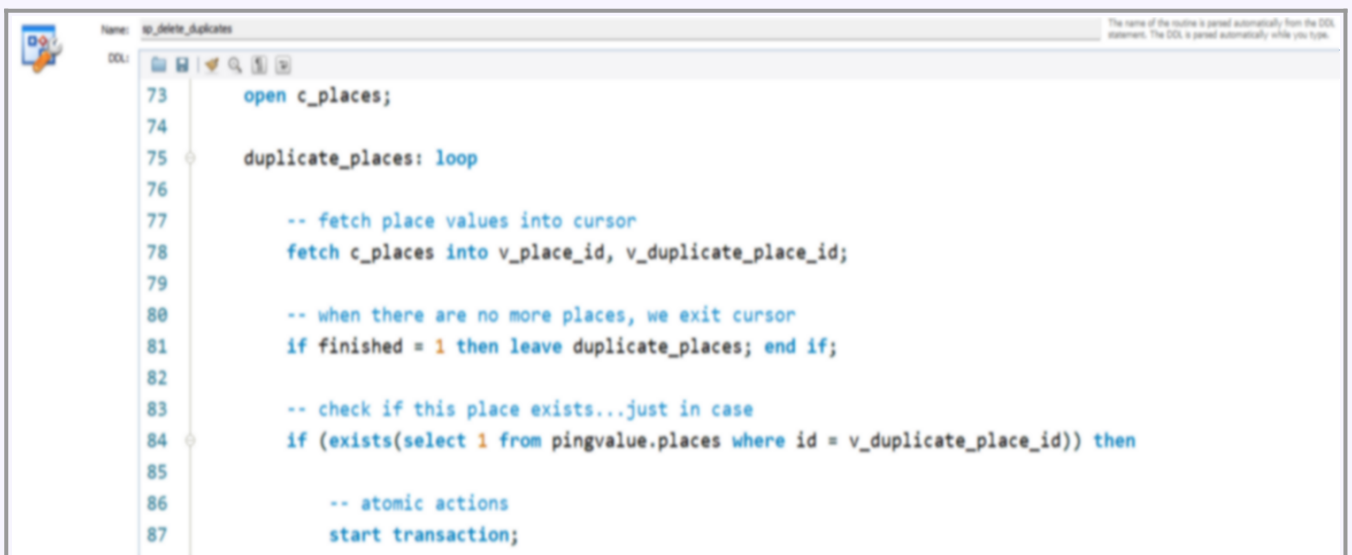


Programación SQL

Unidad 7. Módulo Bases de Datos. CFGS DAW

IES Torrevigía - Curso 2024-25.

Jesús Mañogil (j.manogilferrandez@edu.gva.es)



```
73  open c_places;
74
75  duplicate_places: loop
76
77      -- fetch place values into cursor
78      fetch c_places into v_place_id, v_duplicate_place_id;
79
80      -- when there are no more places, we exit cursor
81      if finished = 1 then leave duplicate_places; end if;
82
83      -- check if this place exists...just in case
84      if (exists(select 1 from pingvalue.places where id = v_duplicate_place_id)) then
85
86          -- atomic actions
87          start transaction;
```

Programación SQL

1. Lenguaje de programación SQL.....	4
2. Base de datos de trabajo: Clínica veterinaria.....	5
3. Procedimientos.....	6
3.1.- Creación, borrado y uso.....	7
3.2.- Uso de Workbench.....	8
3.3.- Parámetros y variables.....	10
4. Funciones.....	13
4.1.- Creación, borrado y uso.....	14
4.2.- Parámetros, resultados de salida y variables.....	15
5. Estructuras de control.....	17
5.1.- Condicionales.....	17
5.1.1. IF-THEN-ELSE.....	17
5.1.2. CASE.....	19
5.2.- Repetitivas.....	21
5.2.1. LOOP.....	21
5.2.2. REPEAT.....	22
5.2.3. WHILE.....	23

6. SQL dinámico.....	24
7. Manejo de errores.....	28
8. Implementación de transacciones.....	31
9. Cursores.....	35
10. Permisos en procedimientos y funciones.....	38
11. Triggers.....	41
12. Exportación de datos en JSON.....	46

Mediante la automatización de determinadas tareas en nuestra base de datos empleando el lenguaje de programación SQL conseguimos, por un lado una mejor modelización de nuestro sistema y por otro incluir parte de la llamada “lógica de negocio” dentro de nuestro servidor de datos, con las ventajas que ello supone.

1. Lenguaje de programación SQL.

En las unidades anteriores hemos trabajado el lenguaje SQL dividiéndolo en diferentes “grupos” de sentencias según su finalidad, distinguiendo entre: Lenguaje de Definición de Datos (DDL), Lenguaje de Manipulación de Datos (DML) y el Lenguaje de Control de Datos (DCL).

Casi todos los Sistemas Gestores de Bases de Datos (SGBD) incorporan sentencias e instrucciones que permiten ampliar el lenguaje SQL añadiendo elementos de la **programación estructurada**, como son bucles, condiciones, variables, etc. Oracle nos ofrece su **PL/SQL**, Microsoft SQL Server lo denomina **T-SQL** y MySQL directamente **MySQL**, tal cual.

Empleando esas extensiones de SQL podremos realizar programas con similares estructuras que otros lenguajes de programación como Java o Python ofrecen. Debemos tener en cuenta, eso si, la limitación de trabajar dentro del entorno de una base de datos.

En esta unidad estudiaremos tres elementos que nos permiten encapsular código de programación SQL bajo un nombre y almacenarlos asociados a una base de datos:

- **Procedimiento almacenado.**
- **Función almacenada.**
- Disparadores o **triggers**.

Para la ejecución de los procedimientos y funciones por parte de los usuarios, éstos deberán tener asignados los permisos correspondientes.

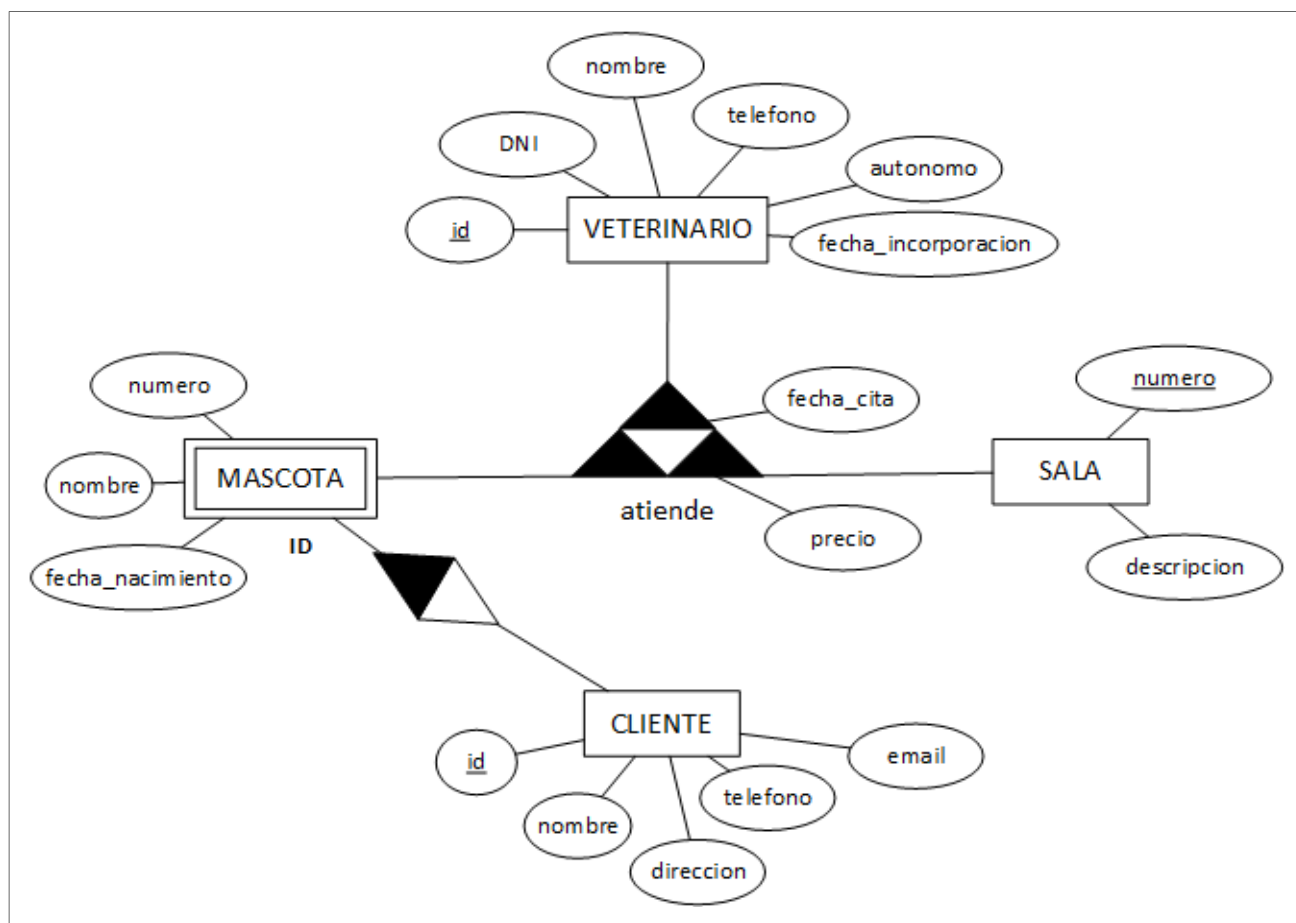
Dedicaremos un apartado a trabajar el llamado **SQL dinámico**, esto es, sentencias SQL que podemos crear en el momento de ejecutar nuestros bloques de código.

También trabajaremos el modo de **gestionar los errores** no controlados producidos durante la ejecución de los bloques de código, así como también el uso de **transacciones** para garantizar la ejecución de sentencias de modo completo.

Finalmente, veremos las extensiones de MySQL que nos permiten exportar datos desde nuestros procedimientos almacenados en formato **JSON**, formato que lenguajes como PHP puede manejar fácilmente.

2. Base de datos de trabajo: Clínica veterinaria

Para el desarrollo de esta unidad vamos a utilizar una base de datos que ya hemos empleado en actividades anteriores, “**Clínica veterinaria**”. Aquí tienes el esquema entidad-relación y el modelo lógico relacional.



VETERINARIO(id, DNI, nombre, telefono, autonomo, fecha_incorporacion)

PK: id

UK: DNI

MASCOTA(id_cliente, numero, nombre, fecha_nacimiento)

PK: (id_cliente, numero)

FK: id_cliente -> CLIENTE

SALA(numero, descripcion)

PK: numero

CLIENTE(id, nombre, telefono, direccion, email)

PK: id

ATIENDE(id_veterinario, numero_sala, id_cliente, numero_mascota, fecha_cita, precio)

PK: (id_veterinario, numero_sala, id_cliente, numero_mascota, fecha_cita)

FK: id_veterinario -> VETERINARIO

FK: numero_sala -> SALA

FK: (id_cliente, numero_mascota) -> MASCOTA

Se ofrece un script sql de creación y carga de la base de datos: [**ud7_clinica_veterinaria.sql**](#). Al lanzar ese script podrás comprobar que las tablas ya aparecen con registros. No contiene las actividades que realizaremos, es decir, si creamos un procedimiento y lanzamos el script el procedimiento desaparece.

3. Procedimientos

Un procedimiento almacenado es un conjunto de instrucciones SQL que se almacena asociado a una base de datos. Puede tener cero o varios parámetros de entrada y cero o varios parámetros de salida. En el caso de incluir una o varias SELECT en MySQL sus resultados se enviarán al programa que hizo la llamada.

3.1.- Creación, borrado y uso

Se crean con la sentencia **CREATE PROCEDURE** y son llamados mediante la sentencia **CALL**. La sintaxis más sencilla para crearlos sería:

```
DELIMITER //
CREATE PROCEDURE nombre_procedimiento ([parametros])
BEGIN
    [variables_locales]
    [cuerpo_procedimiento]
END//
DELIMITER ;
```

La primera y la última instrucción son **DELIMITER //** y **DELIMITER ;**. Esto se debe a que el bloque de código contiene múltiples sentencias separadas por **;** y debemos indicarle a MySQL que no intente ejecutar las sentencias cuando encuentre ese carácter, tal y como haría normalmente. En su lugar le indicamos que utilice como separador o delimitador de sentencias por ejemplo **//**. Una vez definido el procedimiento almacenado, ya podemos volver a emplear el delimitador por defecto.

Si queremos modificar un procedimiento ya creado, deberemos borrarlo y volver a crearlo.

```
DELIMITER //
DROP PROCEDURE IF EXISTS nombre//
CREATE PROCEDURE nombre_procedimiento ([parametros])
...
DELIMITER ;
```

Para ejecutar un procedimiento empleamos la sentencia **CALL**.

```
CALL nombre_procedimiento([parametros]);
```

Un ejemplo muy sencillo lo tenemos con el procedimiento siguiente que nos muestra todas las citas del año en que nos encontremos.

```
DELIMITER //
DROP PROCEDURE IF EXISTS sp_get_citas_anyo_actual//
```

```
CREATE PROCEDURE sp_get_citas_ano_actual()
BEGIN
    SELECT *
    FROM atiende
    WHERE YEAR(fecha_cita) = YEAR(NOW())
    ORDER BY fecha_cita DESC;
END//
DELIMITER ;
```

Podemos llamar el procedimiento con la instrucción:

```
CALL sp_get_citas_ano_actual();
```

Es muy habitual nombrar los procedimientos almacenados con el prefijo “sp_” o “usp_”. En este curso seguiremos el primer prefijo, añadiendo además unos caracteres que nos indiquen la funcionalidad principal. Por ejemplo, si su función es devolver unos datos, usaremos “get_”, si insertamos filas “ins_”, etc.

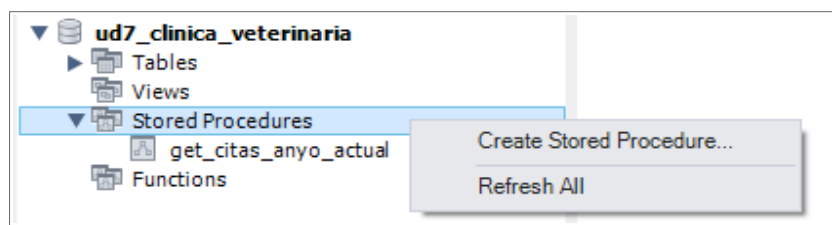


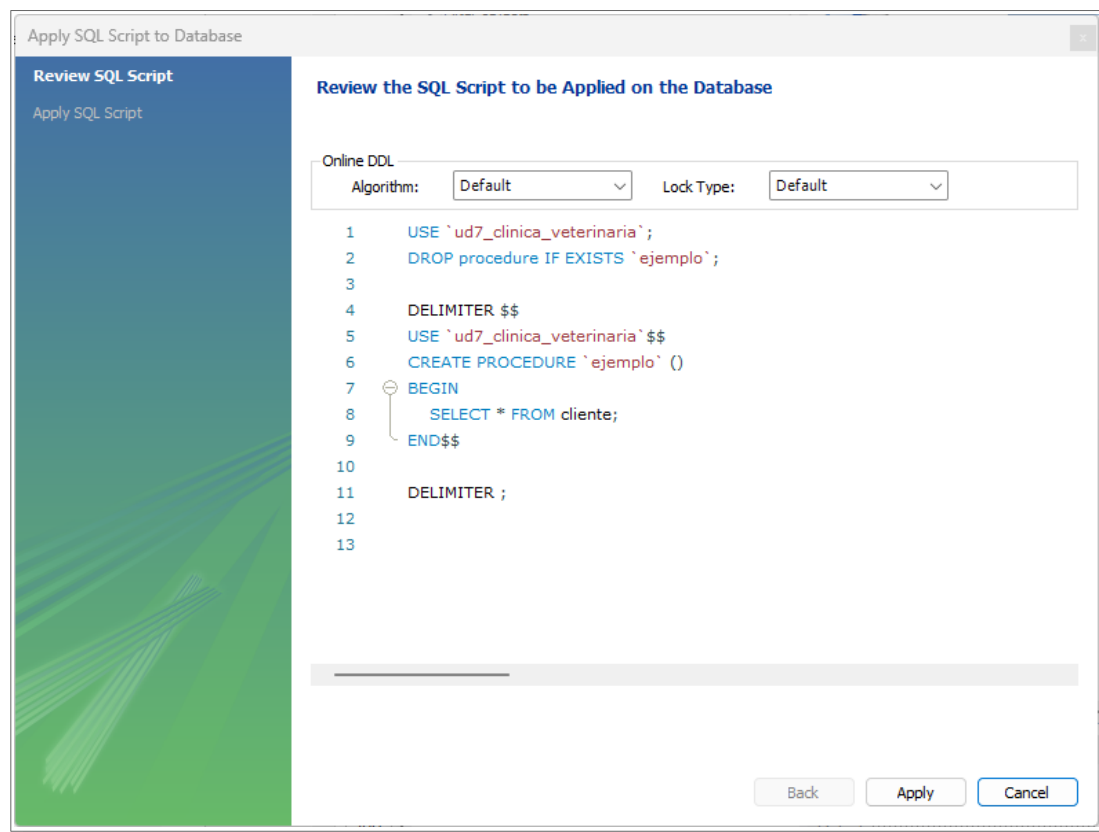
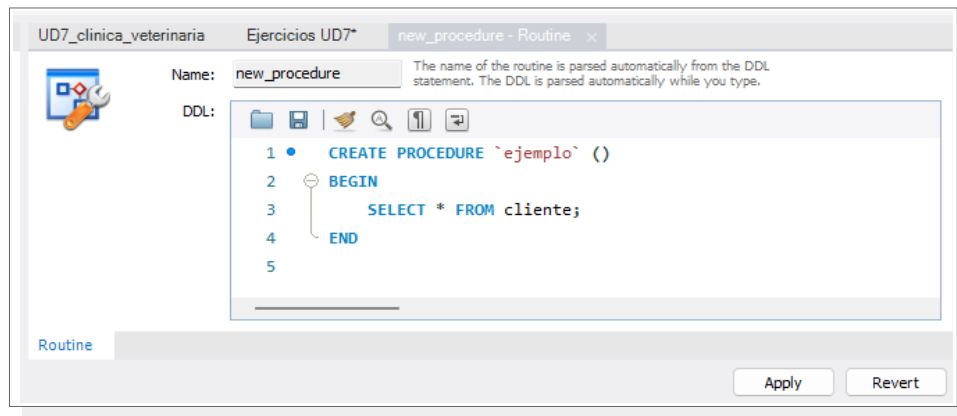
Hazlo tú: Ejecuta el código anterior creando el procedimiento `sp_get_citas_ano_actual`. Comprueba que aparece en el explorador de objetos y ejecútalo.

3.2.- Uso de Workbench

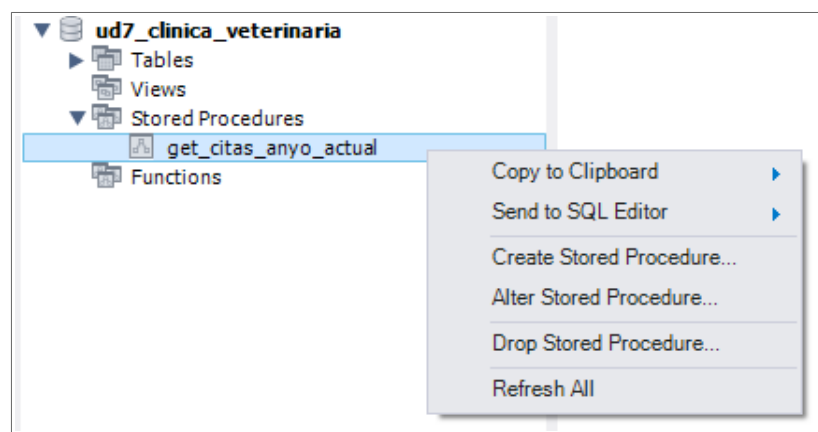
Si estamos empleando la aplicación **MySQL Workbench**, su entorno de desarrollo nos ofrece diversas herramientas para facilitarnos el trabajo en la creación, modificación y borrado de **procedimientos**, **funciones** y **triggers**.

Desde el navegador de objetos y con el botón del ratón podemos operar con los procedimientos almacenados de manera muy sencilla y sin preocuparnos de modificar el valor de DELIMITER. Podemos crear un procedimiento nuevo:





Si el procedimiento está creado, también nos permite modificarlo y/o borrarlo.



3.3.- Parámetros y variables

Como sucede en lenguajes de programación como Java o PHP, los procedimientos en SQL pueden recibir **parámetros** en el momento de la llamada. Tenemos tres tipos de parámetros:

- **Entrada (IN):** al finalizar el procedimiento tienen el mismo valor que en el momento de la llamada y su valor está protegido en el procedimiento. Es un paso por valor.
- **Salida (OUT):** su valor cambia después de la llamada al procedimiento. Paso por referencia.
- **Entrada/salida (INOUT):** combinación de los dos tipos anteriores.

Vamos a mejorar el procedimiento `sp_get_citas_ano_actual`, creando uno nuevo llamado `sp_get_citas_ano_mes` que reciba como parámetros el año y el mes del cual queremos obtener las citas.

```
CREATE PROCEDURE sp_get_citas_ano_mes(  
    IN anyo INT,  
    IN mes INT)  
BEGIN  
    -- Devuelve citas del año y mes recibido  
    SELECT * FROM atiende  
    WHERE YEAR(fecha_cita) = anyo AND MONTH(fecha_cita) = mes  
    ORDER BY fecha_cita DESC;  
END;
```

Ejecutamos el procedimiento anterior con los parámetros de entrada:

```
CALL sp_get_citas_ano_mes(2024, 12);
```

Desde este momento y para minimizar el código SQL, solamente se mostrará el código de creación de los procedimientos de ejemplo.



Hazlo tú: Crea un procedimiento almacenado con nombre `sp_get_citas_entre_dias`, que devuelva las citas entre dos fechas que recibirá como parámetro.

Para trabajar una situación que utilice parámetros de salida, podemos crear un procedimiento llamado `sp_get_estadisticas_citas_dia` que nos devuelva para un día en concreto, el número de citas, la cantidad conseguida y la media por cita.

```
CREATE PROCEDURE sp_get_estadisticas_citas_dia(  
    IN dia DATE,  
    OUT citas INT,  
    OUT total DECIMAL(10, 2),  
    OUT media DECIMAL(10, 2))  
BEGIN  
    -- Para el día recibido devuelve el número de citas, total y media  
    SELECT COUNT(*), SUM(precio), ROUND(AVG(precio), 2)  
        INTO citas, total, media  
    FROM atiende  
    WHERE fecha_cita = dia;  
END;
```

Para **asignarle un valor** a los parámetros `citas`, `total` y `media` usamos la sentencia **SELECT...INTO**, con una asignación de tres valores. Se podría realizar un `SELECT...INTO` para cada parámetro, pero es más óptimo de este modo.

Otro método para asignar un valor a un parámetro es mediante la sentencia **SET**. En ese caso deberemos asignar un solo valor cada vez.

```
...  
SET citas = (SELECT COUNT(*) FROM atiende WHERE fecha_cita = dia);  
SET total = (SELECT SUM(precio) FROM atiende WHERE fecha_cita = dia);  
SET media = (SELECT ROUND(AVG(precio), 2) FROM atiende WHERE fecha_cita = dia);  
...
```

Tanto `SET` como `SELECT...INTO` podemos usarlos para asignar valores concretos que no resulten directamente de una consulta.

Si queremos ejecutar el procedimiento anterior tenemos el inconveniente de que debemos almacenar el resultado de los parámetros de salida en algún espacio de memoria que nos permita recuperar sus valores. Aquí es donde aparece el concepto de **variable** en SQL.

En cuanto al nombre que usemos para los parámetros hay que llevar cuidado si coincide con los campos de las tablas. Aunque no lo haremos en este curso, es habitual usar el prefijo “p_” en su nombre.

Como en cualquier lenguaje de programación, SQL permite **declarar** variables que almacenarán valores y podremos hacer uso de ellos. De hecho, cuando creamos un procedimiento con parámetros, estos parámetros son variables que declaramos al declarar el procedimiento e indicamos un tipo de dato concreto.

Si queremos declarar una variable fuera de un procedimiento, lo que llamamos **variables definidas por el usuario**, como haríamos para ejecutar el procedimiento anterior, es tan fácil como añadir el prefijo “@” al nombre de la variable. No es necesario indicar el tipo de dato que almacenará. Veamos como hacer la llamada:

```
CALL sp_get_estadisticas_citas_dia('2025-03-06', @num_citas, @precio_total,
@precio_medio);
SELECT @num_citas AS citas, @precio_total AS precio, @precio_medio AS media;
```



Hazlo tú: Crea el procedimiento almacenado sp_get_estadisticas_citas_dia y comprueba que funciona

Dentro de los procedimientos y funciones podemos crear variables, llamadas **variables locales**, con ámbito en el bloque BEGIN...END. Para su declaración empleamos la sentencia **DECLARE** con la sintaxis:

```
DECLARE nombre_variable, [nombre_variable....] tipo [DEFAULT valor_defecto]
```

Como norma que seguiremos durante el curso, las declaraciones se harán justo después de BEGIN e incluiremos siempre el prefijo “v_”.

Para practicar el uso de variables locales en procedimientos, vamos a crear una versión del anterior procedimiento de estadísticas de citas, donde no usaremos parámetros de salida para devolver resultados. Almacenaremos los datos en variables locales y los mostraremos mediante una SELECT.

Es aconsejable el uso de comentarios que faciliten la comprensión del código.

```

CREATE PROCEDURE sp_get_estadisticas_citas_dia_v2(IN dia DATE)
BEGIN
    -- Para el día recibido devuelve número de citas, cantidad total y media
    -- variables
    DECLARE v_citas INT;
    DECLARE v_total, v_media DECIMAL(10, 2);

    -- búsqueda de datos
    SELECT COUNT(*), SUM(precio) INTO v_citas, v_total
    FROM atiende
    WHERE fecha_cita = dia;

    -- cálculo de datos
    SET v_media = ROUND(v_total / v_citas, 2);

    -- mostramos info
    SELECT v_citas AS citas, v_total AS precio, v_media AS media;
END;

```

Llamar a este procedimiento es ahora más sencillo, ya que no necesitamos parámetros de salida.

```
CALL sp_get_estadisticas_citas_dia_v2('2025-03-06');
```



Hazlo tú: Crea el procedimiento almacenado `sp_get_estadisticas_citas_dia_v2` y comprueba que funciona. Analiza el modo en que se declaran las variables locales, y como se utiliza SET.

4. Funciones

Al igual que los procedimientos, las funciones están formadas por un conjunto de instrucciones que se almacenan en la base de datos bajo un nombre. Sin embargo, en las funciones todos los **parámetros son de entrada** y siempre devuelven un **resultado de salida**.

4.1.- Creación, borrado y uso

Se crean con la sentencia **CREATE FUNCTION** y se llaman con la sentencia **SELECT** o dentro de una expresión. La sintaxis más sencilla para crearlas sería:

```
DELIMITER //
CREATE FUNCTION nombre_funcion ([parametros])
RETURNS tipo
[características]
BEGIN
    [variables_locales]
    [cuerpo_funcion]
    RETURN resultado_salida;
END//
DELIMITER ;
```

En **características** indicaremos diferentes opciones que pueden ayudar al servidor MySQL a optimizar la ejecución de la función. Su estudio no está dentro del ámbito de este curso, sin embargo sí indicamos que es necesario al menos indicar una de las tres características siguientes.

- **DETERMINISTIC**: indica que la función siempre devuelve el mismo resultado cuando se utilizan los mismos parámetros de entrada.
- **NO SQL**: indica que la función no contiene sentencias SQL.
- **READS SQL DATA**: indica que la función no modifica los datos de la base de datos y que contiene sentencias de lectura de datos, como SELECT.

Si queremos modificar una función ya creada, deberemos borrarla y volver a crearla.

```
DELIMITER //
DROP FUNCTION IF EXISTS nombre//
CREATE FUNCTION nombre_funcion ([parametros])
...
DELIMITER ;
```

Seguiremos la norma, de usar el prefijo “fn_” para nombrar a las funciones.

Un primer ejemplo de función muy sencillo sería `fn_dia_semana`, que devuelve el día de la semana en que nos encontremos.

```
CREATE FUNCTION fn_dia_semana()  
RETURNS VARCHAR(50)  
DETERMINISTIC  
BEGIN  
    RETURN DAYNAME(NOW());  
END;
```

Podemos usar la función anterior de diferentes maneras, por ejemplo directamente en una `SELECT`:

```
SELECT fn_dia_semana() AS dia_semana;
```

También podemos asignar el valor a una variable.

```
SET @dia = fn_dia_semana();  
SELECT @dia;
```

Y como no, podríamos utilizar funciones dentro de otras funciones o procedimientos almacenados que creamos y usemos **en la misma base de datos**.

La aplicación **MySQL Workbench** nos facilita la creación de funciones del mismo modo que sucede con los procedimientos.

4.2.- Parámetros, resultados de salida y variables

En las funciones todos los parámetros recibidos son parámetros de entrada, por lo que no es necesario indicarlo con `IN`. En cuanto al resultado de salida solamente tendremos un valor devuelto y emplearemos la sentencia **RETURN** para ello. Esta sentencia **termina la ejecución** de la función.

Las variables locales a la función se declaran y utilizan del mismo modo que hacíamos con los procedimientos.

Veamos otro ejemplo de función, `fn_num_clientes`, que nos devuelve el número de clientes. Declaramos variables para almacenar el resultado de salida.

```

CREATE FUNCTION fn_num_clientes()
RETURNS INT
READS SQL DATA
BEGIN
    -- número de clientes
    DECLARE v_num INT;

    -- calculamos
    SELECT COUNT(*) INTO v_num FROM cliente;

    -- devolvemos resultado
    RETURN v_num;
END;

```

Podemos usar la función directamente en una SELECT:

```

SELECT fn_num_clientes() AS num_clientes;

```

O con variables.

```

SET @num_clientes = fn_num_clientes();
SELECT @num_clientes;

```

Un último ejemplo. Función que recibe el código de un cliente y nos devuelve el gasto total realizado por sus citas.

```

CREATE FUNCTION fn_gasto_cliente(codigo INT)
RETURNS DECIMAL(10, 2)
READS SQL DATA
BEGIN
    -- gasto total citas de cliente
    DECLARE v_total DECIMAL(10, 2);

    -- calculamos
    SELECT SUM(precio) INTO v_total
    FROM atiende
    WHERE id_cliente = codigo;

```



```
-- devolvemos resultado
RETURN v_total;
END;
```

Probamos que funciona.

```
SELECT fn_gasto_cliente(1) AS gasto_cliente;
```



Hazlo tú: Crea la función `fn_gasto_cliente` y comprueba que funciona correctamente.

5. Estructuras de control

Como cualquier otro lenguaje de programación SQL permite emplear sentencias con las que controlar el flujo de ejecución del código.

5.1.- Condicionales

Instrucciones que permiten ejecutar un bloque de instrucciones u otro, dependiendo de una condición.

5.1.1. IF-THEN-ELSE

Ejecuta una o varias sentencias SQL si se evalúa a cierto una condición lógica, en caso contrario el flujo de ejecución después de la sentencia. Se pueden anidar y su sintaxis sería:

```
IF condicion THEN
    sentencias
[ELSEIF condicion THEN sentencias]
[ELSE
    sentencias]
END IF;
```

Como ejemplo, podemos hacer una variante de la función ya implementada en SQL, `GREATEST()`, que nos devuelve el mayor de los valores recibidos. A continuación, se muestra la función `fn_compara_dos_numeros` que recibe dos números y compara sus valores devolviendo un mensaje con el resultado.

```
CREATE FUNCTION fn_compara_dos_numeros(n1 FLOAT, n2 FLOAT)
RETURNS VARCHAR(100)
DETERMINISTIC
BEGIN
    -- compara dos números devolviendo texto resultado
    DECLARE v_resultado VARCHAR(100);

    -- comparación tres opciones
    IF n1 = n2 THEN
        SET v_resultado = 'Los dos números son iguales';
    ELSEIF n1 > n2 THEN
        SET v_resultado = 'El primer número es mayor';
    ELSE
        SET v_resultado = 'El segundo número es mayor';
    END IF;

    -- devuelve resultado
    RETURN v_resultado;
END;
```

Comprobamos su funcionamiento con:

```
SELECT fn_compara_dos_numeros(2, 5) AS comparacion;
```



Hazlo tú: Crea la función `fn_compara_tres_numeros` con una funcionalidad similar a la anterior pero en este caso la comparación se realizará sobre tres números.

5.1.2. CASE

Similar a la instrucción `switch` de otros lenguajes. Comprueba una serie de condiciones y ejecuta las instrucciones asociadas a aquel valor donde se cumple la condición indicada.

Tenemos dos sintaxis posibles. La primera, probablemente la que más usaremos, donde evaluaremos la rama cuyo valor coincida con el valor del **CASE**:

```
CASE valor_case
  WHEN valor1 THEN sentencias1
  WHEN valor2 THEN sentencias2
  ...
  [ELSE sentencias_else]
END CASE
```

La otra opción evalúa cada condición en su propia sentencia `WHEN`.

```
CASE
  WHEN condicion1 THEN sentencias1
  WHEN condicion2 THEN sentencias2
  ...
  [ELSE sentencias_else]
END CASE
```

Como ejemplo tenemos a continuación el procedimiento `sp_get_mensaje_cumple` que recibe una fecha y comprueba si es el cumpleaños de alguna mascota, construyendo un mensaje con información del cliente.

```
CREATE PROCEDURE sp_get_mensaje_cumple(
  IN dia DATE,
  OUT msj VARCHAR(100))
BEGIN
  -- Busca mascotas con cumple
  -- variables
  DECLARE v_cliente;
  DECLARE v_nombre_cliente, v_nombre_mascota VARCHAR(50);
  DECLARE v_cumple INT;
```

```

-- búsqueda de datos
SELECT COUNT(*) INTO v_cumple
FROM mascota
WHERE fecha_nacimiento = dia;

CASE v_cumple
    WHEN 0 THEN
        SET msj = 'Ninguna mascota con cumple hoy';
    WHEN 1 THEN
        -- buscamos cod cliente y nombre mascota
        SELECT id_cliente, nombre
        INTO v_cliente, v_nombre_mascota
        FROM mascota
        WHERE fecha_nacimiento = dia;

        -- nos falta el nombre del cliente
        SELECT nombre INTO v_nombre_cliente
        FROM cliente
        WHERE id = v_cliente;

        SET msj = CONCAT('Hoy es el cumple de ',
            v_nombre_mascota, ', dueño ', v_nombre_cliente);
    ELSE
        -- en este punto, hay varios cumple
        SET msj = 'Varias mascotas tienen cumple hoy';
END CASE;
END;

```

Se puede comprobar que vamos aumentando la complejidad del código a medida que avanzamos.



Hazlo tú: Implementa el procedimiento anterior y comprueba que funciona correctamente, esto es, deberás buscar casos de prueba en los que se den las tres situaciones posibles implementadas en el CASE.

5.2.- Repetitivas

Las estructuras repetitivas o iterativas, presentes en todos los lenguajes de programación, permiten ejecutar una sección de código una o varias veces en base a la **evaluación de una condición** lógica. MySQL ofrece tres tipos de estructuras repetitivas: **LOOP**, **WHILE** y **REPEAT**.

Llamaremos **bucle** al conjunto de sentencias que se repiten, mientras que definimos **iteración** como la repetición de ese código.

Como sucede en otros lenguajes, SQL nos ofrece sentencias para forzar el abandono de un bucle, en este caso :

- **LEAVE**: la ejecución continúa después del bucle.
- **ITERATE**: la ejecución continúa en la siguiente iteración del bucle.

Tanto LEAVE como ITERATE deben recibir la **etiqueta** que le hemos dado al bucle, lo veremos en los ejemplos siguientes.

Los bucles se pueden crear dentro de otros bucles, es lo que llamamos **anidamiento**. En esos casos el empleo de etiquetas para identificarlos, aunque no usemos ni LEAVE ni ITERATE, es muy útil para mejorar la legibilidad del código. También es muy aconsejable realizar una correcta **indentación** (sangrado).

5.2.1. LOOP

Crea un bucle que continúa hasta que se encuentra una condición explícita de salida del mismo y se aplica LEAVE o ITERATE, en caso contrario será un bucle infinito.

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

En el ejemplo siguiente, el procedimiento `sp_ejemplo_loop` emplea un bucle LOOP que finaliza gracias al uso de LEAVE. El procedimiento suma una serie de valores desde 1 hasta la cantidad indicada.

```

CREATE PROCEDURE sp_ejemplo_loop(
    IN fin INT,
    OUT suma INT)
BEGIN
    -- Suma valores hasta la cantidad fin incluida
    -- para saber la posición actual
    DECLARE v_actual INT DEFAULT 1;

    -- iniciamos la suma
    SET suma = 0;
    -- bucle hasta fin
    bucle: LOOP
        -- si hemos sobrepasado fin salimos
        IF v_actual > fin THEN
            LEAVE bucle;
        END IF;
        -- si no, sumamos y avanzamos
        SET suma = suma + v_actual;
        SET v_actual = v_actual + 1;
    END LOOP bucle;
END;

```

Probamos su funcionamiento con:

```

CALL sp_ejemplo_loop(5, @suma);
SELECT @suma;

```

5.2.2. REPEAT

Ejecuta un bloque de código repetidamente hasta que la condición especificada es cierta, esto es, se ejecuta mientras no se cumpla la condición. La comprobación se realiza **después** de cada iteración.

```

[etiqueta:] REPEAT
    sentencias
UNTIL condicion
END REPEAT [etiqueta];

```

Si modificamos `sp_ejemplo_loop` para que utilice `REPEAT` quedaría:

```
CREATE PROCEDURE sp_ejemplo_repeat(  
    IN fin INT,  
    OUT suma INT)  
BEGIN  
    -- Suma valores hasta la cantidad fin incluida  
    -- para saber la posición actual  
    DECLARE v_actual INT DEFAULT 1;  
  
    -- iniciamos la suma  
    SET suma = 0;  
  
    -- bucle hasta fin  
    bucle: REPEAT  
        SET suma = suma + v_actual;  
        SET v_actual = v_actual + 1;  
    UNTIL v_actual > fin  
    END REPEAT bucle;  
END;
```

Comprobamos que obtenemos el mismo resultado.

```
CALL sp_ejemplo_repeat(5, @suma);  
SELECT @suma;
```

5.2.3. WHILE

Ejecuta un bloque de código siempre que la condición especificada sea cierta, esto es, se ejecuta hasta que la condición sea false. La comprobación se realiza al **inicio** de cada iteración.

```
[etiqueta:] WHILE condicion DO  
    sentencias  
END WHILE [etiqueta];
```

Modificamos `sp_ejemplo_loop` para que utilice `WHILE` dando como resultado:

```

CREATE PROCEDURE sp_ejemplo_while(
    IN fin INT,
    OUT suma INT)
BEGIN
    -- Suma valores hasta la cantidad fin incluida
    -- para saber la posición actual
    DECLARE v_actual INT DEFAULT 1;

    -- iniciamos la suma
    SET suma = 0;

    -- bucle hasta fin
    bucle: WHILE v_actual <= fin DO
        SET suma = suma + v_actual;
        SET v_actual = v_actual + 1;
    END WHILE bucle;
END;

```

Comprobamos que obtenemos el mismo resultado.

```

CALL sp_ejemplo_while(5, @suma);
SELECT @suma;

```



Hazlo tú: Realiza un procedimiento llamado `sp_ejemplo_bucle_desde_hasta` con un funcionamiento muy parecido a los tres ejemplos anteriores, solo que la suma de números se realizará entre dos valores (inicio y fin). Puedes emplear el bucle que prefieras de los tres tipos vistos. Comprueba que funciona correctamente

6. SQL dinámico

El SQL dinámico se refiere a una técnica que incluyen prácticamente todos los SGBD en su lenguaje SQL y que permite crear sentencias SQL **en tiempo de ejecución**. En ocasiones nos encontraremos con situaciones donde no sabemos de antemano, esto es, cuando compilamos (guardamos) el procedimiento o función, el texto completo de la sentencia SQL que necesitamos ejecutar.

Otra de las razones de su uso es evitar el problema de seguridad por **inyección de SQL**, donde un usuario añade código SQL a otro ya existente cambiando la lógica de funcionamiento de la instrucción inicial. Lenguajes como Java o PHP incluyen desde hace tiempo las declaraciones preparadas o **prepared statements**. Esa misma idea es la que MySQL utiliza.

El uso de estas declaraciones preparadas supone seguir tres pasos: **creación** → **uso** → **liberación**, con las siguientes instrucciones.

- **PREPARE**: crea, compila y almacena la instrucción SQL que debemos pasársela como un literal cadena o como una variable de usuario (empieza por @) nunca una variable local. Permite crear instrucciones parametrizadas.

```
PREPARE nombre_stmt FROM @declaracion_stmt
```

- **EXECUTE**: ejecuta la instrucción pudiendo recibir parámetros. Una vez preparada la instrucción, se puede ejecutar las veces que se desee. Si se han definido parámetros se pasan sus valores finales que serán valores concretos o variables de usuario (@)

```
EXECUTE nombre_stmt [USING @valor1, @valor2, ...]
```

- **DEALLOCATE PREPARE**: es muy aconsejable incluir esta sentencia cuando no se vaya a emplear de nuevo la instrucción SQL dentro del procedimiento o función.

```
DEALLOCATE PREPARE nombre_stmt
```

Vamos a realizar un primer ejemplo muy sencillo donde usaremos SQL dinámico. El procedimiento `sp_get_datos_tabla` recibe como parámetro el nombre de una tabla de nuestra base de datos y devolverá una `SELECT` de todos sus campos.

```
CREATE PROCEDURE sp_get_datos_tabla(  
    IN nombre_tabla VARCHAR(50)  
)  
BEGIN  
    -- Muestra todos los campos y filas de la tabla recibida
```

```

SET @declaracion_stmt = CONCAT('SELECT * FROM ', nombre_tabla);

-- preparamos
PREPARE stmt_tabla FROM @declaracion_stmt;

-- ejecutamos
EXECUTE stmt_tabla;

-- liberamos
DEALLOCATE PREPARE stmt_tabla;

END

```

Comprobamos que funciona:

```

CALL sp_get_datos_tabla('cliente');
CALL sp_get_datos_tabla('atiende');

```

En el anterior procedimiento hemos creado la sentencia concatenando trozos hasta tener la intrucción final. También podemos crear una instrucción SQL con **parámetros** que le pasaremos al realizar el EXECUTE. Esos parámetros nunca podrán ser ni nombres de tablas ni nombres de columnas.

En el siguiente ejemplo se ha creado un procedimiento `sp_ins_veterinario` que realizará la inserción de un nuevo veterinario con valores para todos sus campos. Un parámetro de salida nos devolverá el id asignado al nuevo registro, no contemplamos la posibilidad de errores en la inserción. Para obtener el último autonumérico utilizado empleamos la función `LAST_INSERT_ID`. En [este enlace](#) tienes más información sobre esa función.

```

CREATE PROCEDURE sp_ins_veterinario(
    IN DNI VARCHAR(10),
    IN nombre VARCHAR(50),
    IN telefono VARCHAR(50),
    IN autonomo BOOL,
    IN fecha_incorporacion DATE,
    OUT id INT)
BEGIN

```

```

-- inserta nuevo veterinario comprobando la correcta inserción

-- Construimos la sentencia
SET @declaracion_stmt =
    'INSERT INTO veterinario VALUES(NULL, ?, ?, ?, ?, ?)';
PREPARE prepared_stmt FROM @declaracion_stmt;

-- ejecutamos pasándole parámetros
-- solo admite variables de usuario
SET @DNI = dni;
SET @nombre = nombre;
SET @telefono = telefono;
SET @autonomo = autonomo;
SET @fecha_incorporacion = fecha_incorporacion;
EXECUTE prepared_stmt
USING @DNI, @nombre, @telefono, @autonomo, @fecha_incorporacion;

-- obtenemos el id asignado
SET id = LAST_INSERT_ID();

-- liberamos
DEALLOCATE PREPARE prepared_stmt;
END

```

Podemos probar el procedimiento con el ejemplo.

```

CALL sp_ins_veterinario('1A', 'Josefina Altos', NULL, 1, '2024-01-06', @id);
SELECT @id;

```



Hazlo tú: Realiza un procedimiento un procedimiento similar `sp_ins_veterinario` pero sin usar SQL dinámico, esto es, haciendo un `INSERT` directamente con los parámetros de entrada del procedimiento.

7. Manejo de errores

Por defecto, si una instrucción SQL dentro de un procedimiento o función genera un error su ejecución se detendrá devolviendo el control al programa que ha hecho la llamada. Si no queremos que esto suceda debemos crear un manejador de errores o **error handler** con la siguiente sintaxis.

```
DECLARE accion_manejador HANDLER
FOR condiciones
instrucciones
```

Si se produce alguna de las **condiciones** el manejador se activará y ejecutará las **instrucciones** indicadas que podrían ser una simple asignación de valores a una variable o código más complejo en un BEGIN...END. Después de ejecutar las instrucciones se realizará la **acción_manejador** indicada.

Las acciones soportadas son:

- **EXIT**: se finaliza la ejecución del bloque BEGIN...END donde se declare el manejador.
- **CONTINUE**: el programa continua.

Las condiciones pueden ser de varios tipos, lo que nos permite afinar bastante el error concreto que buscamos manejar:

- **mysql_error_code**: entero indicando el código de error MySQL, por ejemplo 1062 nos indica que se intenta insertar un valor de clave repetido.
- **SQLSTATE valor**: indicamos el valor de la variable SQLSTATE, un texto de cinco caracteres que proporciona información sobre el resultado de una instrucción SQL. Los primeros dos caracteres indican la categoría general del error y los tres restantes dan información más específica. Los valores son tomados del ANSI SQL y están **más estandarizados entre los diferentes SGBD** que el código de error MySQL. El código de error 1062 se corresponde con el SQLSTATE "23000".

- **SQLWARNING**: equivale a indicar los SQLSTATE que empiecen por “01”.
- **NOT FOUND**: equivale a indicar los SQLSTATE que empiecen por “02”. Lo usaremos cuando trabajemos con cursores para controlar qué hacer cuando hemos alcanzado el final del conjunto de datos.
- **SQLException**: engloba los SQLSTATE que empiecen por “00”, “01” y “02”.

No debemos olvidar que, si no hemos declarado ningún tipo de manejador sucederá lo siguiente:

- Para una SQLException: el programa termina.
- Para una SQLWARNING: el programa continua ejecutándose.

En general, a la hora de decidir qué condición o condiciones consideraremos en nuestro manejo de errores, se suele optar por dos opciones:

- Manejar **errores concretos** haciendo uso del `mysql_error_code`.
- Gestionar errores de un **modo más amplio** empleando SQLException o incluso incluyendo también SQLWARNING.

Si queremos modificar el procedimiento `sp_ins_veterinario` para que si se produce un intento de insertar un veterinario con un DNI ya existente controle el error, el 1062, y le asigne el valor -1 al parámetro de salida `id` nos bastaría con añadir la siguiente línea de código al principio del procedimiento.

```
...
BEGIN
    -- inserta nuevo veterinario comprobando la correcta inserción
    -- control error clave repetida
    DECLARE EXIT HANDLER FOR 1062
    SET id = -1;

    -- Construimos la sentencia
    SET @declaracion_stmt = 'INSERT INTO veterinario (DNI, nombre, telefono, ';
    ...
```



Hazlo tú: Crea el procedimiento almacenado `sp_ins_veterinario_v2` modificando `sp_ins_veterinario` y añadiendo el manejo del error 1062 tal y como se indica. Comprueba que controla correctamente el error y el valor devuelto en `id` es -1 cuando se intenta insertar un veterinario con DNI repetido.

Si quisiéramos ampliar el abanico de errores controlados en nuestro procedimiento podemos optar por manejar `SQLException`. Para ello podríamos utilizar el control de errores siguiente.

```
DECLARE EXIT HANDLER FOR SQLException SET id = -1;
```



Hazlo tú: Crea el procedimiento almacenado `sp_ins_veterinario_v3` modificando la gestión de errores de `sp_ins_veterinario_v2`. Comprueba que controla correctamente el error tanto cuando se intenta insertar un veterinario con DNI repetido como un nombre con valor `NULL`.



Hazlo tú: Modifica el procedimiento `sp_ins_veterinario_v3` para que, en caso de error, además de poner el parámetro `id` a -1 también devuelva una `SELECT` con el texto: "Error al insertar veterinario con DNI: `valor_dni`". Si no se produce error devolveremos una `SELECT` con el texto "Inserción correcta veterinario con DNI: `valor_dni`".

8. Implementación de transacciones

En la Unidad 1 ya tratamos el concepto de **transacción** y las cuatro propiedades englobadas en el acrónimo **ACID** (Atomicity, Consistency, Isolation y Durability).

Como repaso, comentar que una transacción en un SGBD se puede definir como un conjunto de instrucciones SQL que se deben realizar como **una única unidad** de trabajo, esto es, una transacción nunca se completará a menos que cada una de las instrucciones individuales que la componen se complete correctamente.

En MySQL, las instrucciones que permiten manejar transacciones forman el lenguaje **TCL** (Transaction Control Language). Las principales son:

- **START TRANSACTION**: marca el inicio de la transacción.
- **COMMIT**: aplica en la base de datos los cambios realizados durante la transacción.
- **ROLLBACK**: deshace los cambios realizados volviendo al estado anterior al inicio de la transacción.

Por defecto, cuando ejecutamos una instrucción SQL, el servidor MySQL la engloba dentro de una transacción y realiza automáticamente el COMMIT correspondiente. Este modo de trabajo se le denomina **AUTOCOMMIT**. Podemos comprobar que estamos en ese modo consultando la siguiente variable global, observando que toma el valor 1:

```
SELECT @@autocommit;
```

Podemos modificar ese valor fácilmente:

```
SET AUTOCOMMIT = 0;  
SET AUTOCOMMIT = 1;
```

No olvidemos que hay algunas instrucciones SQL a las que no se puede hacer un ROLLBACK, son las instrucciones DDL, como ALTER TABLE, DROP TABLE, CREATE PROCEDURE...

Si queremos añadir las ventajas del uso de transacciones al diseño de procedimientos o funciones en SQL, existe una forma muy habitual de hacerlo que consiste en:

- Declarar un manejador de errores al principio del procedimiento incluyendo un ROLLBACK.
- Iniciar la transacción justo después del manejador de errores
- Realizar las acciones.
- Hacer COMMIT como última instrucción.

Quedaría un código SQL como:

```
CREATE PROCEDURE nombre_procedimiento ([parametros])
BEGIN
    [variables_locales]
    DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING
    BEGIN
        ROLLBACK;
        ...
    END
    START TRANSACTION;
    ...
    COMMIT TRANSACTION;
END
```

El siguiente procedimiento, `sp_upd_intercambia_propietario`, recibe dos códigos de cliente. Se encarga de intercambiar las mascotas que tienen ambos clientes. Empleamos manejo de errores y transacciones.

```
CREATE PROCEDURE sp_upd_intercambia_propietarios(
    IN propietario_1 INT,
    IN propietario_2 INT)
BEGIN
    -- intercambia las mascotas de dos clientes
    -- no comprobamos si tienen mascotas
    -- control general errores
```



```

DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING
BEGIN
    ROLLBACK;
    SELECT CONCAT('Error al intercambiar mascotas de ',
        propietario_1, ' y ', propietario_2) AS mensaje;
END;

-- inicio transacción
START TRANSACTION;

-- necesario deshabilitar la comprobación de claves ajenas
SET FOREIGN_KEY_CHECKS = 0;
UPDATE mascota SET id_cliente = 0 WHERE id_cliente = propietario_2;
UPDATE mascota SET id_cliente = propietario_2
WHERE id_cliente = propietario_1;
SET FOREIGN_KEY_CHECKS = 1;
UPDATE mascota SET id_cliente = propietario_1 WHERE id_cliente = 0;

-- confirmación cambios
COMMIT;
SELECT CONCAT('Intercambio correcto de mascotas de ',
    propietario_1, ' y ', propietario_2) AS mensaje;
END

```

Podemos comprobar el funcionamiento mediante las llamadas.

```

CALL sp_upd_intercambia_propietarios(1, 2);
CALL sp_upd_intercambia_propietarios(2, 1);

```

Lógicamente, el procedimiento se podría mejorar incluyendo diversas comprobaciones y por ejemplo, con un parámetro de salida que nos indique si se ha producido un error o no con un valor 0 / 1, en lugar de usar SELECT.



Hazlo tú: Crea el procedimiento almacenado `sp_upd_intercambia_propietarios` y comprueba que funciona correctamente. Presta especial atención al modo de incorporar las transacciones y al control de errores.

Para realizar el procedimiento anterior no es necesario el uso de SQL dinámico, sin embargo, podemos tomarlo como un buen ejemplo para repasar las prepared statement, como podemos ver en la siguiente variante.

```
CREATE PROCEDURE sp_upd_intercambia_propietarios_v2(
    IN propietario_1 INT,
    IN propietario_2 INT)
BEGIN
    -- intercambia las mascotas de dos clientes
    -- no comprobamos si tienen mascotas

    -- control general errores
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
        SELECT CONCAT('Error al intercambiar mascotas de ',
            propietario_1, ' y ', propietario_2) AS mensaje;
    END;

    -- inicio transacción
    START TRANSACTION;

    -- preparamos la sentencia, que podremos usar varias veces
    SET @propietario_1 = propietario_1;
    SET @propietario_2 = propietario_2;
    SET @cero = 0;
    SET @declaracion_stmt = 'UPDATE mascota SET id_cliente = ? WHERE id_cliente
= ?';
    PREPARE prepared_stmt FROM @declaracion_stmt;

    -- necesario deshabilitar la comprobación de claves ajenas
    SET FOREIGN_KEY_CHECKS = 0;
    -- podemos reutilizar varias veces la misma prepared statement
    EXECUTE prepared_stmt USING @cero, @propietario_2;
    EXECUTE prepared_stmt USING @propietario_2, @propietario_1;
    SET FOREIGN_KEY_CHECKS = 1;
```

```
EXECUTE prepared_stmt USING @propietario_1, @cero;

-- confirmación cambios
COMMIT;
SELECT CONCAT('Intercambio correcto de mascotas de ',
              propietario_1, ' y ', propietario_2) AS mensaje;

-- liberamos sentencia
DEALLOCATE PREPARE prepared_stmt;
END;
```

9. Cursores

En MySQL un cursor es un objeto de la base de datos empleado para **iterar** (recorrer) el resultado devuelto por una consulta SELECT. Es habitual emplearlos en procedimientos, funciones y disparadores cuando necesitamos procesar de manera individual cada una de las filas devueltas por una consulta. La forma de trabajar con cursores implica seguir los siguientes pasos en el orden indicado:

DECLARAR → ABRIR → RECORRER HASTA FINAL → CERRAR

Las instrucciones que nos permiten realizar esas acciones son:

- **DECLARE:** define el cursor indicándole la consulta a emplear.

```
DECLARE nombre_cursor CURSOR FOR consulta;
```

- **OPEN:** abre el cursor e inicializa su estado.

```
OPEN nombre_cursor;
```

- **FETCH:** una vez abierto el cursor nos recorreremos sus filas con esa instrucción almacenando los valores devueltos. Cuando no queden más filas se lanzará el error NOT FOUND, por lo que debemos crear un control de errores para ese caso que nos permita detectar que se ha llegado al final del cursor y detener la iteración.

```
FETCH nombre_cursor INTO variables;
```

- **CLOSE:** cerramos el cursor al terminar de usarlo.

```
CLOSE nombre_cursor;
```

Hay ocasiones donde lo mismo que podemos hacer implementando un cursor sería posible hacerlo directamente con una sentencia SQL, más o menos compleja, que consulte / actualice / modifique tablas. Es ya decisión del DBA optar por una opción u otra.

Hay que tener en cuenta que:

- Los cursores son de solo lectura, no permiten actualizar los datos.
- Solamente se pueden recorrer en una dirección y no podemos saltarnos filas
- En un procedimiento se declaran después de declarar las variables locales y antes del manejador de errores.
- La estructura repetitiva empleada para recorrer el cursor puede ser cualquiera de las vistas anteriormente. Se aconseja emplear siempre el mismo tipo.

Veamos un ejemplo muy sencillo de un procedimiento donde nos recorremos los veterinarios que son autónomos y calculamos las citas que han tenido en un intervalo de fechas.

```
CREATE PROCEDURE sp_get_citas_autonomos(  
    IN fecha_desde DATE,  
    IN fecha_hasta DATE,  
    OUT num_citas INT)  
BEGIN  
    -- citas de veterinarios autónomos en intervalo fechas  
    DECLARE v_id_veterinario INT;  
    DECLARE v_fin INT DEFAULT 0;  
    DECLARE v_num_citas_vet INT DEFAULT 0;  
    DECLARE c_veterinarios CURSOR  
    FOR SELECT id FROM veterinario WHERE autonomo = 1 ORDER BY id;
```

```

-- control excepción fin del cursor
DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_fin = 1;

-- apertura cursor
OPEN c_veterinarios;

-- recorrido hasta fin
SET num_citas = 0;

veterinarios_autonomos: LOOP
    FETCH c_veterinarios INTO v_id_veterinario;

    -- si hemos terminado
    IF v_fin THEN
        LEAVE veterinarios_autonomos;
    END IF;

    -- obtenemos las citas de ese veterinario
    SELECT COUNT(*) INTO v_num_citas_vet
    FROM atiende
    WHERE id_veterinario = v_id_veterinario AND
        fecha_cita BETWEEN fecha_desde AND fecha_hasta;

    -- acumulamos
    SET num_citas = num_citas + v_num_citas_vet;
END LOOP;

-- cerramos cursor
CLOSE c_veterinarios;
END;

```



Hazlo tú: Crea el procedimiento almacenado `sp_get_citas_autonomos` y comprueba que funciona correctamente. Presta especial atención al modo de crear y gestionar el recorrido del cursor.

10. Permisos en procedimientos y funciones

Sus permisos (privilegios) relacionados son los siguientes. Cuando empleemos el término ROUTINE engloba a procedimientos y funciones:

- **CREATE ROUTINE**: permite su creación.
- **ALTER ROUTINE** y **SYSTEM_USER**: permite su modificación y borrado.
- **EXECUTE**: permite la ejecución.
- **REVOKE**: elimina permisos.

Por ejemplo, vamos a crear un nuevo usuario llamado webclinica@% que inicialmente solo podrá crear y modificar procedimientos almacenados y funciones.

```
CREATE USER webclinica@%' IDENTIFIED BY 'webclinica';  
GRANT CREATE ROUTINE, ALTER ROUTINE ON ud7_clinica_veterinaria.* TO  
webclinica@%';
```

Para eliminar un procedimiento o función necesitamos el permiso ALTER ROUTINE y desde la versión 8 también el de SYSTEM_USER mediante la sentencia:

```
GRANT SYSTEM_USER ON *.* TO webclinica@%';
```

Podemos comprobar los permisos asignados mediante:

```
SHOW GRANTS FOR webclinica@%';
```

Si además queremos darle permisos para ejecutar un procedimiento, por ejemplo:

```
GRANT EXECUTE ON PROCEDURE ud7_clinica_veterinaria.sp_get_citas_autonomos  
TO webclinica@%';
```

Si queremos asignar ese permiso a una función, por ejemplo, la que realizamos en el apartado dedicado a funciones, fn_num_clientes:

```
GRANT EXECUTE ON FUNCTION ud7_clinica_veterinaria.fn_num_clientes  
TO webclinica@%';
```

Para asignar este permiso de ejecución a **todos** los procedimientos y funciones de una base de datos, no es necesario indicar si nos referimos a procedimientos o funciones.

```
GRANT EXECUTE ON ud7_clinica_veterinaria.* TO webclinica@'%';
```

Podemos quitar permisos empleando REVOKE:

```
REVOKE EXECUTE ON PROCEDURE ud7_clinica_veterinaria.sp_get_citas_autonomos
FROM webclinica@'%';
REVOKE EXECUTE ON FUNCTION ud7_clinica_veterinaria.fn_num_clientes
FROM webclinica@'%';
```

Una nota curiosa es que, si borramos un procedimiento o función, sus permisos de ejecución dados a los usuarios **desaparecen**. Al crearlo de nuevo, debemos volver a asignárselos.

En cuanto al permiso de ejecución hay un pequeño detalle que debemos comentar antes de seguir. En MySQL, cuando se crea un procedimiento o función, tenemos la opción de incluir un **DEFINER** que indica una cuenta de usuario. En caso de no indicar ninguno, el valor por defecto es el usuario que crea el objeto. En nuestro caso, hasta ahora, el usuario root.

Por ejemplo, el procedimiento anterior se podría haber declarado también como:

```
CREATE DEFINER = root@'%' PROCEDURE sp_get_citas_autonomos(
...

```

Si le asignamos el permiso de ejecución a un usuario y ejecuta ese procedimiento o función, se ejecutará dentro del llamado “**contexto de seguridad**” del usuario indicado en su DEFINER. Según esto, si el usuario que hace la ejecución tiene menos permisos/privilegios que el usuario creador, podría suceder que realice acciones para las cuales no le hemos dado expresamente permisos. Se puede entender perfectamente con la siguiente actividad.



Hazlo tú: Crea el usuario webclinica como se indica y asígnale el permiso de ejecución a sp_get_citas_autonomos. Crea una conexión nueva en WorkBench con ese usuario, accede a ella, abre la base de datos con la que trabajamos e intenta ejecutar el procedimiento.

Observarás que el procedimiento funciona, y el usuario puede acceder a tablas a las cuales no le hemos dado permisos. Sucede porque está ejecutando el procedimiento dentro del contexto de seguridad de root. Comprueba en el navegador de objetos de Workbench no puede visualizar tablas

Si esta forma de trabajar no nos conviene, siempre podemos modificar el procedimiento o función para que utilice el contexto de seguridad del usuario que hace la ejecución, incluyendo la sentencia **SQL SECURITY INVOKER** al final de la cabecera. Si queremos modificar sp_get_citas_autonomos para que funciona con esta opción:

```
CREATE PROCEDURE sp_get_citas_autonomos(  
    IN fecha_desde DATE,  
    IN fecha_hasta DATE,  
    OUT num_citas INT)  
SQL SECURITY INVOKER  
BEGIN  
    ...
```

Mientras no se indique lo contrario en este curso crearemos los procedimientos y funciones con el usuario root y dejaremos el DEFINER por defecto, el creador.

Hazlo tú: Modifica el procedimiento sp_get_citas_autonomos con la opción SQL SECURITY INVOKER. Comprueba que, sin asignarle permisos extra a webclinica, el procedimiento ya no es operativo. Obtendrás un mensaje de error similar a:



“SELECT command denied to user...”

¿ Por qué sucede esto ? Echa un vistazo al código del procedimiento.

Finalmente, vuelve a dejar el procedimiento como estaba inicialmente.

No elimines el usuario webclinica

11. Triggers

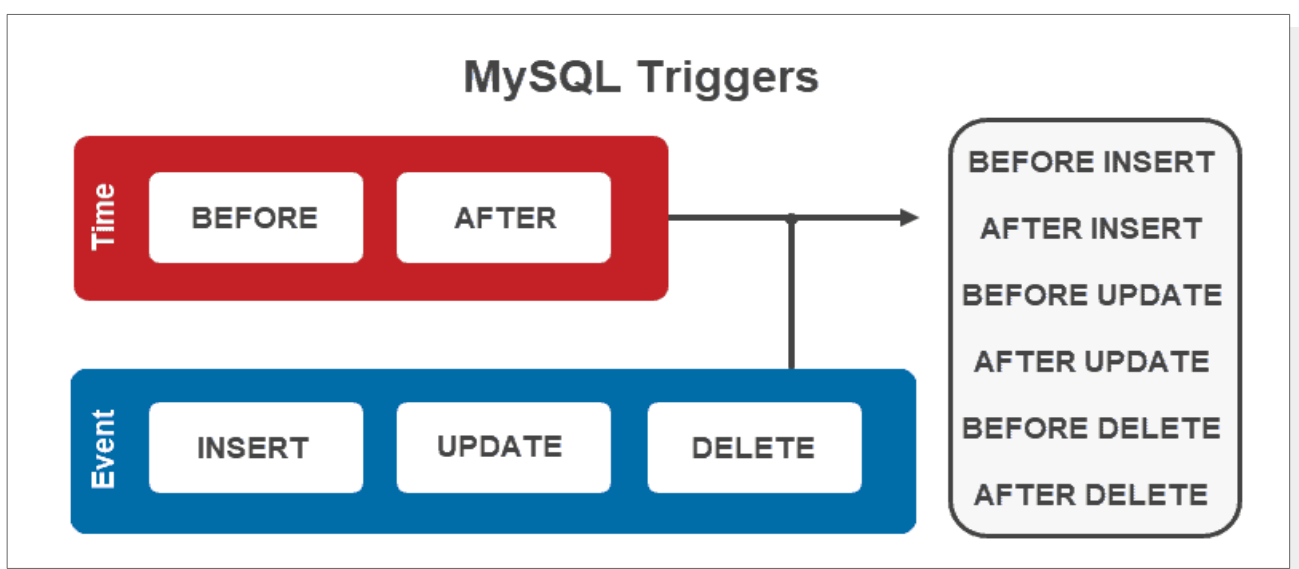
En MySQL un trigger o disparador se define como un conjunto de sentencias SQL que se **invocan automáticamente** en respuesta a un evento del tipo **INSERT**, **UPDATE** o **DELETE** que sucede en una tabla determinada.

El SQL estándar define dos tipos de triggers:

- **De fila:** el trigger se activa para cada fila (registro) que se inserte, actualice o elimine.
- **De instrucción:** el trigger se activa una sola vez por cada evento que lo invoque. Es decir, si una instrucción UPDATE actualiza 10 filas, el trigger se ejecuta una sola vez. Si fuera de nivel de fila se ejecutaría 10 veces.

MySQL solamente soporta triggers a nivel de fila. Otros SGBD como Oracle trabaja con ambos tipos.

Además del evento que lance el trigger, debemos tener en cuenta si queremos ejecutarlo antes (**before**) o después (**after**) del evento en cuestión. Por ejemplo, si lo asociamos al evento INSERT de una tabla, podemos indicar si queremos ejecutarlo antes o después de que la inserción se produzca. Todo esto nos ofrece seis posibilidades:



Dentro del propio código que forman los triggers, podemos acceder a las columnas de la fila que se está insertando, actualizando o borrando. Para ello MySQL crea dos alias **NEW** y **OLD** que podemos usar del siguiente modo:

- **INSERT trigger**: **NEW.columna** indica el valor de la columna que se insertará en la nueva fila. Aquí **OLD** no se permite.
- **DELETE trigger**: **OLD.columna** indica el valor de la columna en la fila que se eliminará. Aquí **NEW** no se permite.
- **UPDATE trigger**: **OLD.columna** y **NEW.columna** se refieren al valor de la columna en la fila antes y después de que la fila se actualice.

Para crear un trigger empleamos la instrucción **CREATE TRIGGER** con la sintaxis:

```
CREATE TRIGGER nombre_trigger momento evento
ON tabla
FOR EACH ROW
BEGIN
    [cuerpo_procedimiento]
END;
```

Eliminamos triggers con la instrucción **DROP TRIGGER** con sintaxis:

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

En cuanto al nombre del trigger, vamos a seguir la siguiente convención: **tr_tabla_momento_evento**.

Podemos consultar todos los triggers de nuestra base de datos con la instrucción:

```
SHOW TRIGGERS;
```

En lo referente al tema permisos, en este curso, solamente dejaremos que el usuario **root** cree, borre y modifique triggers. Además se crearán sin indicar un **DEFINER**, tomando por defecto su creador, esto es, **root**.

Uno de los más habituales usos de los triggers es la **validación de datos** en las tablas, ya que ofrecen más flexibilidad y posibilidades que las CONSTRAINT. Por ejemplo, vamos a crear un trigger sobre la tabla veterinario para que al insertar el campo DNI tenga siempre su letra en mayúsculas, esto es, para que automáticamente la letra del DNI que insertamos se pase a mayúsculas si no lo estuviera.

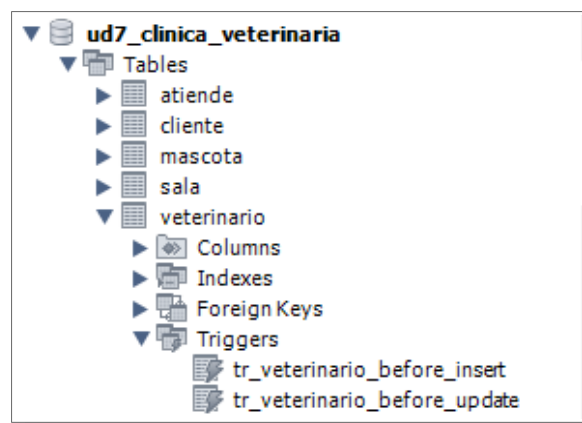
```
CREATE TRIGGER tr_veterinario_before_insert BEFORE INSERT
ON veterinario
FOR EACH ROW
BEGIN
    SET NEW.DNI = UPPER(NEW.DNI);
END;
```

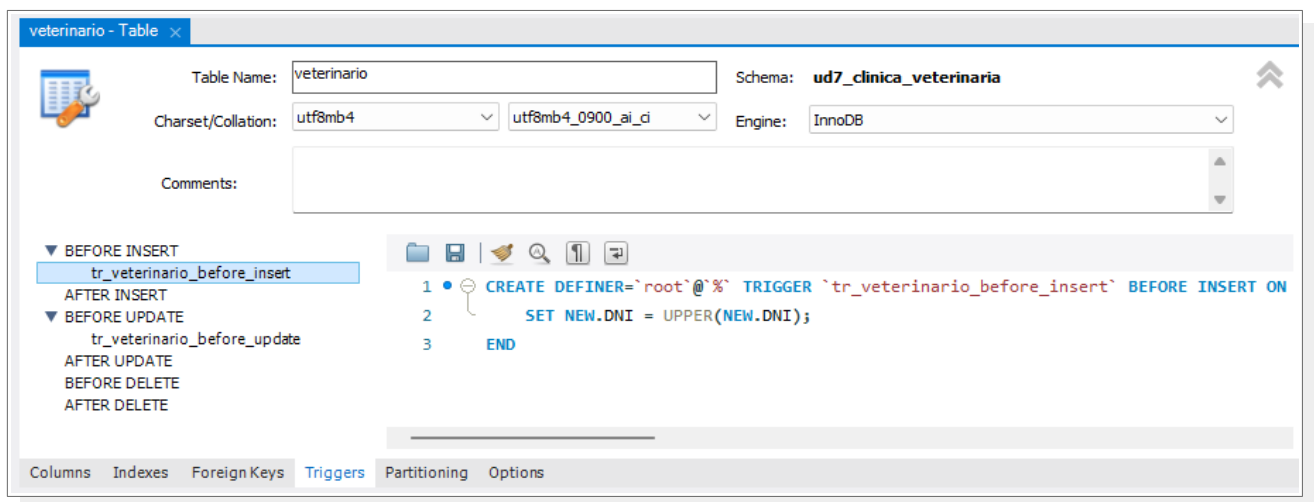
Si el cuerpo solamente lo forma una instrucción SQL no es necesario el emplear el BEGIN...END, sería opcional.

Podemos comprobar su funcionamiento realizando inserciones con DNI que contenga letras en minúsculas y viendo lo que sucede.

```
INSERT INTO veterinario (DNI, nombre, telefono, autonomo, fecha_incorporacion)
VALUES('1a', 'pueba', NULL, 1, '2024-01-06');
SELECT * FROM veterinario;
```

Desde **Workbench** también podemos crear, visualizar y modificar triggers. Deberemos abrir la pestaña que nos permite editar una tabla con el asistente





Hazlo tú: Con el trigger anterior resolvemos el problema de añadir nuevos clientes con DNI que contengan letras en minúsculas, pero ¿qué sucede si modificamos los datos del cliente? Implementa un trigger con nombre `tr_veterinario_before_update` que resuelva ese problema. Comprueba que funciona. Utiliza `SHOW TRIGGERS` para visualizar los triggers creados y analiza la información que te muestra.



Hazlo tú: Dale al usuario `webclinica` que creamos en el apartado anterior, permisos para consultar, insertar, actualizar y borrar en la tabla `VETERINARIO`. Inicia una nueva sesión con ese usuario y comprueba que los triggers creados funcionan haciendo inserciones y actualizaciones sobre el campo `DNI`.

Otro uso muy habitual de los triggers es el dejar constancia en tablas o campos de los **cambios** que se producen en la base de datos, como almacenar un histórico con todas las modificaciones realizadas en las filas de una tabla.

En el siguiente ejemplo creamos un trigger que cada vez que se modifica un registro de la tabla `veterinario`, añada en una nueva tabla histórico el cambio realizado. Lógicamente debemos crear previamente esa nueva tabla. Es interesante, incluir campos que nos puedan ser útiles para hacer un seguimiento de la evolución de la información.

```

DROP TABLE IF EXISTS veterinario_history;
CREATE TABLE veterinario_history (
    id INT AUTO_INCREMENT PRIMARY KEY,
    id_veterinario INT,
    DNI VARCHAR(10),
    nombre VARCHAR(50),
    telefono VARCHAR(50),
    autonomo TINYINT,
    fecha_incorporacion DATE,
    log_cambio VARCHAR(10),
    log_ultima_modificacion DATETIME DEFAULT NOW()
);

```

El campo `log_cambio` contendrá un texto que nos indique el tipo de operación realizada. El campo `log_ultima_modificacion` almacenará el momento en que se realiza la operación.

El trigger quedaría como:

```

CREATE TRIGGER tr_veterinario_after_update AFTER UPDATE
ON veterinario
FOR EACH ROW
BEGIN
    INSERT INTO veterinario_history
        (id_veterinario, DNI, nombre, telefono, autonomo, fecha_incorporacion,
        log_cambio)
    VALUES
        (NEW.id, NEW.DNI, NEW.nombre, NEW.telefono, NEW.autonomo,
        NEW.fecha_incorporacion, 'UPDATE');
END;

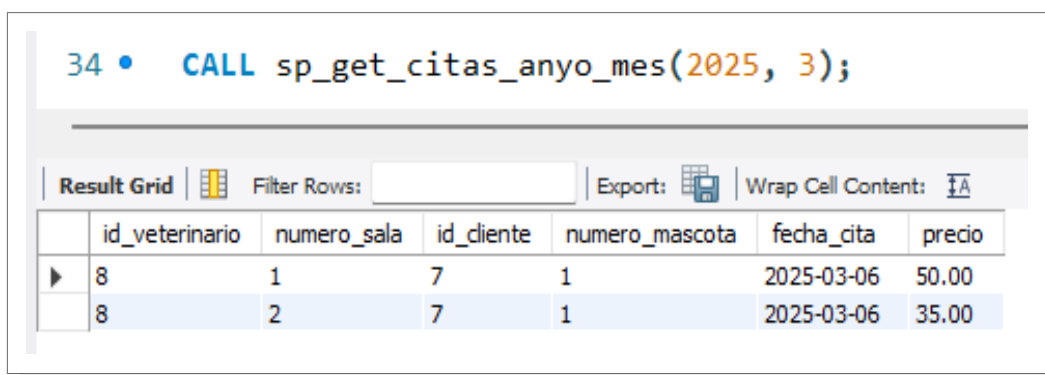
```



Hazlo tú: Crea la tabla `veterinario_history` y el trigger `tr_veterinario_after_update`. Visualiza el código del trigger desde Workbench. Comprueba que funciona correctamente haciendo actualizaciones de diferentes campos y visualizando el contenido de la tabla de histórico.

12. Exportación de datos en JSON

En los procedimientos almacenados que hemos realizado anteriormente, el modo que teníamos para extraer información de la base de datos, además de emplear parámetros de salida, era realizar una instrucción `SELECT` y su resultado, lo que llamamos un **resultset**, sería lo que el programa que hace la llamada recuperaría y procesaría. Por ejemplo, la siguiente llamada al procedimiento que creamos en apartados anteriores, nos devolverá un **resultset** con los datos de las citas para ese año y mes:



The screenshot shows a SQL client interface. At the top, a command line displays the call to a stored procedure: `34 • CALL sp_get_citas_anyo_mes(2025, 3);`. Below the command line, there is a toolbar with options like 'Result Grid', 'Filter Rows', 'Export', and 'Wrap Cell Content'. The main area displays a table with the following data:

	id_veterinario	numero_sala	id_cliente	numero_mascota	fecha_cita	precio
▶	8	1	7	1	2025-03-06	50.00
	8	2	7	1	2025-03-06	35.00

Podríamos crear procedimientos que devolvieran varios **resultset** empleando varias instrucciones `SELECT`, aunque la manera habitual de trabajar es seguir la norma de solo un **resultset** por procedimiento.

El programa que hace la llamada al procedimiento deberá **recuperar esos datos** en forma de tabla e incorporarlos a algún tipo de estructura que su lenguaje de programación le permita manejar ese conjunto de filas y campos. En este punto es cuando el formato **JSON** (JavaScript Object Notacion) nos puede ser muy útil.

JSON es un formato de intercambio de información basado en texto muy utilizado para obtener datos de un servidor web y mostrarlos en una página web.

MySQL ofrece desde la versión 5.7.8 un conjunto de funciones que nos permiten trabajar con el formato JSON. En este curso vamos a ver solamente algunas funciones básicas.

Como la mejor forma de introducir contenidos suele ser mediante ejemplos, vamos a crear un procedimiento llamado `sp_get_citas_entre_dias_JSON` similar al anterior que hemos visto, pero devolverá un resultado en formato JSON. Se añade una comprobación en caso de no existir citas esos días.

```
CREATE PROCEDURE sp_get_citas_entre_dias_JSON(
    IN inicio DATE,
    IN fin DATE)
BEGIN
    -- Devuelve citas entre un intervalo de fechas
    SET @v_json_citas = json_object();

    -- cada fila es un objeto JSON, todas forman un array
    SELECT JSON_ARRAYAGG(
        JSON_OBJECT (
            'id_veterinario', id_veterinario,
            'numero_sala', numero_sala,
            'id_cliente', id_cliente,
            'numero_mascota', numero_mascota,
            'fecha_cita', fecha_cita,
            'precio', precio
        )
    ) AS resultado INTO @v_json_citas
    FROM atiende
    WHERE fecha_cita BETWEEN inicio AND fin;

    -- si no hay citas en ese intervalo
    IF @v_json_citas IS NULL THEN
        SELECT json_object("error", "sin citas") as resultado;
    ELSE
        SELECT @v_json_citas AS resultado;
    END IF;
END;
```

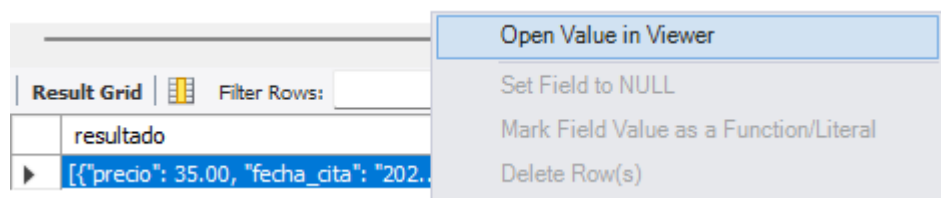
Como se puede comprobar empleamos dos funciones anidadas. La más interna, **JSON_OBJECT**, nos convierte cada fila con los campos indicados en un objeto básico JSON. La más exterior, **JSON_ARRAYAGG**, nos agrupa todos los objetos en un único array JSON. El resultado lo almacenamos en una variable de tipo **JSON_OBJECT**. El empleo de la función **JSON_ARRAYAGG** hace que si la **SELECT** no devolviera nada el JSON resultado tome el valor **NULL**, en ese caso devolvemos un JSON con el error, en caso contrario el array creado.

Si estuviéramos haciendo una consulta que solo devuelve una fila, no sería necesario emplear la función **JSON_ARRAYAGG**, es suficiente emplear **JSON_OBJECT**. Esto es por propia definición del formato JSON. Para comprobar que la función **JSON_OBJECT** ha devuelto un elemento no vacío deberemos emplear la función **JSON_LENGTH** que devuelve el número de elementos que contiene, si es cero, la consulta no ha devuelto nada.



Hazlo tú: Crea el procedimiento **sp_get_citas_entre_dias_JSON** y comprueba que funciona correctamente, incluyendo el caso de no encontrar citas para el intervalo recibido.

Si creamos y ejecutamos el procedimiento anterior veremos que el resultado lo podemos visualizar cómodamente en Workbench seleccionando con el botón derecho “Open Value in Viewer”



El visor nos ofrece varias opciones para mostrar los datos JSON.



Hazlo tú: Crea el procedimiento `sp_get_citas_ano_mes_JSON`, basándote en el procedimiento realizado anteriormente `sp_get_citas_ano_mes`, pero devolverá un objeto JSON con los resultados. Devolverá un error si no hay citas en el mes y año recibido.



Hazlo tú: Crea un procedimiento que reciba al menos un parámetro de entrada y que nos permita filtrar una de las tablas de nuestra base de datos. El resultado se devolverá en formato JSON. Debes contemplar el caso de no encontrar filas que cumplan el filtro.