NOT EXISTS] nombre_bd.nombre_procedimiento([parámetro[s]])
COMMENT 'Objetivo del procedimiento'
BEGIN
    ...
```

***MariaDB* toma como delimitador de sentencias *SQL* el punto y coma por defecto. Quiere decir que cada vez que encuentra un punto y coma intenta ejecutar la sentencia *SQL*.** Se debe **ver al procedimiento almacenado como un todo, como si fuera una única sentencia** que se quiere que *MariaDB* ejecute entera.

¿Pero qué pasa si se quiere poner una sentencia *SQL* dentro del procedimiento almacenado? Que *MariaDB* intentará ejecutarla y no la interpretará como parte del procedimiento almacenado.

Para evitarlo, se debe **indicar con la sentencia DELIMITER cuál es el símbolo que debe encontrar *MariaDB* para ejecutar la sentencia *SQL*** (en nuestro caso el CREATE PROCEDURE) y por lo tanto **dentro del mismo ya se podrán poner sentencias *SQL* acabadas en punto y coma** para separar unas de otras.

Cuando se acaba de definir el procedimiento almacenado se vuelve a indicar que el delimitador sea ';' por si se quiere ejecutar a continuación otras sentencias *SQL*.

El **delimitador** puede ser **cualquiera que no se vaya a emplear dentro del procedimiento almacenado**.

Ejemplo: procedimiento que devuelve el nombre y apellidos de los ponentes separados por coma con el formato "apellido1 apellido2, nombre" ordenados de forma descendente.

```
USE conferencias2;
DROP PROCEDURE IF EXISTS listar_ponentes;
DELIMITER $$
CREATE PROCEDURE listar_ponentes()
COMMENT 'Devuelve el nombre completo de todos los ponentes ordenado descendentemente'
BEGIN
    SELECT CONCAT(CONCAT_WS(' ', apellido1, apellido2),' ,',nombre) AS nombre_completo
    FROM ponente
    ORDER BY nombre_completo DESC;
END$$
DELIMITER ;
```

Para **llamar a un procedimiento** se utiliza la **instrucción CALL**. Desde un procedimiento se puede invocar a su vez a otros procedimientos o funciones. Cada vez que se llama a un procedimiento, se ejecutan todas sus sentencias.
Sintaxis: `CALL [nombre_bd.]nombre_procedimiento([parámetros]);`

Delante del nombre del procedimiento almacenado se puede indicar el nombre de la *BD*. Esto no es necesario si se está ejecutando el procedimiento con la *BD* seleccionada o ejecutando la sentencia **USE nombre_bd;** antes de la llamada al procedimiento.

Ejemplo: llamar al procedimiento almacenado `listar_ponentes()`.

```
CALL listar_ponentes(); -- conferencias2.listar_ponentes() para indicar la BD
```

La **sentencia que permite listar los procedimientos almacenados** es:

```
SHOW PROCEDURE STATUS [LIKE 'pattern' | WHERE expr];
```

Resultado #1 (8r × 1c)

nombre_completo
Soriano López ,Luisa
Shull ,Kevin
Llobregat García ,Juan
Gaul More ,Piere
Gary ,Stephen
Durán Céspedes ,Silvia
Craig ,Robert
Amor Pérez ,Julián

Ejemplo: ver la información de los procedimientos almacenados de la *BD* "*conferencias2*".

```
SHOW PROCEDURE STATUS WHERE db = 'conferencias2';
```

Para **ver el código de un procedimiento** almacenado se puede hacer uso de la sentencia:

`SHOW CREATE PROCEDURE [nombre_bd.]nombre_procedimiento;`

Ejemplo: ver el código del procedimiento almacenado creado anteriormente.

```
SHOW CREATE PROCEDURE listar_ponentes; -- conferencias2.listar_ponentes para indicar la BD
```

En *MariaDB* **no se permite modificar los parámetros y el cuerpo de un procedimiento**, se tendrá que borrar y volver a crear. Lo que **si se permite** es **modificar las características** del mismo mediante la sentencia **ALTER PROCEDURE**.

Ejemplo: modificar el comentario del procedimiento creado anteriormente.

```
ALTER PROCEDURE conferencias2.listar_ponentes COMMENT 'Muestra listado de todos los ponentes';
SHOW PROCEDURE STATUS WHERE db = 'conferencias2';
```

La sentencia que permite **borrar un procedimiento** es:

`DROP PROCEDURE [IF EXISTS] [nombre_bd.]nombre_procedimiento;`

Ejemplo: borrar el procedimiento creado anteriormente.

```
DROP PROCEDURE IF EXISTS conferencias2.listar_ponentes;
```

La **cláusula `IF EXISTS` indica que solamente en caso de que exista**, borre el procedimiento. Si el **procedimiento no existe daría un error y se dejarían de ejecutar las sentencias**.

Cuando se llama a un procedimiento puede ser necesario 'enviarle' información, cada uno de los datos que se envía a un procedimiento almacenado se denomina **parámetro**. Los parámetros son **variables cuyo valor se utiliza dentro del procedimiento** almacenado.

Puede haber más de un parámetro (se separan con comas) o puede **no haber ninguno (**en este caso **deben seguir presentes los paréntesis**, aunque no haya nada dentro**)**.

Los **parámetros tienen la siguiente estructura: `[modo] nombre_parámetro tipo_dato`**

- ✓ **`modo`**: **`IN`** (**por defecto**, si no se indica nada es el modo que se asignará, son los parámetros que el procedimiento recibirá), **`OUT`** (son los parámetros que el procedimiento podrá modificar) e **`INOUT`** (mezcla de los dos anteriores).
- ✓ **`nombre_parámetro`**: es el **nombre del parámetro**.
- ✓ **`tipo_dato`**: es cualquier tipo de dato de los provistos por *MariaDB* (ver unidad 3).

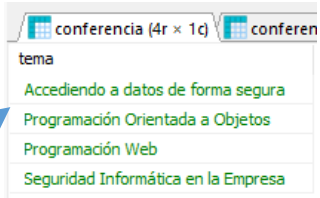
Se debe **tener cuidado con el nombre de los parámetros y que no coincidan con el nombre de las columnas de una tabla**, ya que *MariaDB* **interpretará siempre que se trata del nombre del parámetro** y nunca el de la columna. Esto se puede **solucionar** referenciando el nombre de la columna de la forma: **`nombre_tabla.nombre_columna`**

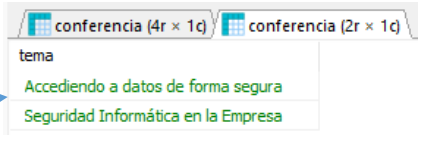
Ejemplo: procedimiento que devuelva las conferencias que da un ponente. Llevará como parámetro el *id* del ponente.

```
USE conferencias2;
DROP PROCEDURE IF EXISTS listar_conferencias_ponente_por_id;
DELIMITER $$
CREATE PROCEDURE listar_conferencias_ponente_por_id(p_id CHAR(6))
COMMENT 'Devuelve las conferencias que da un ponente. Llevará como parámetro el id del ponente.'
BEGIN
   SELECT DISTINCT tema
   FROM conferencia JOIN participa USING (idConferencia)
   WHERE idPonente = p_id
   ORDER BY tema;
END$$
DELIMITER ;
```

Ejemplos de llamadas:

```
CALL listar_conferencias_ponente_por_id('USA001');
SET @id_buscar='ESP004';
CALL listar_conferencias_ponente_por_id(@id_buscar);
```





Ejemplo: procedimiento al que se le pase el nombre de una sala y devuelva en forma de parámetro de salida su capacidad.

```
USE conferencias2;
DROP PROCEDURE IF EXISTS obtener_sala_capacidad;
DELIMITER $$
CREATE PROCEDURE obtener_sala_capacidad(p_sala VARCHAR(50), OUT p_capacidad SMALLINT)
    COMMENT 'Devuelve en p_capacidad la capcidad de la sala indicada por p_sala'
BEGIN
    SELECT capacidad INTO p_capacidad FROM sala WHERE nombreSala = p_sala;
END$$
DELIMITER ;
```

Ejemplo de llamada:

```
CALL obtener_sala_capacidad('Afrodita', @capacidad);
SELECT @capacidad;
```

Resultado #1 (1r × 1c)
@capacidad
80

Ejemplo: procedimiento al que se le envíe como parámetros el nombre de la sala y una cantidad que representa el incremento de la capacidad y actualice la capacidad de dicha sala.

```
USE conferencias2;
DROP PROCEDURE IF EXISTS incrementar_capacidad_sala;
DELIMITER $$
CREATE PROCEDURE incrementar_capacidad_sala(p_sala VARCHAR(50), INOUT p_inc_capacidad SMALLINT)
    COMMENT 'Devuelve en p_inc_capacidad la nueva capacidad incrementada en la sala p_sala'
BEGIN
   UPDATE sala SET capacidad = capacidad + p_inc_capacidad WHERE nombreSala = p_sala;
   SELECT capacidad INTO p_inc_capacidad FROM sala WHERE nombreSala = p_sala;
END$$
DELIMITER ;      GRANT CREATE ROUTINE, ALTER ROUTINE, EXECUTE ON nombre_bd.* TO usuario@'hostname';
```

Ejemplo de llamada:

```
SET @dato = 50;       -- Incremento de capacidad
CALL incrementar_capacidad_sala('Afrodita', @dato);
SELECT @dato;
```

Más información sobre la sentencia CREATE PROCEDURE *en* https://mariadb.com/kb/en/create-procedure/

TAREA

1. **Crear un procedimiento que devuelva las conferencias que da un ponente. Recibirá como parámetros el nombre, apellido1 y apellido2 del ponente. Suponer que el nombre y los apellidos no se pueden repetir.**
2. **Crear un procedimiento que devuelva los datos del/os ponente/s que dan más conferencias de las indicadas en un parámetro de entrada que se le pasa.**
3. **Crear un procedimiento que inserte una nueva sala. Los parámetros que se le pasen deben ser del mismo tipo y tamaño que el de columnas de la tabla *sala*.**
4. **Crear un procedimiento que permita modificar los datos de una conferencia (no se permite actualizar su clave primaria). Los parámetros que se le pasen deben ser de los tipos y tamaños adecuados.**
5. **Crear un procedimiento que borre una sala dado su nombre. La sala se debe eliminar haciendo uso del patrón *nombre%* empleando para ello el operador LIKE.**
6. **Crear un procedimiento al que se le pase el *idPonente* de un ponente y devuelva en forma de parámetro de salida en cuantas conferencias participa.**
7. **Crear un procedimiento al que se le pase como parámetro el *idPonente* y devuelva en el mismo parámetro el nombre completo del ponente con el formato: "*apellido1 apellido2, nombre*".**

3 Sentencias preparadas (*prepared statements*).

Imaginar que se quieren dar de alta 20 asistentes en momentos diferentes, se tendrían que enviar 20 sentencias INSERT con los datos de cada uno de los asistentes. El *SGBD* tendrá que analizar esa misma sentencia para cada asistente, comprobar si está bien escrita y ejecutarla.

Con las **sentencias preparadas (PREPARED)**, lo que se hace es **enviar al servidor la sentencia *SQL* con parámetros,**

pero sin datos. El ***SGBD* la analiza una sola vez para comprobar que no tiene errores sintácticos** y una vez hecho esto, solamente se tendrían que enviar los datos con los que la sentencia va a ser ejecutada.

El uso de sentencias preparadas **va a permitir**:

- ✓ Una **mayor velocidad de ejecución**, ya que las **sentencias están pre-compiladas**.
- ✓ Una **mayor seguridad**, ya que **impiden la inyección de código *SQL***.
- ✓ La posibilidad de **crear sentencias *SQL* dinámicas**, es decir, que se pueden construir las sentencias como si fueran una cadena a la que se va a añadiendo partes hasta tenerla completa y ejecutarla (esta es la mayor ventaja si se usan en procedimientos almacenados).

Los **pasos** a seguir para emplear este tipo de sentencias son:

```
PREPARE stmt_name FROM preparable_stmt
```

- ✓ **PREPARE**: se **define la sentencia *SQL* con parámetros** que se quiere ejecutar (por ejemplo, INSERT lleva como parámetros los valores a añadir o SELECT puede llevar parámetros en el WHERE).
- ✓ **EXECUTE**: para **ejecutar la sentencia**.
  - Paso de datos a los parámetros que están definidos en la sentencia.

```
EXECUTE stmt_name
    [USING expression[, expression] ...]
```

- ✓ **DEALLOCATE PREPARE**: para **liberar la memoria** utilizada por el *SGBD* para la sentencia preparada.

```
{DEALLOCATE | DROP} PREPARE stmt_name
```

La **sentencia PREPARE es visible a nivel de sesión**, hasta que no se hace un DEALLOCATE PREPARE.

Ejemplo: eliminar el usuario "pepe".

```
SET @bd_user = "pepe";
SET @sql = CONCAT('DROP USER IF EXISTS ', @bd_user, ';');
PREPARE stmt FROM @sql;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

Dentro de la definición, **?** se puede utilizar como **marcador de parámetro** para indicar dónde se vincularán los valores de los datos a la sentencia cuando se ejecute. ? **no debe ir entre comillas**, incluso si se desea vincularlos a valores de cadena.

Ejemplo: genérico (no aplicable sobre la *BD* "*conferencias2*").

```
SET @sentencia = 'SELECT * FROM tabla WHERE col1 = ? AND col2 = ?';
-- Se prepara la sentencia. El nombre sent_preparada puede ser el que se quiera
PREPARE sent_preparada FROM @sentencia;
-- Se usan dos variables para establecer los valores que se le van a enviar como parámetros
SET @valor_parametro1 = 10;
SET @valor_parametro2 = 'Luis Carlos';
EXECUTE sent_preparada USING @valor_parametro1, @valor_parametro2; -- Se ejecuta la sentencia
DEALLOCATE PREPARE sent_preparada; -- Se libera la memoria
```

Ejemplo: crear un procedimiento que devuelva la lista de ponencias filtradas por columnas dinámicamente. Los datos a enviar será el nombre de la columna, el tipo de operación y el valor que debe cumplir. Por ejemplo, 'turno','=','M'.

```
USE conferencias2;
DELIMITER $$
CREATE PROCEDURE conferencias_filtro(p_nom_col VARCHAR(20), p_oper CHAR(2), p_val_par VARCHAR(10))
   COMMENT 'Devuelve las conferencias que cumplan la condición que se le pasa como parámetro.'
BEGIN
   -- Se podría poner como tercer parámetro directamente el parámetro p_val_par
   DECLARE v_consulta VARCHAR(200);
   SET v_consulta = CONCAT('SELECT * FROM conferencia WHERE ', p_nom_col, p_oper, '?');
   PREPARE prep_consulta FROM v_consulta;
   EXECUTE prep_consulta USING p_val_par;
   DEALLOCATE PREPARE prep_consulta;
END$$
DELIMITER ;
```

Ejemplos de llamadas:

```
CALL conferencias_filtro('turno', '=', 'M');
CALL conferencias_filtro('precio', '>=', '15');
```

TAREA

8. **Crear una sentencia preparada que devuelva las conferencias en las que participa un ponente. La sentencia se ejecutará utilizando como parámetros el nombre y apellidos del ponente. No olvidar liberar la memoria utilizada por la sentencia preparada. No es necesario crear un procedimiento almacenado.**
9. **Crear un procedimiento que devuelva el número de conferencias indicado en un parámetro. También se le pasará otro parámetro con las columnas que se deben mostrar de la forma '*col1, col3, col5*'. Poner varios ejemplos de llamada al procedimiento. ¿Se podría hacer sin utilizar sentencias preparadas?**

4 Guiones.

Los guiones o scripts son un conjunto de instrucciones, que, ejecutadas de forma ordenada, realizan operaciones avanzadas o de mantenimiento de los datos almacenados en la *BD* (permiten la automatización de tareas).

Todas las **instrucciones serán ejecutadas por el *SGBD* y afectarán a la *BD* 'activa'**, o bien podrá ser seleccionada de forma explícita la *BD* haciendo uso de USE o poniendo en el FROM el nombre de la *BD* de la forma: `nombre_bd.nombre_tabla`.

En un guion, entre otras instrucciones, **pueden aparecer**:

- ✓ **Consultas**.
- ✓ **Procedimientos**.
- ✓ **Funciones**.
- ✓ **Operaciones de manipulación de datos**.
- ✓ **Sentencias *SQL* asociadas a los permisos y seguridad**.
- ✓ etc.

Por lo tanto, un guion, se puede ver como un **conjunto de sentencias *SQL*** que pueden ser cualquiera de las anteriores y que normalmente estarán **dentro de un archivo** con la extensión **`.sql`**, siendo este un archivo de texto que se podrá editar con cualquier editor, y que contiene el conjunto de sentencias que se quieren ejecutar.

Es decir, un guion **puede estar construido con cualquier sentencia *DML*, *DCL*, *DDL*, *TCL* o *PLSQL*** junto con **otras instrucciones** que son extensiones y **dependen** del propio ***SGBD***.

Normalmente:

- ✓ Se emplearán los guiones **para importar y exportar *BD***, tanto su estructura como sus datos y demás componentes (procedimientos, funciones, disparadores, …).
- ✓ Se crearán **procedimientos, funciones o *triggers*** y **se hará uso de la programación procedimental** (declaración de variables, `IF`, `WHILE`, control de errores, …) en su implementación.

Ejemplo: guion que crea una *BD*, una tabla, se añaden datos y se consulta.

```
-- ----------------------------------------------------------
-- Base de datos: conferencias3
-- ----------------------------------------------------------
SET @bd_user = "conferencias3";

SET @sql = CONCAT('DROP USER IF EXISTS "', @bd_user, '"@"%";');
PREPARE stmt FROM @sql;
EXECUTE stmt;
SET @sql = CONCAT('CREATE USER "', @bd_user, '"@"%" IDENTIFIED BY "', @bd_user, '";');
PREPARE stmt FROM @sql;
EXECUTE stmt;

SET @sql = CONCAT('DROP DATABASE IF EXISTS ', @bd_user, ';');
PREPARE stmt FROM @sql;
EXECUTE stmt;

SET @sql = CONCAT('CREATE DATABASE IF NOT EXISTS ', @bd_user, ';');
PREPARE stmt FROM @sql;
EXECUTE stmt;

SET @sql = CONCAT('GRANT ALL PRIVILEGES ON ', @bd_user, '.* TO ', @bd_user, ';');
PREPARE stmt FROM @sql;
EXECUTE stmt;

DEALLOCATE PREPARE stmt;

USE conferencias3;  -- No se puede preparar la sentencia y ejecutarla, da error

-- Tabla: sala
DROP TABLE IF EXISTS sala;
CREATE TABLE IF NOT EXISTS sala (
  nombreSala VARCHAR(50) PRIMARY KEY,
  capacidad SMALLINT UNSIGNED
);

-- Datos para la tabla: sala
START TRANSACTION;
INSERT INTO sala VALUES ('Afrodita', 300), ('Hermes', 250);
COMMIT;

SET @nombre_sala = 'Hermes';
SELECT * FROM sala WHERE nombreSala = @nombre_sala;
```

NOTA: Indicar que **en *MariaDB***, a diferencia de otros *SGBD*, **no se pueden crear bloques anónimos de código**, es decir, un conjunto de instrucciones que forman un "bloque" (normalmente se indica con `BEGIN …… END`), dentro de los cuales es posible hacer uso de instrucciones `IF`, `WHILE`, definición de variables locales, …). Por eso **en los guiones que se han creado no se han empleado instrucciones para la programación procedimental**, como `IF`, `WHILE`, ...

5 Documentación.

Al igual que cuando se está programando, cuando se escriben sentencias *SQL* se pueden (y se deben) añadir comentarios sobre el código.

Para poder **añadir comentarios en *SQL* se tienen varias formas**. La **primera es la definida en el estándar *ANSI/SQL* que es mediante dos guiones al principio de una línea**:

```
-- Esto es un comentario
```

Esta forma de poner comentarios sirve para anular partes de sentencias. Por ejemplo, si se tiene la sentencia:

```
SELECT * FROM libros
WHERE id_autor = 1;
```

Se puede anular el filtro del autor de una forma sencilla comentándolo mediante los dos guiones:

```
SELECT * FROM libros;
-- WHERE id_autor = 1
```

Se pueden añadir **comentarios de varias líneas mediante: `/* … */`**.

```
/*
 * Comentario en varias líneas.
 * Compatible con el estándar SQL99
 */
```

Esta forma de añadir comentarios fue añadida en la revisión *SQL99* del lenguaje.

Existen otras formas adaptadas a diferentes *SGBD*. Es por ello que es posible encontrarse con comentarios en sentencias *SQL* de las **siguientes formas**:

```
{Comentario con llaves}
# Esto también es un comentario en MariaDB
REM Esto es un comentario
```

Aunque lo recomendable es definir los comentarios *SQL* mediante las dos formas estándar explicadas anteriormente.

La **documentación también se puede utilizar en la definición de procedimientos, funciones, *triggers* o eventos**. En estos casos, se puede poner un comentario delante de la definición con el siguiente **formato**:

```
/*********************************************************************
* NOMBRE {PROCEDIMIENTO | FUNCIÓN | TRIGGER | EVENTO}:               *
* FECHA CREACIÓN:                                                    *
* AUTOR:                                                             *
* TAREA A AUTOMATIZAR:                                               *
* PARÁMETROS REQUERIDOS:                                             *
* RESULTADOS PRODUCIDOS:                                             *
*--------------------------------------------------------------------*/
Código del procedimiento / función / trigger / evento – Comentarios de una línea o varias
```

Ejemplo: guion completo que incluye dos procedimientos que permiten dar de alta y eliminar salas respectivamente.

```
USE conferencias2;
/*********************************************************************
* NOMBRE PROCEDIMIENTO: anadir_sala                                  *
* FECHA CREACIÓN: 20/03/2024                                         *
* AUTOR: JL                                                          *
* TAREA A AUTOMATIZAR: Dar de alta una nueva sala.                   *
* PARÁMETROS REQUERIDOS: IN  p_nombre: nombre de la sala.            *
*                            p_capacidad: capacidad de la sala.      *
* RESULTADOS PRODUCIDOS: Número de filas añadidas. -1 si no añade ninguna. *
*--------------------------------------------------------------------*/
DELIMITER $$
CREATE PROCEDURE anadir_sala(p_nombre VARCHAR(50), p_capacidad SMALLINT)
    COMMENT 'Añade una nueva sala'
```

```
BEGIN
          INSERT INTO sala VALUES (p_nombre, p_capacidad);
          SELECT ROW_COUNT(); -- Devuelve el número de filas actualizadas, insertadas o eliminadas
END$$                     -- por la sentencia anterior.
/*************************************************************************
* NOMBRE PROCEDIMIENTO: borrar_sala                                      *
* FECHA CREACIÓN: 20/03/2024                                             *
* AUTOR: JL                                                              *
* TAREA A AUTOMATIZAR: Dar de baja una sala.                             *
* PARÁMETROS REQUERIDOS: IN  p_nombre: nombre de la sala a borrar.       *
* RESULTADOS PRODUCIDOS: Número de filas borradas. -1 si no borra ninguna.  *
*-------------------------------------------------------------------------*/
CREATE PROCEDURE borrar_sala (p_nombre VARCHAR(50))
    COMMENT 'Borra una sala'
BEGIN
      DELETE FROM sala WHERE nombreSala LIKE CONCAT(p_nombre, '%');
          SELECT ROW_COUNT();
END$$
DELIMITER ;
```

6 Control de flujo.

6.1 IF.

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

Para indicar alternativas binarias de ejecución dependiendo de ciertas condiciones. **Sintaxis**:

```
IF condición THEN lista_instrucciones
  [ELSEIF condición THEN lista_instrucciones] ...
  [ELSE lista_instrucciones]
END IF;
```

Su funcionamiento es el mismo que en cualquier lenguaje de programación.

Ejemplo: crear un procedimiento que recupere los datos de un ponente enviando un *idPonente*. En caso de que el ponente no exista, devolverá la cadena '*PONENTE NO ENCONTRADO*' sin utilizar un parámetro de salida. La cadena a devolver si existe el ponente tendrá el formato: "*ponente > idPonente: nombre apellido1 apellido2 - especialidad*".

```
USE conferencias2;
DROP PROCEDURE IF EXISTS obtener_ponente_por_id;
DELIMITER $$
CREATE PROCEDURE obtener_ponente_por_id(IN p_id CHAR(6))
    COMMENT 'Información ponente: ponente > idPonente: nombre apellido1 apellido2 - especialidad'
BEGIN
   DECLARE v_id_ponente CHAR(6);
   DECLARE v_nombre, v_apellido1, v_apellido2, v_especialidad VARCHAR(50);

   SELECT idPonente, nombre, apellido1, apellido2, especialidad
        INTO v_id_ponente, v_nombre, v_apellido1, v_apellido2, v_especialidad
    FROM ponente
    WHERE idPonente = p_id;

   IF (v_id_ponente IS NULL) THEN  -- No hay un ponente con ese id
      SELECT 'PONENTE NO ENCONTRADO' AS ponente;
   ELSE
      SELECT CONCAT('ponente > ', v_id_ponente, ': ', v_nombre, ' ', v_apellido1, ' ',
                    NVL(v_apellido2, ''), ' - ', v_especialidad) AS ponente;
   END IF;
END$$
DELIMITER ;
```

Ejemplos de uso:

```
CALL obtener_ponente_por_id('USA003');
```


```
CALL obtener_ponente_por_id('USA010');     -- No existe
```

6.2 CASE.

Para indicar varias alternativas de ejecución dependiendo de ciertas condiciones. **Sintaxis**:

```
CASE
    WHEN condición THEN lista_instrucciones
    [WHEN condición THEN lista_instrucciones] ...
    [ELSE lista_instrucciones]
END CASE;
```

La instrucción CASE **viene a ser como un IF – ELSEIF – ELSEIF – ELSEIF ...** pero de una forma mucho más legible.

Ejemplo: crear un procedimiento al que se le pase como parámetro un número del 1 al 7 y devuelva, en forma de parámetro de salida el nombre del día (lunes, martes, miércoles, ...). En caso de que el número esté fuera del rango debe devolver la cadena NULL.

```
USE conferencias2;
DROP PROCEDURE IF EXISTS obtener_dia;
DELIMITER $$
CREATE PROCEDURE obtener_dia(p_num_dia TINYINT, OUT p_nombre_dia VARCHAR(9))
   COMMENT 'Devuelve en p_nombre_dia el nombre del día indicado en p_num_dia'
BEGIN
   CASE p_num_dia
      WHEN 1 THEN
          SET p_nombre_dia = 'lunes';
      WHEN 2 THEN
          SET p_nombre_dia = 'martes';
      WHEN 3 THEN
          SET p_nombre_dia = 'miércoles';
      WHEN 4 THEN
          SET p_nombre_dia = 'jueves';
      WHEN 5 THEN
          SET p_nombre_dia = 'viernes';
      WHEN 6 THEN
          SET p_nombre_dia = 'sábado';
      WHEN 7 THEN
          SET p_nombre_dia = 'domingo';
      ELSE
          SET p_nombre_dia = NULL;
   END CASE;
END$$
DELIMITER ;
```

Ejemplos de uso:

```
CALL obtener_dia(6, @nombreDia);
SELECT @nombreDia;
```

Resultado #1 (1r × 1c)
@nombreDia
sábado

```
CALL obtener_dia(20, @nombreDia);
SELECT @nombreDia;
```

Resultado #1 (1r × 1c)
@nombreDia
(NULL)

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

Or:

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

```
DECLARE level varchar(20);
CASE
   WHEN value <= 4000 THEN
     SET level = 'Low';
   WHEN value > 4000 AND value <= 5000 THEN
     SET level = 'Medium';
   ELSE
     SET level = 'Other';
END CASE;
```

TAREA

10. **Crear un procedimiento que devuelva las conferencias (*tema*, *precio* y *turno*) que tienen lugar en la sala indicada como parámetro. En caso de que no haya conferencias deberá devolver "*SIN CONFERENCIAS*" y en el caso de que no exista la sala, "*LA SALA NO EXISTE*".**
11. **Crear un procedimiento que devuelva el número de conferencias que se celebraron en la fecha indicada como parámetro. En caso de que no hubiera conferencias ese día debe devolver -1.**
12. **Crear un procedimiento al que se le pase una capacidad como parámetro y devuelva, empleando el mismo parámetro, cuantas salas tienen una capacidad superior a la pasada. Al mismo tiempo debe mostrar los nombres de las salas. En caso de que no se encuentren salas, debe devolver -1 y mostrar el texto "*SIN SALAS*".**
13. **Crear un procedimiento al que se le pase como parámetros un *idPonente* y dos fechas, y devuelva en forma de parámetro de salida en cuantas conferencias participó ese ponente entre las dos fechas indicadas.**
    - **En caso de que el ponente no exista, el parámetro de salida debe devolver -1 y mostrarse la cadena "*NO EXISTE EL PONENTE*".**

- **En caso de que las fechas no tenga un formato correcto (`STR_TO_DATE(str,format)` devuelve `NULL`), el parámetro de salida debe devolver -1 y mostrarse la cadena "*FECHAS CON FORMATO INCORRECTO*".**
- **Si existe el ponente y hay conferencias, el procedimiento también debe mostrar el nombre de las conferencias (sin repetirse) ordenadas alfabéticamente.**

6.3 REPEAT.

La instrucción REPEAT permite **repetir unas acciones hasta cumplir una determinada condición**. **Sintaxis**:

```
[begin_label:] REPEAT
  lista_instrucciones
  UNTIL condición
END REPEAT [end_label];
```

IMPORTANTE: al trabajar con bucles, si no se realiza una **programación correcta**, se pueden dar **bucles infinitos**.

El uso es el mismo que en el caso de la programación estructurada, **se repite el conjunto de sentencias entre `REPEAT` y `END REPEAT` hasta que se cumpla la condición indicada en `UNTIL`**.

Ejemplo: crear un procedimiento que sume los números comprendidos entre los valores indicados en dos parámetros de entrada (incluidos ambos) y devuelva la suma en un parámetro de salida. En el caso de que el primer parámetro tenga un valor superior al segundo, se debe devolver -1 y mostrar la cadena "*EL PRIMER NÚMERO TIENE QUE SER INFERIOR AL SEGUNDO*".

```
DELIMITER $$
CREATE OR REPLACE PROCEDURE suma_entre_numeros(p_num_inf INT, p_num_sup INT, OUT p_suma_num BIGINT)
  COMMENT 'Suma los números entre un rango indicado y devuelve la suma en un parámetro de salida.'
BEGIN
   SET p_suma_num = 0;
   IF (p_num_sup < p_num_inf) THEN
      SET p_suma_num = -1;
      SELECT 'EL PRIMER NUMERO TIENE QUE SER INFERIOR AL SEGUNDO' AS error;
   ELSE
      REPEAT
         SET p_suma_num = p_suma_num + p_num_inf;
         SET p_num_inf = p_num_inf + 1;
      UNTIL p_num_inf > p_num_sup
      END REPEAT;
   END IF;
END$$
DELIMITER ;
```

Ejemplos de uso:

```
CALL suma_entre_numeros(3, 50, @suma);
SELECT @suma;
CALL suma_entre_numeros(60, 1, @suma);
SELECT @suma;
```

Resultado #1 (1r × 1c) / @suma / 1.272

Resultado #1 (1r × 1c) / Resultado #2 (1r × 1c) / error / EL PRIMER NUMERO TIENE QUE SER INFERIOR AL SEGU…

Resultado #1 (1r × 1c) / Resultado #2 (1r × 1c) / @suma / -1

Con REPEAT (y lo mismo es aplicable a `WHILE` y LOOP) **se puede hacer uso de etiquetas para salir del bucle (o empezar nuevamente) sin necesidad de esperar a que deje de cumplirse la condición indicada en `UNTIL`**. La idea es **etiquetar el `BEGIN` - `END`** de forma que se puedan utilizar las **instrucciones**:

- ✓ **`ITERATE label`: deja de ejecutar las sentencias que estén a continuación y vuelve al comienzo del bucle**.
- ✓ **`LEAVE label`: sale del bucle que se encuentra etiquetado con el `label` indicado**. Si no se pone `label`, saldría del procedimiento.

Sintaxis de la definición del bucle con etiquetas de esta forma:

```
label_bucle: REPEAT
      ...
UNTIL condición
END REPEAT label_bucle; -- En caso de tenerse las dos etiquetas se deben llamar igual
```

El **uso de etiquetas** sigue las siguientes **reglas**:

- ✓ **`begin_label` debe ir seguido de dos puntos**.
- ✓ **`begin_label` se puede dar sin `end_label`**. Si **`end_label` está presente, debe ser el mismo que `begin_label`**.
- ✓ **`end_label` no se puede dar sin `begin_label`**.
- ✓ Las **etiquetas pueden tener hasta 16 caracteres**.

Dentro del bucle se podrán utilizar las órdenes:

- ✓ **LEAVE label_bucle;** -- Sale del bucle
- ✓ **ITERATE label_bucle;** -- No ejecuta las sentencias que vengan después y vuelve al REPEAT.

6.4 WHILE.

La instrucción WHILE permite **repetir unas instrucciones hasta que se deje de cumplir una determinada condición indicada en el WHILE**. **Sintaxis**:

```
[begin_label:] WHILE condición DO
  lista_instrucciones
END WHILE [end_label];
```

Al igual que con REPEAT, **se puede hacer uso de etiquetas para poderlas utilizar con las instrucciones LEAVE o ITERATE**.

Ejemplo: crear un procedimiento al que se le pasen dos datos, un mes en letra (*ENERO*, *FEBRERO*, …) y un año en número. Tendrá que calcular el total de conferencias celebradas en el mes y año indicado, y devolver el dato en un parámetro de salida. En el caso de que el año indicado sea NULL, tomar el año actual. Si el mes no existe debe devolver -1 en el parámetro de salida.

```
DELIMITER $$
CREATE OR REPLACE PROCEDURE conferencias_por_mes(p_mes VARCHAR(10), p_ano SMALLINT, OUT p_num_conf INT)
  COMMENT 'Devuelve en un parámetro el total de conferencias celebradas en el mes y año indicado'
label_proc: BEGIN
   DECLARE v_mes_en_numero CHAR(2) DEFAULT '00';
   DECLARE v_dia TINYINT DEFAULT 1; -- Día del mes. Usado para hacer el bucle hasta el día 31
   DECLARE v_num_conferencias_por_dia INT;

   SET p_num_conf = 0;
   CASE p_mes
      WHEN 'ENERO' THEN SET v_mes_en_numero = '01';
      WHEN 'FEBRERO' THEN SET v_mes_en_numero = '02';
      WHEN 'MARZO' THEN SET v_mes_en_numero = '03';
      WHEN 'ABRIL' THEN SET v_mes_en_numero = '04';
      WHEN 'MAYO' THEN SET v_mes_en_numero = '05';
      WHEN 'JUNIO' THEN SET v_mes_en_numero = '06';
      WHEN 'JULIO' THEN SET v_mes_en_numero = '07';
      WHEN 'AGOSTO' THEN SET v_mes_en_numero = '08';
      WHEN 'SEPTIEMBRE' THEN SET v_mes_en_numero = '09';
      WHEN 'OCTUBRE' THEN SET v_mes_en_numero = '10';
      WHEN 'NOVIEMBRE' THEN SET v_mes_en_numero = '11';
      WHEN 'DICIEMBRE' THEN SET v_mes_en_numero = '12';
      ELSE
         SET p_num_conf = -1;
         LEAVE label_proc; -- Sale del procedimiento
   END CASE;

   IF (p_ano IS NULL) THEN
      SET p_ano = YEAR(CURDATE());
   END IF;
   bucle_dia: WHILE (v_dia <= 31) DO
        IF (DATE(CONCAT(p_ano, '-', v_mes_en_numero, '-', v_dia)) IS NULL) THEN
          -- No es una fecha válida por meses que pueden tener menos de 31 días
          LEAVE bucle_dia;
        END IF;
        SELECT COUNT(*) INTO v_num_conferencias_por_dia FROM conferencia
        WHERE fecha = CONCAT(p_ano, '-', v_mes_en_numero, '-', v_dia);
        SET v_dia = v_dia + 1;
        SET p_num_conf = p_num_conf + v_num_conferencias_por_dia;
   END WHILE bucle_dia;
END$$
DELIMITER ;
```

Ejemplos de uso:

```
-- Se pueden añadir nuevas entradas a la tabla para comprobar el funcionamiento
CALL conferencias_por_mes('OCTUBRE', 2013, @num_conferencias);
SELECT @num_conferencias;
-- Devuelve -1
CALL conferencias_por_mes('NO EXISTE', 2013, @num_conferencias);
SELECT @num_conferencias;
-- Buscar en el año actual --> No hay datos, añadir filas a la tabla para realizar pruebas
CALL conferencias_por_mes('ABRIL', NULL, @num_conferencias);
SELECT @num_conferencias;
```

6.5 LOOP.

La instrucción LOOP permite **repetir unas instrucciones hasta que se obligue a abandonar el bucle**. Sintaxis:

```
begin_label: LOOP
  lista_instrucciones – Debe tenerse alguna que permita salir del bucle
END LOOP end_label;
```

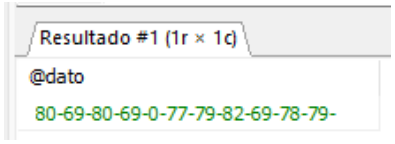
Vendría a ser equivalente a `WHILE (true)`. En este tipo de bucle **es necesario emplear la orden `LEAVE` para poder salir del mismo**. Se puede hacer uso de **etiquetas para emplear con las instrucciones `LEAVE` e `ITERATE`**.

Ejemplo: crear un procedimiento al que se le pase un dato en forma de cadena y devuelva en el mismo parámetro la cadena enviada cambiando las letras por números y separados por un guion.

```
DELIMITER $$
CREATE OR REPLACE PROCEDURE cambiar_formato(INOUT p_cadena VARCHAR(1000))
 COMMENT 'Devuelve en el mismo parámetro la cadena enviada cambiando las letras por números'
BEGIN
   DECLARE v_indice SMALLINT DEFAULT 1; -- Empleado para recorrer la cadena carácter a carácter
   DECLARE v_caracter CHAR(1); -- Guarda el carácter de la cadena
   DECLARE v_cadena VARCHAR(1000) DEFAULT ''; -- Cadena a devolver, default para poder concatenar
   bucle_loop: LOOP
      SET v_caracter = SUBSTRING(p_cadena, v_indice, 1);
      SET v_cadena = CONCAT(v_cadena, ORD(v_caracter), '-');
      SET v_indice = v_indice + 1;
      IF (v_indice > CHAR_LENGTH(p_cadena)) THEN
         LEAVE bucle_loop;
      END IF;
   END LOOP bucle_loop;
   SET p_cadena = v_cadena;
END$$
DELIMITER ;
```

Ejemplo de uso:

```
SET @dato = 'PEPE MORENO';
CALL cambiar_formato(@dato);
SELECT @dato;
```



TAREA

14. **Crear un procedimiento que devuelva en forma de parámetro de salida la suma de las capacidades de las salas enviadas como parámetro. El formato de la lista de salas es una única cadena: '*sala1, sala2, ...*'. Emplear las funciones `SUBSTRING` y `CHAR_LENGTH` para descomponer el dato en cada una de las salas (otras funciones que pueden resultar útiles son `TRIM` y `LOCATE`) y buscar la capacidad de cada una de ellas, devolviendo la suma de las capacidades de todas. En caso de que alguna sala no exista, se deberá mostrar la cadena '*La sala "XXXXXX" no existe*' por cada sala que no exista. Se debe controlar que al menos siempre se envíe una sala, en caso contrario debe de enviarse -1 en el parámetro de salida.**

7 Funciones.

Para crear una función, ***MariaDB* ofrece la sentencia `CREATE FUNCTION`**. La diferencia entre una función y un procedimiento es que la función devuelve valores. Estos **valores pueden ser utilizados como argumentos para sentencias *SQL***, tal como se hace con otras funciones como `MAX()` o `COUNT()`.

Se puede ver una **función** como un **procedimiento con un parámetro de tipo OUT**. Con respecto a los **procedimientos, la función se diferencia principalmente en que se pueden usar dentro de sentencias *SQL***, como por ejemplo un `SELECT`, de la forma `SELECT función(), col1 FROM Tabla` o `SELECT * FROM Tabla WHERE col = función()...` En estos casos, la función se va a llamar por cada fila de la tabla que cumpla los criterios de selección indiciados en la consulta `SELECT`.

La cláusula `RETURNS` es obligatoria al definir una función y sirve para especificar el tipo de dato devuelto.

Su **sintaxis** es:

```
CREATE FUNCTION nombre_función ([parámetro[s]]) RETURNS tipo [DETERMINISTIC]
    [características]
[BEGIN] definición [END]
```

Puede haber más de un parámetro (separados con comas) o ninguno (en este caso deben seguir presentes los paréntesis, aunque no haya nada dentro). Los **parámetros tienen la siguiente estructura: `[modo] nombre tipo`**

- ✓ **`modo`**: **`IN`** (**por defecto**), **`OUT`** e **`INOUT`**.
- ✓ **`nombre`**: es el nombre del parámetro.
- ✓ **`tipo`**: es cualquier tipo de dato de los provistos por *MariaDB* (ver unidad 3).

Dentro de **`características`** es posible incluir **comentarios o definir si la función devolverá los mismos resultados ante entradas iguales, entre otras cosas**.

`definición` es el **cuerpo de la función** y está compuesto por instrucciones, aquí se define qué hace, cómo lo hace, …

Para llamar a una función se invoca su nombre. Desde una función se puede invocar a su vez a otras funciones o procedimientos.

Una función se denomina **determinista si siempre va a devolver el mismo resultado al aplicar la misma entrada**. **Por defecto**, las **funciones** son consideradas como **no deterministas**.

```
CREATE [OR REPLACE]
    [DEFINER = {user | CURRENT_USER | role | CURRENT_ROLE }]
    [AGGREGATE] FUNCTION [IF NOT EXISTS] func_name ([func_parameter[,...]])
    RETURNS type
    [characteristic ...]
    RETURN func_body
func_parameter:
    [ IN | OUT | INOUT | IN OUT ]  param_name type
type:
    Any valid MariaDB data type
characteristic:
    LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
  | COMMENT 'string'
func_body:
    Valid SQL procedure statement
```

Otros modificadores que se pueden añadir a la función y que van **después del tipo de dato** son:

- ✓ **`CONTAINS SQL`**: la función **contiene al menos una sentencia *SQL*, pero no lee ni escribe ningún dato** almacenado en una *BD*.
- ✓ **`NO SQL`**: la función **no contiene sentencias *SQL***.
- ✓ **`READS SQL DATA`**: la función **contiene sentencias que leen datos** (SELECT) **pero no que modifiquen** datos.
- ✓ **`MODIFIES SQL DATA`**: la función **contiene sentencias que modifican datos** (INSERT, DELETE, UPDATE, REPLACE).

Ejemplo:
```
CREATE FUNCTION cuadrado(s SMALLINT) RETURNS SMALLINT DETERMINISTIC NO SQL
    RETURN s*s;
```
Ejemplo de uso:
```
SELECT cuadrado(2);
SET @valorDevuelto = cuadrado(2);
SELECT @valorDevuelto;
```

Ejemplo: crear una función a la que se le pase un número y devuelva el nombre del mes. En caso de que el número no se corresponda con ningún mes, debe devolver NULL.

```
DELIMITER $$
CREATE OR REPLACE FUNCTION mes_en_letra(p_mes TINYINT) RETURNS VARCHAR(10) DETERMINISTIC NO SQL
    COMMENT 'Devuelve el mes en letra que se corresponde con el número de mes'
BEGIN
   DECLARE v_mes_en_letra VARCHAR(10); -- Valor por defecto NULL, no es necesario poner DEFAULT
   CASE p_mes
          WHEN 1 THEN SET v_mes_en_letra = 'ENERO';
          WHEN 2 THEN SET v_mes_en_letra = 'FEBRERO';
          WHEN 3 THEN SET v_mes_en_letra = 'MARZO';
          WHEN 4 THEN SET v_mes_en_letra = 'ABRIL';
          WHEN 5 THEN SET v_mes_en_letra = 'MAYO';
          WHEN 6 THEN SET v_mes_en_letra = 'JUNIO';
          WHEN 7 THEN SET v_mes_en_letra = 'JULIO';
          WHEN 8 THEN SET v_mes_en_letra = 'AGOSTO';
          WHEN 9 THEN SET v_mes_en_letra = 'SEPTIEMBRE';
          WHEN 10 THEN SET v_mes_en_letra = 'OCTUBRE';
          WHEN 11 THEN SET v_mes_en_letra = 'NOVIEMBRE';
          WHEN 12 THEN SET v_mes_en_letra = 'DICIEMBRE';
   END CASE;
   RETURN v_mes_en_letra;
END$$
DELIMITER ;
```

Ejemplos de uso:

```
SET @mes = mes_en_letra(10);
SELECT @mes;
SELECT DISTINCT MONTH(fecha) as mes_numero, mes_en_letra(MONTH(fecha)) as mes_cadena
FROM conferencia
WHERE idConferencia='ADS1314'
ORDER BY mes_numero;
SELECT * FROM conferencia
WHERE mes_en_letra(MONTH(fecha)) = 'OCTUBRE' COLLATE 'utf8mb4_spanish_ci'
ORDER BY tema, fecha;
```

Más información sobre CREATE FUNCTION en https://mariadb.com/kb/en/create-function/

TAREA

15. **Crear una función que dada una sala devuelva el número de veces que se celebró una conferencia en dicha sala. Llamar a dicha función por cada una de las salas.**
16. **Crear una función a la que se le pase el *idPonente* de un ponente y devuelva en cuantas conferencias participa. En caso de que no exista el ponente debe devolver -1. Llamar a la función y hacer que se muestre en cuantas conferencias participa cada uno de los ponentes. Guardar en una variable de sesión en cuantas conferencias participa el ponente con *idPonente* "*ESP001*" y mostrar su valor. Mostrar los ponentes que participan en 2 o más conferencias empleando la función.**

8 Gestión de errores. Excepciones.

8.1 Introducción.

Es **necesario capturar los errores** que se puedan producir **para intentar dar una respuesta mediante programación** al resultado de dicho error. La captura de los errores es **necesaria para que el programa o *script* que se ejecute pueda continuar**.

A la hora de programar, por medio de procedimientos/funciones, se pueden realizar varias **aproximaciones diferentes a la gestión de errores**:

- ✓ **Capturar las excepciones** que se puedan dar **y mandar un mensaje al programa cliente** (por medio de un parámetro de salida o un valor concreto devuelto con un SELECT). En este caso, el tratamiento del error se gestiona dentro del procedimiento/función y se programa la consecuencia que provoca dicho error.
- ✓ **No capturar excepciones** y que **sea el programa cliente el que gestione los mismos**. En este caso, incluso, puede ser uno mismo el que provoque una excepción dentro del código del procedimiento/función (por ejemplo, el cliente envía un *idPonente* de un ponente que no existe).
- ✓ **Combinación de las dos anteriores.**

Si se trabaja directamente contra el *SGBD MariaDB* sin hacer uso de un programa cliente, se llamará directamente a los procedimientos/funciones por lo que lo normal sería intentar capturar las excepciones para que la ejecución de los scripts no parase si se produjera algún error.

8.2 Declaración de excepciones (**DECLARE CONDITION**).

Se podrán capturar **excepciones**:

- ✓ Ya **definidas por el *SGBD*** (por ejemplo, DUPLICATE ENTRY FOR KEY, que se produce cuando se intentan añadir dos filas con la misma clave primaria).

```
DECLARE condition_name CONDITION FOR condition_value
condition_value:
    SQLSTATE [VALUE] sqlstate_value
  | mysql_error_code
```

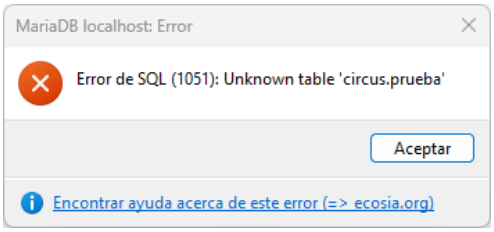
- ✓ Excepciones **personalizadas por el usuario**. Por ejemplo, se puede lanzar una excepción cuando se intente añadir una sala con una capacidad negativa.

En cualquiera de los dos casos, **cada excepción va a estar asociada a un número** ya predefinido por el *SGBD*.

Para hacer más legible el manejo de excepciones, **se puede asociar dicho número a un nombre de excepción** y después controlar la captura por dicho nombre y no por el número asociado.

Se puede consultar la **lista de códigos con el tipo de excepción asociado** en https://mariadb.com/kb/en/mariadb-error-codes/.

Por ejemplo, el código de error que aparece cuando se intenta borrar una tabla que no existe es el 1051. Ej.: `DROP TABLE prueba;`



Dentro de un procedimiento se va a poder capturar esta excepción, pero en vez de capturarla por su número (1051) se puede hacer por un nombre de la forma:

```
DECLARE no_existe_tabla CONDITION FOR 1051;
```

En la lista de mensajes de error aparecen dos **tipos de errores asociados a cada mensaje**:

- ✓ ***Error Code***: son números específicos de *MariaDB* que no valen para otros *SGBD*.
- ✓ ***SQLSTATE***: cadena de 5 dígitos basado en *ANSI SQL* y *ODBC*, y por tanto sus códigos son estandarizados.

Error Code	SQLSTATE	Error	Description
1000	HY000	ER_HASHCHK	hashchk
1001	HY000	ER_NISAMCHK	isamchk
1002	HY000	ER_NO	NO
1003	HY000	ER_YES	YES
1004	HY000	ER_CANT_CREATE_FILE	Can't create file '%s' (errno: %d)
1005	HY000	ER_CANT_CREATE_TABLE	Can't create table '%s' (errno: %d)
			Can't create database '%s'

Emplear uno u otro va a depender del programa cliente, **si va a ser empleado en *SGBD* diferentes sería mejor emplear los *SQLSTATE***. En **caso contrario, mejor los específicos de cada *SGBD***.

En el caso de querer **dar un nombre a la excepción empleando el *SQLSTATE*** la **sintaxis** cambia:

```
DECLARE no_existe_tabla CONDITION FOR SQLSTATE '42S02';
```

8.3 Lanzamiento de excepciones (`SIGNAL`).

`SIGNAL` es la **manera** que tiene *MariaDB* de que un **usuario pueda lanzar una excepción**.

Puede llevar como dato, un nombre de excepción definido como se ha visto antes con `DECLARE CONDITION` **o un código *SQLSTATE*** de 5 caracteres:

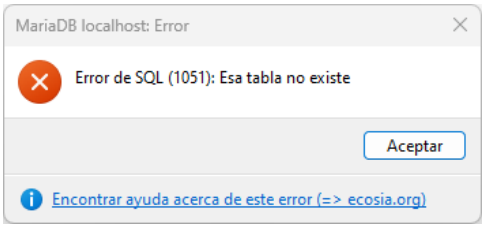
- ✓ Si **se emplea un nombre de condición**, esta **tiene que ser del tipo *SQLSTATE***, no se podrá usar un nombre de condición asociado a un *Error Code*.
- ✓ **Si se emplea el código de 5 caracteres, este no debe comenzar con '00'** (tanto si se emplea un código directamente como si se emplea un nombre de excepción basado en un código que empiece por 00) ya que esto **indica que es correcto y por tanto no lanzará la excepción**.

Existe un **código estándar para indicar que la excepción está definida por el usuario** (no es ninguna de las que ya existen): **45000**.

Ejemplos:

```
-- DECLARE no_existe_tabla CONDITION FOR 1051; /* No se puede usar si no es del tipo SQLSTATE */
DECLARE no_existe_tabla CONDITION FOR SQLSTATE '42S02';
SIGNAL no_existe_tabla;
```

Aparte de provocar una excepción, se pueden realizar diferentes acciones que se encuentran indicadas en la sintaxis de la sentencia. Por ejemplo, si se quiere provocar una excepción en la que se muestre el *SQLSTATE*, el número de error *MariaDB* y un texto asociado a la excepción se puede utilizar:




```
SIGNAL SQLSTATE '42S02'
       SET MESSAGE_TEXT = 'Esa tabla no existe',
           MYSQL_ERRNO = 1051;
```

Aclaraciones:

- ✓ **Cuando se lanza una excepción se sale del procedimiento**.
- ✓ El **uso que se haga de las excepciones va a depender del tipo de programa que se esté desarrollando**.
- ✓ Si se está desarrollando un **programa cliente**, el **control de las excepciones se hará en dicho programa, y se**

tendrá que programar una respuesta a la misma.

- Si, por ejemplo, se está desarrollando una aplicación en *Java*, se podrán capturar las excepciones y consultar en el objeto *Exception* los datos de la excepción, como su número, mensaje o *SQLSTATE*.

```
SIGNAL error_condition
    [SET error_property
    [, error_property] ...]
error_condition:
    SQLSTATE [VALUE] 'sqlstate_value'
  | condition_name
error_property:
    error_property_name = <error_property_value>
error_property_name:
    CLASS_ORIGIN
  | SUBCLASS_ORIGIN
  | MESSAGE_TEXT
  | MYSQL_ERRNO
  | CONSTRAINT_CATALOG
  | CONSTRAINT_SCHEMA
  | CONSTRAINT_NAME
  | CATALOG_NAME
  | SCHEMA_NAME
  | TABLE_NAME
  | COLUMN_NAME
  | CURSOR_NAME
```

Ejemplo: crear un procedimiento que dado el nombre de una sala la borre. Antes tendrá que borrar todas las tablas relacionadas (conferencia, participa y asiste). En el caso de que la sala tenga una capacidad menor a 200 no estará permitido borrarla y lanzará una excepción (recordar que el código de error para excepciones definidas por el usuario es el 45000) con el texto: "*No es posible dar de baja salas con capacidad de menos de 200*". En el caso de que la sala no exista, deberá lanzar una excepción con el texto "*Esa sala no existe*" y un *Error Code* 1643.

```
DELIMITER $$
CREATE OR REPLACE PROCEDURE borrar_sala(p_nombre_sala VARCHAR(50))
COMMENT 'Da de baja una sala siempre que su capacidad sea superior a 200'
bloque_proc:BEGIN
    DECLARE v_capacidad SMALLINT DEFAULT -1;
    SELECT capacidad INTO v_capacidad FROM sala WHERE nombreSala = p_nombre_sala;
    CASE
       WHEN v_capacidad = -1 THEN
           SIGNAL SQLSTATE '45000'
              SET MESSAGE_TEXT = 'Esa sala no existe',
                  MYSQL_ERRNO = 1643;
           -- LEAVE bloque_proc; --> No hace falta ya que SIGNAL hace salir del procedimiento
       WHEN v_capacidad < 200 THEN
            SIGNAL SQLSTATE '45000'
               SET MESSAGE_TEXT = 'No se puede dar de baja salas con menos de 200 de capacidad';
            -- LEAVE bloque_proc; --> No hace falta ya que SIGNAL hace salir del procedimiento
       ELSE
            BEGIN END;
    END CASE;
    DELETE FROM participa WHERE idConferencia IN (SELECT idConferencia
                                                  FROM conferencia WHERE nombreSala = p_nombre_sala);
    DELETE FROM asiste WHERE idConferencia IN (SELECT idConferencia
                                               FROM conferencia WHERE nombreSala = p_nombre_sala);
    DELETE FROM conferencia WHERE nombreSala = p_nombre_sala; -- Conferencias asociadas a la sala
    DELETE FROM sala WHERE nombreSala = p_nombre_sala;
END$$
DELIMITER ;
```

Ejemplos de uso:

```
CALL borrar_sala('no_existe'); -- Devuelve el código 1643
CALL borrar_sala('Afrodita'); -- No cumple que la capacidad sea superior a 200
CALL borrar_sala('Zeus'); -- La da de baja de todas las tablas necesarias
```

8.4 Captura de excepciones (DECLARE HANDLER).

Una excepción es el aviso de un error que se está produciendo durante la ejecución de un trozo de código.

Si no se quiere que se interrumpa la ejecución se deberá capturar el error para decidir qué hacer en ese caso. Para ello se debe **declarar un HANDLER** (manejador de excepciones) de la siguiente manera:

```
DECLARE handler_type HANDLER
    FOR condition_value [, condition_value] ...
    statement
handler_type:
    CONTINUE
  | EXIT
  | UNDO
condition_value:
    SQLSTATE [VALUE] sqlstate_value
  | condition_name
  | SQLWARNING
  | NOT FOUND
  | SQLEXCEPTION
  | mariadb_error_code
```

No soportado por *MariaDB*

```
DECLARE tipo_handler HANDLER FOR condición [, ...]
  [BEGIN]
        lista_instrucciones
  [END;]
```

donde **tipo_handler** puede tomar los valores:

- ✓ **CONTINUE**: la **ejecución** del programa **continua**.
- ✓ **EXIT**: la **ejecución termina**.

- ✓ **`UNDO`**: no implementado por *MariaDB*.

y ***condición*** toma un valor de:

- ✓ **`mariadb_error_code`** y **`SQLSTATE`** son valores que se encuentran en el mensaje de error que se quiere controlar (https://mariadb.com/kb/en/mariadb-error-codes/).
- ✓ **`condition_name`** es el nombre de condición previamente especificado con `DECLARE ... CONDITION`. Debe estar en el mismo programa almacenado.
- ✓ **`SQLWARNING`** es una forma abreviada de la clase de valores *SQLSTATE* que comienzan con '01'.
- ✓ **`SQLEXCEPTION`** es la abreviatura de la clase de valores *SQLSTATE* que no comienzan con '00', '01' o '02'.
- ✓ **`NOT FOUND`** es la abreviatura de la clase de valores *SQLSTATE* que comienzan con '02'. Esto es relevante solo para el contexto de los cursores y se usa para controlar lo que sucede cuando un cursor llega al final de un conjunto de datos.

Al mismo tiempo, **se debe indicar** una o un conjunto de **instrucciones** (`lista_instrucciones`) **que se quiere que se ejecuten cuando se produzca la excepción** (si son **varias** deben ir dentro de un **`BEGIN...END`**).

Basta con ejecutar el código y esperar a que aparezca el mensaje de error, y entonces apuntar dicho código para introducirlo en el manejador, o bien, **consultar en la ayuda de *MariaDB* el listado de todos los mensajes de error susceptibles de ser capturados y controlados** por los *handlers*.

El siguiente ejemplo capturaría el error de tabla desconocida **continuando con la ejecución** permitiendo que el resto de instrucciones puedan finalizar:

```
DECLARE CONTINUE HANDLER FOR 1051
BEGIN
    -- cuerpo handler
END;
```

El siguiente ejemplo capturaría ese mismo error, pero forzando la **finalización de la ejecución**:

```
DECLARE EXIT HANDLER FOR SQLSTATE '42S02'
BEGIN
    -- cuerpo handler
END;
```

Si se quiere **capturar una excepción por un nombre**:

```
DECLARE no_existe_tabla CONDITION FOR SQLSTATE '42S02';
DECLARE CONTINUE HANDLER FOR no_existe_tabla
BEGIN
   -- Instrucciones a ejecutar en caso de excepción
END;
```

Ejemplo: crear un procedimiento que añada una nueva sala. En caso de intentar dar de alta una sala que ya exista, capturar la excepción y hacer que el procedimiento devuelva -1. En caso de que el alta sea correcta, que devuelva 0. Trabajar con nombres de excepciones en vez de con los números asociados.

```
DELIMITER $$
CREATE OR REPLACE PROCEDURE anadir_sala(p_nombre_sala VARCHAR(50), p_capacidad SMALLINT)
   COMMENT 'Añade una nueva sala'
BEGIN
   DECLARE ex_clave_duplicada CONDITION FOR 1062;
   DECLARE EXIT HANDLER FOR ex_clave_duplicada
   BEGIN
      SELECT -1 AS error_code;
   END;
   INSERT INTO sala VALUES (p_nombre_sala, p_capacidad);
   SELECT 0 AS error_code;
END$$
DELIMITER ;
```

Ejemplos de uso:

```
CALL anadir_sala('Apolo', 200); -- Clave duplicada
CALL anadir_sala('Júpiter', 300); -- Todo correcto
```

TAREA

17. Crear un procedimiento al que se le pase el nombre de una sala y el *idConferencia* de una conferencia y asigne

la sala a la conferencia. Deberá comprobar que la sala y la conferencia existen. En caso de que no, deberá lanzar una excepción con el *Error Code* 1643 (*SQLSTATE*: *02000* - *ER_SIGNAL_NOT_FOUND*) y texto "*La sala no existe*" o "*La conferencia no existe*". En el caso de que la conferencia ya tenga esa sala asignada lanzar una excepción con el *Error Code* 1643 y texto "*Cambios no realizados. La sala ya se encuentra asignada a la conferencia*".

18. **Crear un procedimiento que añada un nuevo ponente. Se debe comprobar si el *idPonente* existe. Esta comprobación se hará capturando la excepción correspondiente. En caso de error, se mandará en un parámetro de salida el valor -1. Si todo está correcto se mandará el valor 0 y se añadirá el ponente.**

9 *Triggers* o disparadores.

9.1 ¿Qué es un *trigger*?

También se le conoce con el nombre de **disparador** y más concretamente **es un programa almacenado (*Stored Program* o *SP*), creado para ejecutarse automáticamente cuando ocurra un evento en una tabla o vista de la *BD*. Dichos eventos son generados por las sentencias INSERT, UPDATE y DELETE**. Hasta *MariaDB 10.2.3* sólo podía haber un *trigger* de cada INSERT, UPDATE y DELETE por tabla o vista y por cada opción de tiempo BEFORE o AFTER.

Un *trigger* **nunca se llama directamente, en cambio, cuando una aplicación o usuario intenta insertar, actualizar, o borrar una fila en una tabla, la acción definida en el disparador se ejecuta automáticamente** (se dispara).

Un *trigger* viene a ser como un procedimiento almacenado que se ejecuta automáticamente cuando sobre una tabla se realiza alguna operación que implique modificar sus datos (DELETE, INSERT, UPDATE). Por lo tanto, un ***trigger* va a estar asociado a una tabla y a un tipo de operación sobre la tabla**.

Las **ventajas de los *triggers*** son varias:

- ✓ Se podrán **validar todos aquellos valores que no pudieron ser validados mediante restricciones** (*constraints*), asegurando así la integridad de los datos.
- ✓ **Permitirán ejecutar reglas de negocio**.
- ✓ Utilizándolos en combinación con los eventos se **pueden realizar acciones sumamente complejas**.
- ✓ **Permitirán llevar un control de los cambios realizados en una tabla**. Para esto se debe utilizar una segunda tabla de *log*.
- ✓ El **mantenimiento de la aplicación se reduce**, los cambios a un disparador se reflejan automáticamente en todas las aplicaciones que utilizan la tabla sin la necesidad de recompilar o enlazar.

Las **desventajas de utilizar *triggers*** son:

- ✓ **Al ejecutarse de forma automática puede dificultar llevar un control sobre qué sentencias *SQL* fueron ejecutadas**.
- ✓ **Incrementan la sobrecarga del *SGBD***. Un mal uso de los *triggers* puede tornarse en respuestas lentas por parte del servidor.
- ✓ Hay que **definir con anticipación la tarea que realizará** el *trigger*.
- ✓ **Hay que programarlos para cada *SGBD***.
- ✓ Los *triggers* **no se desarrollan pensando en un sólo registro**, los mismos deben funcionar en conjunto con los datos ya que se disparan por operación y no por registro.
- ✓ Por funcionalidad, **solo se aplican a las sentencias de INSERT, UPDATE y DELETE**.
- ✓ **No se pueden utilizar en tablas temporales**.

Los *triggers* se invocan para ejecutar un conjunto de instrucciones que protegen, restringen, actualizan o preparan la información de las tablas cuando se manipula (inserta, modifica o elimina) la información. **Para crear *triggers* son necesarios los privilegios SUPER y TRIGGER**.

Básicamente **cuando se decide crear un *trigger* se necesita determinar**:

- ✓ Sobre qué **tabla** va a aplicarse el *trigger*.
- ✓ Sobre qué **operación** se va a aplicar (INSERT, DELETE, UPDATE).
- ✓ **Si se quiere que el *trigger* se ejecute antes** (BEFORE) **o después** (AFTER) **de que se realice la operación** sobre la tabla.

Si la sentencia *SQL* que va a provocar la ejecución del *trigger* afecta a múltiples filas (por ejemplo, un borrado de muchas filas), el ***trigger* se ejecutará una vez por cada fila afectada**.

Es **posible tener más de un *trigger* sobre la misma tabla y con el mismo evento**. En estos casos, la **ejecución** se hará **uno después de otro en el orden en que fueron creados**.

```
CREATE [OR REPLACE]
    [DEFINER = { user | CURRENT_USER | role | CURRENT_ROLE }]
    TRIGGER [IF NOT EXISTS] trigger_name trigger_time trigger_event
    ON tbl_name FOR EACH ROW
[{ FOLLOWS | PRECEDES } other_trigger_name ]
trigger_stmt;
```

9.2 Creación de *triggers*.

Sintaxis básica:
```
CREATE [OR REPLACE] [DEFINER = {usuario | CURRENT_USER}]
TRIGGER [IF NOT EXISTS] nombre_trigger {BEFORE | AFTER} {UPDATE | INSERT | DELETE}
   ON nombre_de_la_tabla
   FOR EACH ROW
   [BEGIN]
        <bloque_de_instrucciones>
   [END]
```

Explicación de las **partes de la definición**:

- ✓ **`DEFINER = {usuario | CURRENT_USER}`**: indica al *SGBD* qué **usuario tiene privilegios, para la invocación de los *triggers* cuando surjan los eventos *DML***. **Por defecto** toma el valor **`CURRENT_USER`** que hace referencia al usuario actual que está creando el *trigger*.
- ✓ **`nombre_trigger`**: indica el **nombre del *trigger***. Por defecto existe una **nomenclatura** práctica para nombrar un *trigger*, la cual da mejor legibilidad en la administración de la *BD*. Primero, escribir el **nombre de tabla**, seguido usar la **inicial del momento de ejecución** (`AFTER` o `BEFORE`) y finalmente especificar con la **inicial de la operación *DML*** (`UPDATE`, `INSERT` o `DELETE`) y. Por ejemplo: `sala_BI_TRIGGER`.
- ✓ **`BEFORE | AFTER`**: especifica si el ***trigger* se ejecuta antes o después de la ejecución de la sentencia *DML***.
- ✓ **`UPDATE | INSERT | DELETE`**: aquí se elige **que sentencia dispara la ejecución del *trigger***.
- ✓ **`ON nombre_de_la_tabla`**: en esta sección se establece el **nombre de la tabla asociada**.
- ✓ **`FOR EACH ROW`**: establece que **el *trigger* se ejecute por cada fila** en la tabla asociada.
- ✓ **`<bloque_de_instrucciones>`**: define el **bloque de instrucciones que el *trigger* ejecuta** al ser invocado.

Cuando se define un ***trigger***, este está **asociado al usuario que lo creó**. Dicho **usuario debe tener el permiso `TRIGGER`** otorgado con la orden `GRANT` **sobre la tabla**, lo que le da derecho a crear, borrar, mostrar y ejecutar un *trigger*.

Cuando **otro usuario realiza una operación *SQL* sobre una tabla a la que está asociada el *trigger***, el **usuario que creó el *trigger* debe conservar el permiso `TRIGGER`** para que se pueda ejecutar.

Si se quiere que otro usuario, además del usuario *root* tenga **permisos para crear *triggers*** o disparadores **en una tabla**, ejecutar el comando *SQL*:

```
GRANT TRIGGER ON nombre_bd.nombre_tabla TO nombre_usuario;
```

Para **dar permisos de creación de *triggers* a un usuario para todas las tablas** de una *BD* ejecutar el comando *SQL*:

```
GRANT TRIGGER ON nombre_bd.* TO nombre_usuario;
```

Más información sobre la sentencia `CREATE TRIGGER` *en* https://mariadb.com/kb/en/create-trigger/

9.2.1 Identificadores **NEW** y **OLD**.

Dentro del *trigger* se va a poder acceder a los valores de la fila, anterior y posterior a la ejecución de la sentencia *SQL*. Es decir, si se realiza una operación UPDATE sobre una columna, dicha columna tendrá un valor antes de la ejecución de la orden *SQL* y otro después de la ejecución, ya que se le está enviando un nuevo valor.

Si se necesita **relacionar el *trigger* con columnas específicas de una tabla** se deben usar los identificadores OLD y NEW. **OLD indica el valor antiguo de la columna** y **NEW el valor nuevo que pudiese tomar**. Por ejemplo: `OLD.nombreSala` o `NEW.nombreSala`.

Las palabras clave **OLD y NEW permiten acceder a columnas en los registros afectados por un disparador** (OLD y

NEW no son sensibles a mayúsculas):

- ✓ En un **disparador para INSERT solamente puede utilizarse `NEW.nom_col`** ya que no hay una versión anterior del registro.
- ✓ En un **disparador para DELETE sólo puede emplearse `OLD.nom_col`** porque no hay un nuevo registro.
- ✓ En un **disparador para UPDATE se puede emplear `OLD.nom_col` para referirse a las columnas de un registro antes de que sea actualizado y `NEW.nom_col` para referirse a las columnas del registro tras actualizarlo**.

Una **columna precedida por OLD es de sólo lectura**. Es **posible hacer referencia a ella, pero no modificarla**. Una **columna precedida por NEW puede ser referenciada** si se tiene el privilegio SELECT sobre ella. En un disparador BEFORE, **también es posible cambiar su valor con `SET NEW.nom_col = valor`** si se tiene el privilegio de UPDATE sobre ella. Esto significa que un disparador puede usarse para modificar los valores antes de que se inserten en un nuevo registro o se empleen para actualizar uno existente.

En un **disparador BEFORE, el valor de NEW para una columna AUTO_INCREMENT es 0**, no el número secuencial que se generará en forma automática cuando el registro sea realmente insertado.

9.3 Tipos de *triggers*.

Existen dos tipos de disparadores que **se clasifican según la cantidad de ejecuciones a realizar**:

- ✓ ***Row triggers*** (o disparadores de fila / **`FOR EACH ROW`**): son **aquellos que se ejecutaran *n*-veces si se llaman *n*-veces desde la tabla asociada** al *trigger*.
- ✓ ***Statement triggers*** (o disparadores de secuencia / **`FOR EACH STATEMENT`**): son **aquellos que sin importar la cantidad de veces que se cumpla con la condición, su ejecución es única**. No soportado por *MariaDB*.

Ejemplo: *trigger* que añade un registro a una tabla auxiliar *log_accesos* cada vez que se realiza una inserción en una de las tablas. Para ello se crea la tabla de *log* que utilizará el *trigger* con la sentencia:

```
CREATE TABLE log_accesos(
    codigo INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    usuario VARCHAR(100),
    fecha DATETIME
);
```

En la tabla auxiliar de auditoría se almacena el nombre del usuario de *MariaDB* y la fecha/hora en la que se haya realizado una inserción en una tabla de la *BD*. A continuación, se crea el *trigger* correspondiente para auditar la inserción de registros en la tabla *ponente*:

```
DELIMITER $$
CREATE TRIGGER ponente_BI_TRIGGER BEFORE INSERT ON ponente FOR EACH ROW
BEGIN
   INSERT INTO conferencias2.log_accesos (usuario, fecha) VALUES (CURRENT_USER(), NOW());
END$$
DELIMITER ;
```

Para **obtener el usuario actual** de *MariaBD* se usa la **función `USER()` o `CURRENT_USER()`** y para **obtener la fecha y hora actuales** se usa la **función `NOW()`**.

9.4 Información sobre los *triggers*.

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

La **sentencia *SQL* para ver los *triggers*** asociados a una *BD* o a una tabla es **`SHOW TRIGGERS`**:

- ✓ **Por defecto muestra los *triggers* de la *BD* activa**. Si se quieren mostrar los de otra se debe utilizar la opción `FROM nombre_BD` o `IN nombre_BD`. Ej.: `SHOW TRIGGERS FROM conferencias2;`
- ✓ Si se quieren **buscar los *triggers* asociados a una determinada tabla**, se puede **hacer uso de la cláusula *LIKE*** que busca por patrón nombres de tablas. Ej.: `SHOW TRIGGERS LIKE "pon%";`
- ✓ Si se quiere **buscar dentro de los resultados aquellas filas que cumplan alguna condición asociada al valor de alguna columna** se debe hacer uso la **cláusula WHERE**. Ej.: `SHOW TRIGGERS WHERE 'Definer' LIKE 'root%';`

El comando **`SHOW CREATE TRIGGER`** muestra la **sentencia que creó el *trigger*** dado. **Sintaxis**:

```
SHOW CREATE TRIGGER nombre_trigger;
SHOW TRIGGERS; -- Triggers que hay en la BD
```

Los *triggers* o disparadores se almacenan en la tabla *triggers* del catálogo del sistema *information_schema*, para verlos se puede ejecutar (similar a `SHOW TRIGGERS;`):

```
SELECT * FROM information_schema.triggers;
```

9.5 Eliminación de *triggers*.

Sintaxis: `DROP TRIGGER [IF EXISTS] nombre_trigger;`

Recordar que se puede añadir `IF EXISTS` para indica que, si el *trigger* existe entonces lo borre.

9.6 Modificación de *triggers*.

La modificación de *triggers* **no es posible**, es necesario **borrar y volver a crear el *trigger***.

9.7 Ejemplos de *triggers*.

Ejemplo: crear un *trigger* que no permita añadir una nueva conferencia si la sala es 'Apolo', el turno es 'T' y el precio es menor que 20. En caso de no cumplirse la condición se debe lanzar una excepción.

```
DELIMITER $$
CREATE OR REPLACE TRIGGER conferencia_check_BI BEFORE INSERT ON conferencia FOR EACH ROW
BEGIN
   IF (NEW.nombreSala = 'Apolo' AND NEW.turno = 'T' AND NEW.precio < 20) THEN
       SIGNAL SQLSTATE '45000' SET message_text='La conferencia no se puede dar de alta';
   END IF;
END$$
DELIMITER ;
```

Ejemplo para probar el *trigger* creado:

```
INSERT INTO conferencia VALUES ('PRO1314', 'Programación', 15, '2023-03-20', 'T', 'Apolo');
```

Ejemplo: crear un *trigger* para que cuando se añada una nueva conferencia, se haga que dicha conferencia sea asignada a la sala que menos conferencias tiene independientemente de la sala indicada en la sentencia *INSERT*.

```
DELIMITER $$
CREATE OR REPLACE TRIGGER anadir_conferencia_BI BEFORE INSERT ON conferencia FOR EACH ROW
BEGIN
    DECLARE v_nombre_sala VARCHAR(50);
    DECLARE v_temp INT DEFAULT 0;
     -- Buscar la sala con menos conferencias
    SELECT nombreSala, COUNT(idConferencia) AS num INTO v_nombre_sala, v_temp
    FROM sala LEFT JOIN conferencia USING (nombreSala)
    GROUP BY nombreSala
    ORDER BY num, nombreSala
    LIMIT 1;
    IF (v_nombre_sala IS NOT NULL) THEN
       SET NEW.nombreSala = v_nombre_sala;
    END IF;
END$$
DELIMITER ;
```

Ejemplo para probar el *trigger* creado:

```
INSERT INTO conferencia VALUES ('LMA1314', 'Lenguajes de marcas', 15, '2023-03-20', 'T', 'Afrodita');
```

Ejemplo: agregar a la tabla ponente un campo *ganancias* de tipo `DECIMA(10,2)` y actualizar su valor con los datos actuales de participación que se encuentran en la *BD*. Seguidamente crear los *triggers* necesarios para que el campo *ganancias* de la tabla *ponente* se actualice cuando se añadan, borren o modifiquen datos en la tabla *participa*.

```
USE conferencias2;
ALTER TABLE ponente DROP COLUMN IF EXISTS ganancias;
ALTER TABLE ponente ADD COLUMN ganancias DECIMAL(10,2) UNSIGNED DEFAULT 0;
UPDATE ponente SET ganancias = (SELECT SUM(gratificacion)
                                FROM participa
                                WHERE participa.idPonente = ponente.idPonente);
DELIMITER $$
```

```
CREATE OR REPLACE TRIGGER ponente_actualizar_ganancias_AI AFTER INSERT ON participa FOR EACH ROW
BEGIN
    UPDATE ponente
    SET ganancias = NVL(ganancias, 0) + NEW.gratificacion
    WHERE idPonente = NEW.idPonente;
END$$
CREATE OR REPLACE TRIGGER ponente_actualizar_ganancias_AU AFTER UPDATE ON participa FOR EACH ROW
BEGIN
    UPDATE ponente
    SET ganancias = ganancias + NEW.gratificacion - OLD.gratificacion
    WHERE idPonente = OLD.idPonente;
END$$
CREATE OR REPLACE TRIGGER ponente_actualizar_ganancias_AD AFTER DELETE ON participa FOR EACH ROW
BEGIN
    UPDATE ponente
    SET ganancias = ganancias - OLD.gratificacion
    WHERE idPonente = OLD.idPonente;
END$$
DELIMITER ;
```

Ejemplos para probar los diferentes *triggers* creados:

```
INSERT INTO participa VALUES ('ESP001', 'ADS1314', 6, 800.00), ('ESP002', 'PWB1314', 5, 1000.00);
UPDATE participa SET gratificacion = 350.00 WHERE idPonente = 'ESP002' AND idConferencia = 'PWB1314';
DELETE FROM participa WHERE idPonente = 'ESP002' AND idConferencia = 'PWB1314';
```

TAREA

19. **Modificar la tabla "*sala*" y añadir una nueva columna de nombre contador, de tipo numérico y valor por defecto cero, que lleve la cuenta de cuantas veces se ha celebrado una conferencia en ella y que se actualice con cualquier operación (inserción, borrado o modificación). Crear los *triggers* necesarios. Nota: los datos ya añadidos tendrán como valor NULL. Realizar una operación de UPDATE para ponerlos a su valor correcto.**
20. **Crear los *triggers* necesarios que impidan que se pueda añadir o modificar una sala con una capacidad superior a 500 o inferior a 50 (esto se podría implementar con un CHECK, pero se va a practicar el uso de *triggers*).**
21. **Crear un *trigger* que si se intenta dar de alta una nueva conferencia y se envía un valor de *nombreSala* que no exista, se cambie su valor por el nombre de la sala en la que menos conferencias se celebran.**
22. **Crear una tabla de nombre "registro" con los campos:**
    - ***id*: INT, sin signo, autonumérico y clave primaria.**
    - ***usuario*: VARCHAR(30).**
    - ***tabla*: VARCHAR(50).**
    - ***operacion*: VARCHAR(10).**
    - ***datos_antiguos*: VARCHAR(70) (guardarán los datos nombreSala:capacidad que se borren o modifiquen).**
    - ***datos_nuevos*: VARCHAR(70) (guardarán los datos nombreSala:capacidad que se añadan o modifiquen).**
    - ***fecha_hora*: DATETIME.**

    Hacer que se registren todas las operaciones de alta, baja y modificación que se realicen sobre la tabla "*sala*".
23. **Crear una tabla de nombre "*contador*" con los campos:**
    - ***id*: INT, sin signo, autonumérico y clave primaria.**
    - ***tipo*: VARCHAR(100) y no nulo.**
    - ***valor*: INT, sin signo y no nulo.**

    Añadir dos filas con los valores para tipo/valor: salas/0 y conferencias/0.
    Hacer que cada vez que haya alguna operación que modifique (alta/baja) el número de salas o de conferencias, se actualice el número total de las mismas.
    Ejecutar la sentencia UPDATE necesaria que actualice la tabla contador con los datos actuales de las tablas.
24. **Mostrar los *triggers* creados sobre la tabla "*sala*". Borrar alguno de ellos.**

10 Cursores (CURSOR).

Un cursor es un **recurso de programación**, que ofrecen casi todos los *SGBD* relacionales, que **da la posibilidad de**

recorrer fila a fila un conjunto de resultados provenientes de una sentencia SELECT.

La **ventaja** que tiene el uso de cursores es que **se va a poder guardar en variables locales cada fila de resultados y manejar esa información como se quiera** (para realizar otras consultas, operaciones de INSERT, UPDATE o DELETE).

Recordar que en algunos ejercicios se ha tenido que limitar el resultado de SELECT a una única fila, ya que el INTO no podría hacerse. **Con un cursor, la consulta va a poder devolver muchas filas**, ya que permite obtener los datos de cada fila e ir pasando de una fila a la siguiente.

Un cursor tiene las siguientes **características**:

- ✓ Son **de sólo lectura**: sólo sirven para leer datos. Es decir, sólo se podrá declarar un cursor para leer los datos que provienen de una consulta SELECT y nunca se podrá modificar los datos de la tabla a través del cursor.
- ✓ De **acceso secuencial**: la información que va a procesar el cursor (el resultado de un SELECT) es secuencial. Se va a recorrer fila a fila desde la primera a la última de forma secuencial, una detrás de otra y no se puede saltar a una fila cualquiera de forma directa, se tendrá que pasar por todas las anteriores.
- ✓ Puede **crearse dentro** de **procedimientos, funciones, *triggers* o eventos**.

10.1 Declaración de cursores.

La declaración del cursor **permite definir el nombre del cursor y la consulta que va a devolver el conjunto de resultados**. La **sintaxis** a utilizar es:

```
DECLARE cursor_name CURSOR [(cursor_formal_parameter[,...])] FOR select_statement
cursor_formal_parameter:
    [IN] name type [collate clause]
```

```
DECLARE nombre_cursor CURSOR FOR sentencia_select;
```

- ✓ La sentencia **SELECT no puede contener INTO** y se seleccionarán las **columnas que se quieran guardar 'por cada fila'** al recorrer el cursor**.**
- ✓ Se deben **declarar después de la declaración de variables**.
- ✓ Dentro de un procedimiento, función, *trigger* o evento **se pueden tener varios cursores, pero con nombres diferentes**.
- ✓ Para salir del bucle se va a **capturar una excepción de 'no hay más datos'**. La definición de dicha captura tiene que ir **después de DECLARE**.

Ejemplo:
```
DECLARE v_no_hay_mas_datos INT DEFAULT 0;
DECLARE c_ponentes CURSOR FOR SELECT nombre, apellido1 FROM ponente WHERE apellido2 IS NULL;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_no_hay_mas_datos = 1;
```

En el ejemplo, en la primera línea se declara una variable de tipo INT que se utiliza para saber cuándo se acaba de recorrer las filas del cursor. Recordar que la **definición de variables tiene que ir antes de la definición del cursor**.

En la segunda línea, **se declara el cursor**. Y en la tercera línea **se captura la excepción que se produce cuando el cursor no encuentra más datos**. Lo que indica dicha línea es, "cuando se produce la excepción, asigna a la variable v_no_hay_mas_datos el valor TRUE".

10.2 Apertura del cursor.

```
OPEN cursor_name [expression[,...]];
```

Para **abrir un cursor** que se haya definido se utiliza: **`OPEN nombre_cursor;`**

Ejemplo:
```
DECLARE v_no_hay_mas_datos INT DEFAULT 0;
DECLARE c_ponentes CURSOR FOR SELECT nombre, apellido1 FROM ponente WHERE apellido2 IS NULL;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_no_hay_mas_datos = 1;
OPEN c_ponentes;
```

10.3 Lectura del cursor.

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

Esto se realiza con la **orden FETCH** que **permite guardar la información de una fila en variables locales y pasa a la siguiente fila del conjunto de resultados**. La lectura va en un bucle que se ejecuta hasta que se terminan de leer todas las filas. **Sintaxis**:

```
FETCH nombre_cursor INTO nombre_variable1 [, nombre_variable1, ...];
```

- ✓ En la parte **INTO** deben ir **tantas variables como columnas se tengan en la sentencia SELECT**.

- ✓ **Cada variable guardará el dato de la columna que venga en el SELECT**, de tal forma que la primera variable guardará el dato de la primera columna, la segunda variable, el dato de la segunda columna y así sucesivamente.
- ✓ **Cada variable debe estar definida con el mismo tipo de dato que la columna a la que está asociada**.

Esta es la parte que **va a necesitar de un bucle** (con cualquiera de las formas ya vistas), en el que se va 'fila a fila' recorriendo el resultado de ejecutar la sentencia SELECT que define el cursor.

El **paso de una fila a la siguiente** se realiza con la sentencia **FETCH**.

Ejemplo:

```
DECLARE v_no_hay_mas_datos INT DEFAULT 0;
DECLARE v_nombre, v_apellido1 VARCHAR(50);
DECLARE c_ponentes CURSOR FOR SELECT nombre, apellido1 FROM ponente WHERE apellido2 IS NULL;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_no_hay_mas_datos = 1;
OPEN c_artistas;
read_loop: LOOP
    FETCH c_ponentes INTO v_nombre, v_apellido1;
    IF v_no_hay_mas_datos = 1 THEN
      LEAVE read_loop;
    END IF;
    -- Por cada fila se dispone del nombre y apellido1. Ahora se pueden utilizar estos
    -- datos para lo que se quiera.
END LOOP;
```

Notas al ejemplo:

- ➢ Al estar dentro de un **bucle se debe asegurar que la condición de salida siempre se alcanza** ya que si no se estaría en un bucle infinito.
- ➢ Se debe **asegurar** que el **FETCH está dentro del bucle ya que es la orden que va a provocar la excepción** y va a permitir salir del bucle.

10.4 Cierre del cursor.

```
CLOSE cursor_name
```

El cierre del cursor **libera la memoria que está utilizando** el cursor creado. Para **cerrar el cursor**:

```
CLOSE nombre_cursor;
```

Ejemplo:

```
DECLARE v_no_hay_mas_datos INT DEFAULT 0;
DECLARE v_nombre, v_apellido1 VARCHAR(50);
DECLARE c_ponentes CURSOR FOR SELECT nombre, apellido1 FROM ponente WHERE apellido2 IS NULL;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_no_hay_mas_datos = 1;
OPEN c_artistas;
read_loop: LOOP
    FETCH c_ponentes INTO v_nombre, v_apellido1;
    IF v_no_hay_mas_datos = 1 THEN
      LEAVE read_loop;
    END IF;
    -- Por cada fila se dispone del nombre y apellido1. Ahora se pueden utilizar estos
    -- datos para lo que se quiera.
END LOOP;
CLOSE c_ponentes;
```

10.5 Ejemplos de uso de cursores.

Ejemplo: siguiendo con el ejemplo del punto anterior, crear un procedimiento que compruebe si las ganancias totales de cada ponente coinciden con la suma de las gratificaciones recibidas. El procedimiento debe devolver una cadena con el formato: idPonente1:gananciatotal:gananciasumada, idPonente2:gananciatotal:gananciasumada con los ponentes que no cumplen que la suma sea igual.

```
DELIMITER $$
CREATE OR REPLACE PROCEDURE ponentes_check_ganancias()
COMMENT 'Devuelve los ponentes cuyas ganancias no coincide con la suma de las gratificaciones.'
BEGIN
   DECLARE v_id_ponente CHAR(6);
   DECLARE v_gan_totales, v_gan_totales_calculada DECIMAL(10, 2);
```

```
        DECLARE v_cadena_salida VARCHAR(1000) DEFAULT '';
        DECLARE v_final INT DEFAULT 0;
        DECLARE c_check_ganancias CURSOR FOR SELECT idPonente, ganancias FROM ponente;
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_final = 1;
        OPEN c_check_ganancias;
        read_loop: LOOP
             FETCH c_check_ganancias INTO v_id_ponente, v_gan_totales;
             IF v_final = 1 THEN
                 LEAVE read_loop;
             END IF;
             SELECT SUM(gratificacion) INTO v_gan_totales_calculada
             FROM participa
             WHERE idPonente = v_id_ponente;
             IF (v_gan_totales_calculada <> v_gan_totales) THEN
                 SET v_cadena_salida = IF(v_cadena_salida<>'', CONCAT(v_cadena_salida, ', '), '');
                 SET v_cadena_salida = CONCAT(v_cadena_salida, v_id_ponente, ':', v_gan_totales, ':',
                                              v_gan_totales_calculada);
             END IF;
         END LOOP;
         CLOSE c_check_ganancias;
         SELECT v_cadena_salida AS lista_ponentes;
    END$$
    DELIMITER ;
```

Ejemplo de llamada al procedimiento:

```
        CALL ponentes_check_ganancias();   -- Modificar el campo ganancias en algún ponente
                                           -- para que salga algún ponente en la lista
```

Ejemplo: crear un procedimiento que compruebe en cuantas conferencias participa cada ponente. Aquellos ponentes que participen en más de un número de conferencias indicado por un parámetro se le dará un plus igual al número conferencias en las que participa multiplicado por 150. Mostrar el nombre completo y complemento de cada ponente.

```
DELIMITER $$
CREATE OR REPLACE PROCEDURE ponente_add_suplemento(p_num_conferencias TINYINT)
COMMENT 'Muestra un suplemento para aquellos ponentes que participan en más conferencias de las indicadas como
parámetro'
BEGIN
    DECLARE v_id_ponente CHAR(6);
    DECLARE v_num_conferencias TINYINT DEFAULT 0;
    DECLARE v_nombre, v_apellido1, v_apellido2 VARCHAR(50);
    DECLARE v_final INT DEFAULT 0;
    DECLARE c_complemento CURSOR FOR SELECT idPonente, COUNT(*)
                                     FROM participa
                                     GROUP BY idPonente HAVING COUNT(*) > p_num_conferencias;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_Final = 1;
    CREATE TEMPORARY TABLE t_temporal(nombre_completo VARCHAR(153), suplemento DECIMAL(10,2));
    OPEN c_complemento;
    read_loop: LOOP
        FETCH c_complemento INTO v_id_ponente, v_num_conferencias;
        IF v_final = 1 THEN
            LEAVE read_loop;
        END IF;
        IF (v_num_conferencias > p_num_conferencias) THEN
            SELECT nombre, apellido1, apellido2 INTO v_nombre, v_apellido1, v_apellido2
            FROM ponente
            WHERE idPonente = v_id_ponente;
            INSERT INTO t_temporal VALUES (CONCAT(v_apellido1, ' ', NVL(v_apellido2, ''), ', ',
                                                  v_nombre), v_num_conferencias*150);
        END IF;
    END LOOP;
    SELECT nombre_completo, suplemento FROM t_temporal;
    CLOSE c_complemento;
    DROP TEMPORARY TABLE t_temporal;
END$$
DELIMITER ;
```

Ejemplo de llamada al procedimiento:

```
            CALL ponente_add_suplemento(2);
```

TAREA

25. Crear un procedimiento que muestre los asistentes junto al total gastado en conferencias, en caso de que no haya asistido a ninguna conferencia deberá mostrar la cadena "*Sin gasto*". Para realizar el cálculo del total gastado utilizar un cursor que recorra todos los asistentes y por cada uno de ellos vaya calculando su gasto.

11 Eventos.

El uso de eventos **permite realizar tareas programadas.** Son **tareas que se ejecutan en base a un horario definido** previamente. Se identifican por su nombre. Cada evento **está asociado a una *BD* y se ejecuta en un intervalo de tiempo determinado**. Se pueden programar para que **se ejecuten una única vez o periódicamente**.

Para **iniciar el uso de los eventos** antes se debe **activar el planificador de *MariaDB***, que es quien se encarga de buscar en segundo plano eventos a ejecutar: `SET GLOBAL EVENT_SHEDULER = ON;` Y del mismo modo para desactivar el planificador: `SET GLOBAL EVENT_SHEDULER = OFF;`

```
CREATE [OR REPLACE]
    [DEFINER = { user | CURRENT_USER | role | CURRENT_ROLE }]
    EVENT
    [IF NOT EXISTS]
    event_name
    ON SCHEDULE schedule
    [ON COMPLETION [NOT] PRESERVE]
    [ENABLE | DISABLE | DISABLE ON SLAVE]
    [COMMENT 'comment']
    DO sql_statement;
schedule:
    AT timestamp [+ INTERVAL interval] ...
  | EVERY interval
    [STARTS timestamp [+ INTERVAL interval] ...]
    [ENDS timestamp [+ INTERVAL interval] ...]
interval:
    quantity {YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE |
              WEEK | SECOND | YEAR_MONTH | DAY_HOUR | DAY_MINUTE |
              DAY_SECOND | HOUR_MINUTE | HOUR_SECOND | MINUTE_SECOND}
```

11.1 Creación de eventos.

Sintaxis:

```
CREATE [OR REPLACE] [DEFINER = {USER | CURRENT_USER}] EVENT [IF NOT EXISTS] nombre_evento
  ON SCHEDULE calendario
  [ON COMPLETION [NOT] PRESERVE]
  [ENABLE | DISABLE | DISABLE ON SLAVE]
  [COMMENT 'comentario']
  DO EVENT_BODY;
```

calendario:

```
AT TIMESTAMP [+ INTERVAL intervalo] ...
| EVERY intervalo
  [STARTS TIMESTAMP [+ INTERVAL intervalo] ...]
  [ENDS TIMESTAMP [+ INTERVAL intervalo] ...]
```

intervalo:

```
cantidad {YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE |
          WEEK | SECOND | YEAR_MONTH | DAY_HOUR | DAY_MINUTE |
          DAY_SECOND | HOUR_MINUTE | HOUR_SECOND | MINUTE_SECOND}
```

Ejemplo: evento que aumenta el valor de un campo de una tabla cada hora.

```
CREATE EVENT nombre_evento
ON SCHEDULE AT CURRENT_TIMESTAMP+INTERVAL 1 HOUR
DO UPDATE db.tabla SET campo=campo+1;
```

Ejemplo: similar al anterior en el que se usan las cláusulas STARTS y ENDS para indicar cuándo debe empezar a ejecutar el evento y cuanto debe acabar.

```
DELIMITER $$
CREATE EVENT nombre_evento
ON SCHEDULE EVERY 1 HOUR
STARTS CURRENT_TIMESTAMP + INTERVAL 1 DAY
ENDS CURRENT_TIMESTAMP + INTERVAL 1 YEAR
DO BEGIN
       UPDATE db.tabla SET campo = campo + 1;
END$$
DELIMITER ;
```

11.2 Borrado de eventos.

```
DROP EVENT [IF EXISTS] event_name
```

La **sentencia *SQL*** que modifica un evento es **DROP EVENT**. Se necesita el **privilegio EVENT sobre la *BD***. Sintaxis:

```
DROP EVENT [IF EXISTS] nombre_evento;
```

11.3 Modificación de eventos.

La **sentencia *SQL*** que modifica un evento es **ALTER EVENT**. Se necesita el **privilegio EVENT sobre la *BD***.

Modificar el nombre de un evento:

```
ALTER EVENT evento RENAME TO nuevo_nombre_evento;
```

Detener un evento (deshabilitar) o activarlo nuevamente:

```
ALTER EVENT nombre_evento [DISABLE|ENABLE];
```

```
ALTER
    [DEFINER = { user | CURRENT_USER }]
    EVENT event_name
    [ON SCHEDULE schedule]
    [ON COMPLETION [NOT] PRESERVE]
    [RENAME TO new_event_name]
    [ENABLE | DISABLE | DISABLE ON SLAVE]
    [COMMENT 'comment']
    [DO sql_statement]
```

Ejemplo: sobre el evento creado en el punto anterior, hacer una modificación para que comience en el mes actual.

```
ALTER EVENT nombre_evento ON SCHEDULE EVERY 1 MONTH STARTS NOW();
```

11.4 Consulta de eventos.

La **sentencia *SQL*** que permite obtener información de un evento es **SHOW EVENTS**. Se necesita el **privilegio EVENT sobre la *BD***.

Para **ver los eventos asociados a una *BD* específica** se puede hacer uso de la palabra FROM de la forma:

```
SHOW EVENTS FROM nombre_bd;
```

Si se quiere buscar dentro de los resultados aquellas filas que cumplan alguna condición asociada al valor de alguna columna se debe hacer uso la cláusula WHERE. Por ejemplo: `SHOW EVENTS WHERE Definer LIKE 'root%';`

11.5 Ejemplos de uso de eventos.

Ejemplo: crear un evento que se ejecute cada minuto y compruebe si las ganancias totales de cada ponente coinciden con la suma de las gratificaciones recibidas por las conferencias en las que participa. En caso de no coincidir se debe insertar una fila en una tabla control que debe tener los siguientes campos: datos ponentes no coincide, fecha y hora del chequeo.

```
USE conferencias2;
DROP TABLE IF EXISTS control;
CREATE TABLE IF NOT EXISTS control (
       resultado VARCHAR(1000) NOT NULL DEFAULT '',
       momento DATETIME NOT NULL DEFAULT CURRENT_DATE
);
SET GLOBAL EVENT_SCHEDULER = OFF;

DELIMITER $$
CREATE OR REPLACE PROCEDURE check_ganancias()
COMMENT 'Comprueba si las ganancias de los ponentes coinciden con las gratificaciones'
BEGIN
   DECLARE v_id_ponente CHAR(6);
       DECLARE v_gan_totales, v_gan_totales_calculada DECIMAL(10, 2);
       DECLARE v_cadena_salida VARCHAR(1000) DEFAULT '';
       DECLARE v_final INT DEFAULT 0;
   DECLARE c_check_ganancias CURSOR FOR SELECT idPonente, ganancias FROM ponente;
       DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_final = 1;
       OPEN c_check_ganancias;
       read_loop: LOOP
        FETCH c_check_ganancias INTO v_id_ponente, v_gan_totales;
        IF v_final = 1 THEN
            LEAVE read_loop;
        END IF;
        SELECT SUM(gratificacion) INTO v_gan_totales_calculada
        FROM participa
        WHERE idPonente = v_id_ponente;

        IF (v_gan_totales_calculada <> v_gan_totales) THEN
            SET v_cadena_salida = IF(v_cadena_salida<>'', CONCAT(v_cadena_salida, ', '), '');
            SET v_cadena_salida = CONCAT(v_cadena_salida, v_id_ponente, ':', v_gan_totales, ':',
                                         v_gan_totales_calculada);
        END IF;
    END LOOP;
    CLOSE c_check_ganancias;
```

```
        IF v_cadena_salida = '' THEN
             INSERT INTO control VALUES('TODO COINCIDE', NOW());
        ELSE
             INSERT INTO control VALUES(v_cadena_salida, NOW());
        END IF;
END$$
DELIMITER ;
```

```
SET GLOBAL EVENT_SCHEDULER = ON;
CREATE OR REPLACE EVENT valida_ganancias_ponentes
ON SCHEDULE EVERY 1 MINUTE
STARTS NOW()
COMMENT 'Registrar coincidencia de ganancias con gratificaciones de los ponentes'
DO
       CALL check_ganancias();

SHOW EVENTS;
```

Recordar que en ejemplos anteriores se agregó y actualizó el campo ganancias en ponente. Si no se tiene esto realizado se debe agregar este campo y actualizarlo ejecutando:

```
ALTER TABLE ponente DROP COLUMN IF EXISTS ganancias;
ALTER TABLE ponente ADD COLUMN ganancias DECIMAL(10,2) UNSIGNED DEFAULT 0;
UPDATE ponente SET ganancias = (SELECT SUM(gratificacion)
                                FROM participa
                                WHERE participa.idPonente = ponente.idPonente);
```

Pasados unos minutos ejecutar la siguiente sentencia:

```
UPDATE ponente
SET ganancias = 5000
WHERE idPonente = 'ESP001';
```

Dejar pasar unos minutos y comprobar el contenido de la tabla control.

conferencias2.control: 5 filas en total (aproximadamente)

resultado	momento
TODO COINCIDE	2023-03-29 22:59:23
TODO COINCIDE	2023-03-29 23:00:23
TODO COINCIDE	2023-03-29 23:01:23
TODO COINCIDE	2023-03-29 23:02:23
ESP001:5000.00:450.00	2023-03-29 23:03:23

Ejemplo: crear un evento que cada mes realice una copia de los datos contenidos en cada una de las tablas de la *BD "conferencias2"*. La copia se realizará en unas nuevas tablas que se llamarán de la misma forma, pero añadiéndole el prefijo "*c_*".

```
USE conferencias2;
DELIMITER $$
CREATE OR REPLACE EVENT copia_tablas
ON SCHEDULE EVERY 1 MONTH
STARTS NOW()
DO BEGIN
   CREATE OR REPLACE TABLE c_sala AS SELECT * FROM sala;
   CREATE OR REPLACE TABLE c_asistente AS SELECT * FROM asistente;
   CREATE OR REPLACE TABLE c_ponente AS SELECT * FROM ponente;
   CREATE OR REPLACE TABLE c_conferencia AS SELECT * FROM conferencia;
   CREATE OR REPLACE TABLE c_asiste AS SELECT * FROM asiste;
   CREATE OR REPLACE TABLE c_participa AS SELECT * FROM participa;
END$$
DELIMITER ;
```

```
SET GLOBAL EVENT_SCHEDULER = ON;
```

Para comprobar el funcionamiento del evento programado cada menos tiempo, modificarlo y hacer que se ejecute cada minuto (modificar y eliminar datos en las tablas para comprobarlo):

```
ALTER EVENT copia_tablas ON SCHEDULE EVERY 1 MINUTE STARTS NOW();
```

TAREA

26. Crear un evento que registre cada minuto los usuarios que están accediendo a la *BD "conferencias2"*. Los usuarios y la información relativa a ellos deberá quedar registrada en una tabla *log_accesos* que se debe crear con los campos que se consideren necesarios. Para probar el funcionamiento del evento conectarse con dos usuarios distintos desde *HeidiSQL* y la terminal (mantenerse conectados varios minutos).

Column	Explanation
ID	Unique identifier
USER	User name (ie: root, techonthenet, etc)
HOST	Host for the user
DB	Database that thread is running in
COMMAND	Command that is being run (ie: Query, Sleep, etc).
TIME	Number of seconds that thread has been running (ie: 3, 12, 353)
STATE	State of thread (ie: executing)
INFO	Displays information about the thread. (ie: if COMMAND='Query' and STATE='executing', the SQL that the user is running will be displayed)

Para obtener la información de los usuarios conectados a la *BD "conferencias2"* utilizar la sentencia siguiente:

```
SELECT *
FROM information_schema.processlist
WHERE DB = "conferencias2";
```

12 Transacciones en procedimientos y eventos.

Si el **procedimiento/evento realiza varias operaciones lo normal es que lleve asociado una transacción y un control de errores capturando las posibles excepciones**. Se debe **utilizar START TRANSACTION para iniciar la transacción**, **no se puede utilizar BEGIN [WORK]** ya que se trataría como un bloque BEGIN - END.

Se puede utilizar el manejo de errores para decidir si se hace ROLLBACK de una transacción. El **código de un procedimiento que capture los errores que se produzcan de tipo SQLEXCEPTION o SQLWARNING** y deshaga la transacción en caso de producirse algún error podría quedar de la siguiente forma:

```
DELIMITER $$
CREATE OR REPLACE PROCEDURE mi_procedimiento_con_transacciones()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING
    BEGIN
        SELECT -1; -- Informar a quien llamó de que algo ha fallado (error o warning)
        ROLLBACK;
    END;
    START TRANSACTION; -- No se puede utilizar BEGIN [WORK], se trata como bloque BEGIN - END
    -- Sentencias SQL a ejecutar dentro de la transacción
    COMMIT;
    SELECT 0; -- Informar a quien llamó de que no se han producido fallos
END$$
DELIMITER ;
```

Ejemplo: crear un procedimiento que inserte una nueva sala y la conferencia que se va a tener lugar en ella. Caso de producirse un error al dar de alta la sala o bien la conferencia se debe deshacer la transacción.

```
DELIMITER $$
CREATE PROCEDURE alta_sala_conferencia(p_id_conferencia CHAR(7), p_tema VARCHAR(60), p_precio
  DECIMAL(5,2), p_fecha DATE, p_turno CHAR(1), p_nombre_sala VARCHAR(50), p_capacidad SMALLINT)
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION -- , SQLWARNING
    BEGIN
        SELECT -1;        -- Se informa a quien llamó de que algo fue mal
        ROLLBACK;         -- Se cancela la transacción en caso de ERROR
    END;
    START TRANSACTION;    -- INICIO DE LA TRANSACCIÓN
    INSERT INTO sala VALUES (p_nombre_sala, p_capacidad);
    INSERT INTO conferencia VALUES (p_id_conferencia, p_tema, p_precio, p_fecha, p_turno, p_nombre_sala);
    COMMIT;   -- Confirma la transacción
    SELECT 0; -- Se informa a quien llamó de que no se producen errores
END$$
DELIMITER ;
```

Ejemplos de llamadas:

```
-- No produce error y se da de alta la sala y la conferencia
CALL alta_sala_conferencia('END2223', 'Entornos de Desarrollo', 15.0, CURDATE(), 'M', 'Galeón', 350);
-- Genera un error y no se da de alta ni la sala ni la conferencia (clave repetida al dar
-- de alta la conferencia END2223). La inserción de la sala 'Granada' se cancela.
CALL alta_sala_conferencia('END2223', 'Entornos de Desarrollo II', 15.0, CURDATE(), 'M', 'Granada', 450);
```

TAREA

27. Crear un procedimiento que inserte un nuevo asistente junto a la conferencia en la que se inscribe. Añadir una transacción para que en el caso de que se produzca algún error no se realice ninguna modificación en la *BD*. Poner varios ejemplos de llamada al procedimiento en los que se confirme la transacción y se deshaga por algún error que se pueda llegar a producir.