

Construcción de guiones.

Caso práctico

Con el trabajo desarrollado por el equipo de **Vindio, Noiba y Naroba**, en **BK Sistemas Informáticos** ya disponen de una **base de datos** que integra toda la información que necesitaba el cliente para llevar la **gestión del taller mecánico**. Atrás han quedado los archivos de facturas en papel, las fichas manuales con los datos de nuestros clientes, los listados para consultar los recambios y sus precios, etc. El trabajo se ha simplificado enormemente, el acceso a la información es rápido y los socios están muy satisfechos con su trabajo. Pero **Juan** les recuerda que aún le pueden sacar más partido a la base de datos si incorporan al lenguaje **SQL** algunas características que, en general, están disponibles en cualquier lenguaje de programación y que permiten a los usuarios autorizados escribir bloques de sentencias **SQL**, y guardarlos en el servidor como cualquier otro objeto de la base de datos para utilizarlos cuando lo necesiten.



[Jonny Goldstein \(CC BY\)](#)

¿Es eso posible con SQL? Sí, la mayoría de los **SGBD** permiten que los usuarios creen sus propios procedimientos y funciones mediante una extensión del lenguaje **SQL**.

¿Cómo podrían aplicar esto en el taller? La creación de un procedimiento guardado podría servir, por ejemplo, para recoger todas las sentencias que tenemos que realizar cada vez que un nuevo cliente nos trae su vehículo para que solucionemos una avería. Si podemos hacer esto de forma automática sin más que activar el procedimiento nos ahorraremos trabajo y posibles errores, ¿no crees? Vamos a ello.

Ahora que ya dominas el uso de **SQL** para la manipulación y consulta de datos, es el momento de dar una vuelta de tuerca adicional para mejorar las aplicaciones que utilicen nuestra base de datos. Para ello nos vamos a centrar en la programación de bases de datos o construcción de guiones, utilizando el lenguaje **SQL/PSM (Persistent Stored Modules)**. En esta unidad conoceremos qué es **SQL/PSM**, cuál es su sintaxis y veremos cómo podemos sacarle el máximo partido a una base de datos mediante su uso.

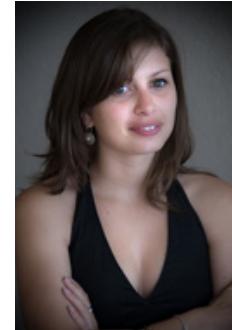
Aunque en un principio **SQL** fue un lenguaje dirigido exclusivamente a la manipulación interactiva de conjuntos de datos, ejecutando sentencias individuales más o menos complejas como las que hemos visto en unidades anteriores, con el tiempo quedaron patentes los beneficios que aportaría la inclusión en los **SGBDR** de toda la lógica relacionada con el tratamiento de datos, incluyendo procesos más complejos que una simple inserción, actualización o consulta.

Por ello, casi todos los grandes Sistemas Gestores de Base de Datos incorporan utilidades que permiten ampliar el lenguaje **SQL** para producir pequeñas utilidades que añaden al **SQL** mejoras de la programación estructurada (bucles, condiciones, funciones, etc.)

1.- Introducción. Lenguaje de programación.

Caso práctico

Noiba, Vindio y Naroba se plantean la necesidad de añadir funcionalidades nuevas a su aplicación. Para eso van a tener que pasar de un lenguaje de consultas a un **lenguaje de procedimientos**. Creen que diseñar sus propios guiones de sentencias y hacer que se ejecuten cuando lo necesiten les va a permitir mejorar en el control y mantenimiento de los datos. Para dar este paso, tienen que empezar por revisar cómo se pueden crear esas rutinas o subprogramas con MySQL.



Alain Bachelier (CC BY-NC-SA)

Hasta ahora hemos visto que el **SQL** es un lenguaje que permite consultar, añadir, modificar y eliminar datos en las tablas de la base de datos, pero existen otras funciones propias de un lenguaje de programación que no hemos visto hasta ahora.

Algunos de los procesos que llevamos a cabo habitualmente, como es el caso del registro de una reparación en el taller, implican que se ejecuten varias sentencias, unas detrás de otras. Sería muy interesante que este conjunto de sentencias estuvieran recogidas en una rutina, y que se pudieran guardar y ejecutar cuando lo necesitemos, sin necesidad de enviar las consultas de una en una.

Tampoco podíamos, hasta ahora, establecer condiciones para que se lleven a cabo o no unas determinadas acciones o hacer que estas se repitan. Si queremos que se muestre un listado para hacer una compra a los proveedores, cada vez que el stock de los recambios baje por debajo de una cantidad que consideramos como el stock mínimo, y que este proceso se ejecute automáticamente, necesitaríamos establecer condiciones, algo que no hemos visto con SQL.

Cuando tengamos que realizar operaciones que impliquen la ejecución condicional, así como operaciones repetitivas, necesitaremos un lenguaje de programación de procedimientos como Java, Perl, Php, Visual Basic, etc. Podemos insertar sentencias SQL dentro de estos lenguajes que actuarían como Lenguaje anfitrión. Este método es muy común pero no el único.

Con la aparición de las bases de datos distribuidas y las bases de datos orientadas a objetos se necesita que ese código se pueda guardar en la base de datos y sea compartido por las aplicaciones que accedan a los datos, porque así se mejora el mantenimiento y el control lógico. Por este motivo actualmente la mayoría de los SGBD han incluido extensiones de programación que incluyen:

Estructuras de control como: IF-THEN-ELSE y DO-WHILE.

Declaración de variables y utilización dentro de los procedimientos.

Manejo de errores.

El SQL de procedimientos permite utilizar código de procedimientos y sentencias SQL que se guardan dentro de la base de datos y se ejecutan cuando el usuario los invoca.

En el caso de MySQL la extensión de SQL que permite utilizar código y crear rutinas se denomina **SQL /PSM**, y simplemente es un lenguaje procedural estructurado en bloques que amplía la funcionalidad de SQL. Con SQL /PSM podemos usar sentencias SQL para manipular datos y sentencias de control de flujo para procesar los datos. Por tanto, SQL /PSM combina la potencia de SQL para la manipulación de datos, con la potencia de los lenguajes procedimentales para procesar los datos.

El usuario puede crear:

- Procedimientos guardados.
- Funciones.
- Disparadores o triggers.

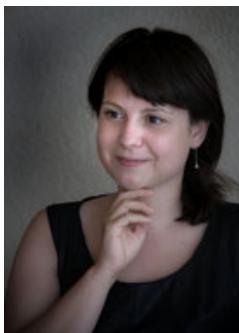
Veremos en los siguientes apartados qué son y para qué sirven.

Reflexiona

En esta unidad vamos a crear procedimientos, funciones y disparadores, utilizando estructuras alternativas y estructuras repetitivas dentro de guiones escritos en SQL. La posibilidad de incrustar lenguaje SQL dentro de un lenguaje de programación como es el caso MySQL con PHP, es muy utilizada en la creación de páginas Web dinámicas. En el módulo de Aplicaciones Web se estudia esta alternativa.

2.- Estructuras funcionales.

Caso práctico



[Alain Bachellier \(CC BY-NC-SA\)](#)

Naroba se dispone a estudiar junto a **Noiba** las **nuevas posibilidades** que se abren al analizar cómo puede crear sus propios procedimientos y funciones, además de almacenarlos en TalleresFaber y asociar disparadores o **triggers** a las tablas.

Con el resto de los empleados del taller que acceden a la base de datos estudia qué procesos se componen de varias sentencias y se utilizan frecuentemente, para poder crear estas estructuras como **procedimientos almacenados**; por ejemplo registrar una nueva reparación, o algunas operaciones habituales con las que sería interesante crear **funciones** y simplemente utilizarlas en nuestras consultas, como podría ocurrir con el cálculo del coste de la mano de obra empleada en una reparación. También que algunas operaciones se realicen automáticamente creando disparadores o **triggers**, caso de comprobar la existencia de recambios en el almacén. Para ver si esto es posible, estudiaremos las estructuras funcionales que maneja nuestro SGBD.

Empezamos por definir algunos conceptos:

Módulos: Se denominan así a las partes en que se puede dividir un programa para reducir su complejidad, que son independientes de la aplicación en sí y de las restantes partes del programa, aunque tienen conexiones con otros módulos. También se les denomina rutinas o **subrutinas**. En bases de datos las rutinas pueden ser procedimientos, funciones o disparadores (triggers).

Procedimientos: Se trata de un conjunto de instrucciones **SQL** que se guardan en el servidor y se ejecutan cuando ese procedimiento es llamado.

Funciones: Son subrutinas que devuelven un valor.

Triggers: Es un procedimiento que va asociado a una tabla y se ejecuta automáticamente cuando se cumple una determinada condición o evento en esa tabla.

Los triggers pueden utilizarse para sentencias **INSERT**, **UPDATE** y **DELETE**. Se utilizan para prevenir errores y mejorar la administración de la base de datos sin necesidad de que el usuario ejecute la sentencia SQL, ya que se ejecuta automáticamente en función de que se cumpla una determinada condición.

En general las funciones y los procedimientos no son muy diferentes. Ambos están constituidos por un conjunto de sentencias lógicamente agrupadas para realizar una tarea específica. Se trata de un bloque de código que se almacena en tablas del sistema de la base de datos.

Ventajas del uso de rutinas.

Las ventajas que presenta el uso de rutinas almacenadas son:

Seguridad. Se pueden establecer autorizaciones variadas de ejecución para los usuarios sobre los procedimientos y funciones que se deseen, con lo que se puede **gestionar la seguridad**. Proporcionan un único punto de acceso a la base de datos lo que permite controlar más fácilmente qué operaciones se realizan, quienes pueden utilizarlas y en qué momento.

Reutilización. Es posible **disponer de librerías de procedimientos y funciones** almacenadas en el servidor, que pueden ser invocadas desde otros lenguajes de programación o ser reutilizados para construir otras rutinas.

Mejoran el rendimiento, ya que están almacenados en la base de datos, y ello supone que se necesita enviar menos información entre el servidor y el cliente.

Se simplifican las tareas cotidianas.

Son útiles cuando queremos realizar una misma operación que afecte a varias bases de datos y tablas de un mismo servidor.

Un procedimiento almacenado en el servidor ayuda a mantener la **consistencia y la integridad** de los datos, ya que evita que éstos puedan ser corrompidos por el acceso de programas defectuosos.

Permiten la validación de datos, y se integran en la estructura de la base de datos. Cuando funcionan con este propósito se denominan *triggers*.

Son dinámicos ya que admiten parámetros que se pasan antes de su ejecución y puedan realizar diferentes tareas dependiendo de esos parámetros que se hayan pasado.

La diferencia más importante entre los procedimientos y las funciones es que una función, al final de su ejecución, devuelve un valor; sin embargo, en los procedimientos esto no es posible, aunque sí que podemos definir múltiples parámetros de salida. Esto último también es posible en las funciones.

Autoevaluación

Relaciona los siguientes conceptos con el término adecuado:

Ejercicio de relacionar

Términos.	Relación.	Conceptos.
Procedimiento.	0	1. Conjunto de instrucciones que se ejecutan automáticamente cuando se cumple una condición.
Función.	0	2. Conjunto de instrucciones que se ejecutan cuando son invocados.
Trigger.	0	3. Conjunto de instrucciones que cuando se ejecutan devuelven un valor.

Enviar

Las rutinas o módulos pueden ser procedimientos, funciones o triggers.

2.1.- Herramientas gráficas para procedimientos y funciones.

En todas las herramientas gráficas (MySQL Workbench, PhpMyAdmin, Navicat, SQLMaestro, etc.) que te hemos presentado hasta ahora podemos encontrar recursos para crear, modificar y/o eliminar procedimientos y funciones en modo gráfico.

Veamos cómo crear crear en modo gráfico estas rutinas con MySQL Workbench.



Everaldo Coelho (YellowIcon)
(GNU/GPL)

◀ 1 2 3 4 5 6 7 8 9 ▶

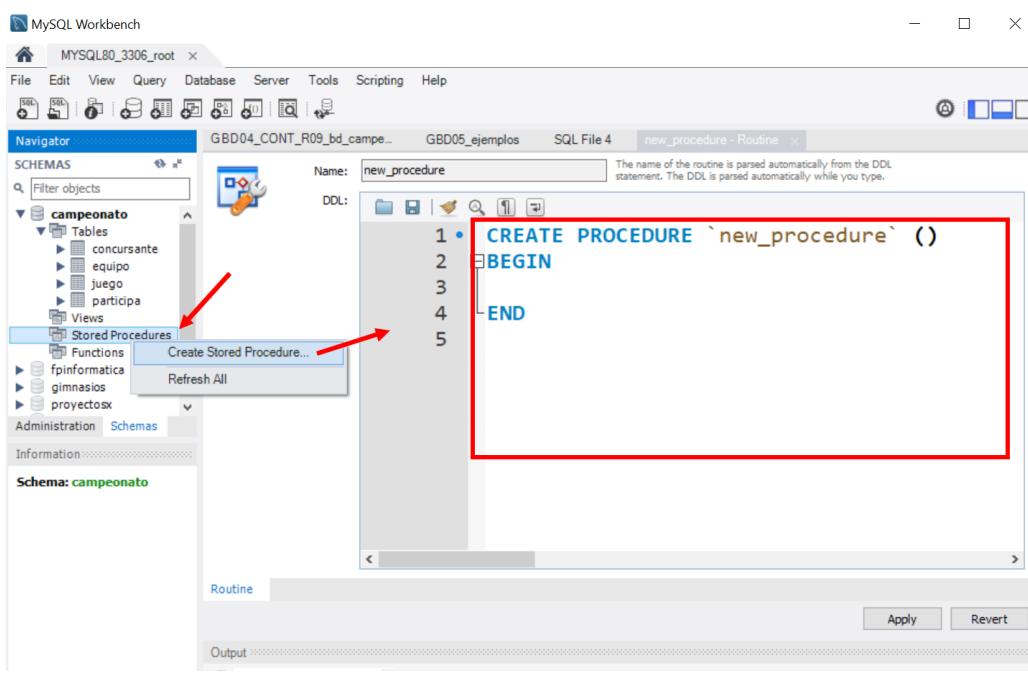
1.- Crear procedimiento .

Realiza con tu usuario una conexión con Workbench a MySQL y pon en uso o activa la base de datos campeonato para trabajar con ella.

En la sección izquierda aparecen contenedores para organizar de forma gráfica diferentes objetos de la base de datos: tablas, procedimientos almacenados, funciones, vistas, ...

Haz clic derecho sobre el contenedor de procedimientos almacenados y después pulsa en la opción "Crear procedimiento almacenado".

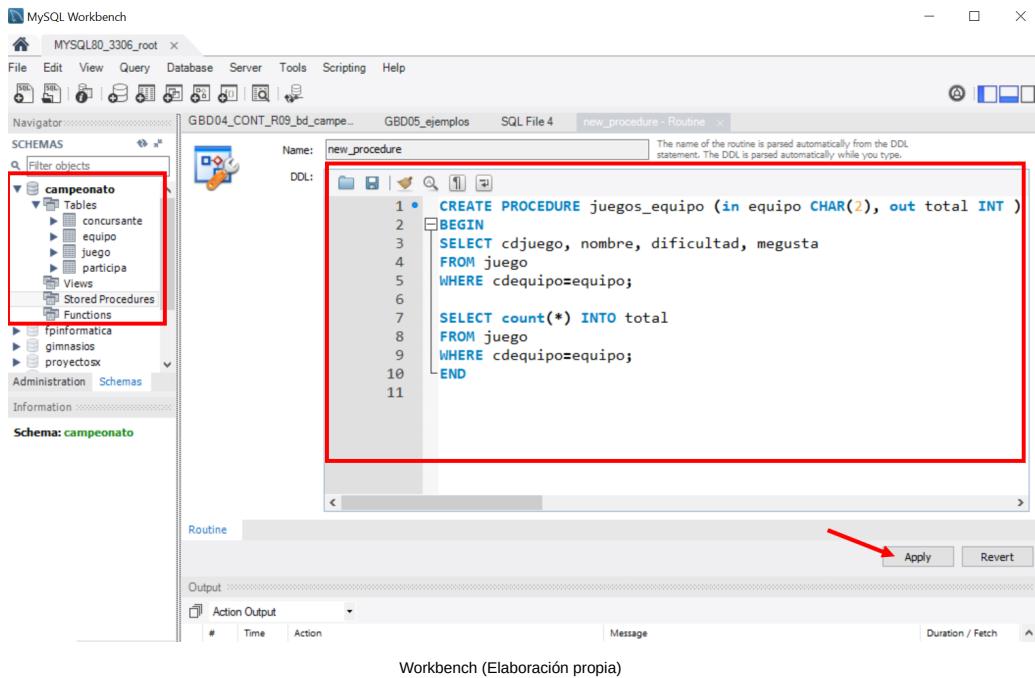
En esa estructura básica que te aparece en la ventana derecha debes incluir el código del procedimiento.



Workbench (Elaboración propia)

2.- Crear procedimiento.

Un vez que hayas escrito el código apropiado para el procedimiento debes pulsar en aplicar.

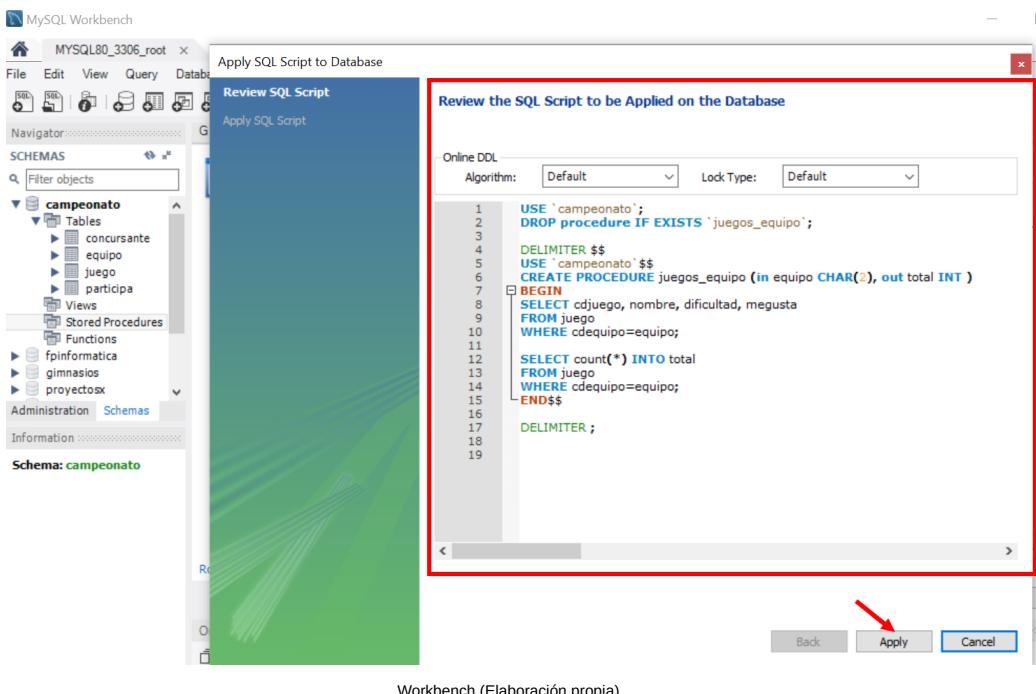


3.- Crear procedimiento.

Aparece en una nueva ventana el código generado por MySQL. Ese será el formato que seguiremos nosotros para escribir mediante SQL/PSM los procedimientos almacenados.

Pulsamos aplicar y, si todo es correcto y se ejecuta sin errores, nos permite finalizar. El procedimiento ya se ha creado.

Si al pulsar el botón de aplicar se producen errores, habrá que corregirlos y volver a intentarlo.



Workbench (Elaboración propia)

4.- Ejecutar procedimiento.

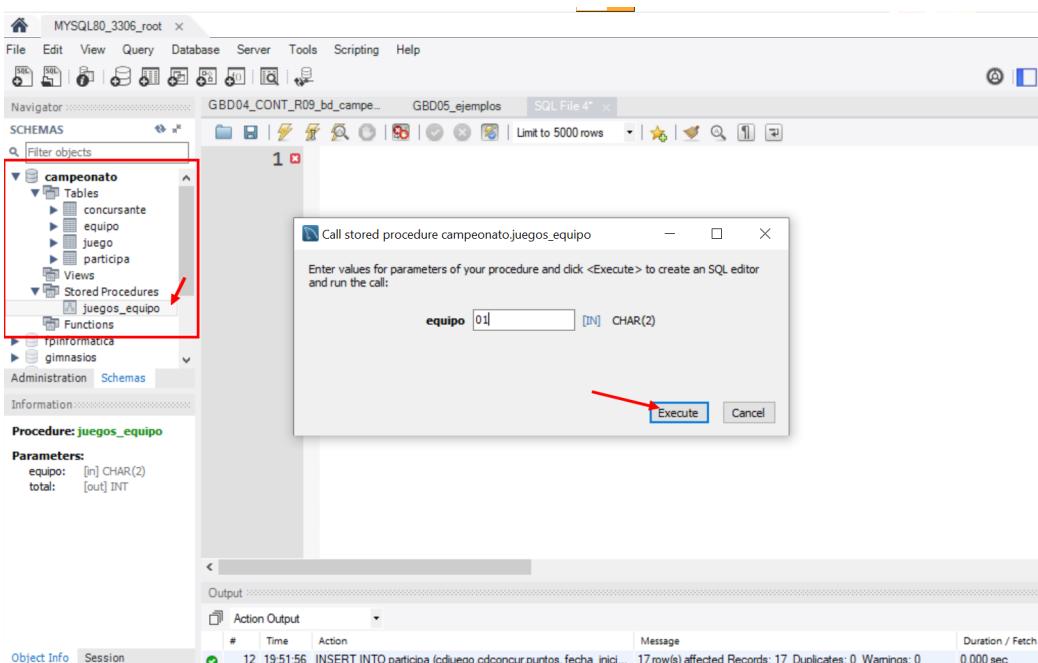
Una vez creado el procedimiento, este se habrá almacenado en la base de datos. Realmente se almacena en una de las tablas del sistema, pero la herramienta Workbench organiza los objetos de la base de datos en esos contenedores para facilitar su tratamiento visual o gráfico.

Para ejecutarlo:

Pulsa sobre el rayo que aparece sobre el nombre del procedimiento y mostrará el formulario de la derecha, pues ese procedimiento requiere un parámetro de entrada.

Se introduce el valor de un código de equipo, el '01'.

Y pulsas Ejecutar.



5.- Ejecutar procedimiento.

El resultado de la ejecución del procedimiento en modo gráfico muestra dos cosas:

En la sección superior se ven las sentencias SQL que ha generado Workbench y ha enviado al servidor MySQL para ejecutarlo.

En la sección inferior se ve el resultado de la ejecución.

```

1 • set @total = 0;
2 • call campeonato.juegos_equipo('01', @total);
3 • select @total;
4

```

cdjuego	nombre	dificultad	megusta
ELV	Elvenar	baja	3
GOE	Goodgame Empire	alta	NULL
GW2	Guild Wars 2.	baja	5
VIK	Vikings	media	5

Result 1 Result 2

6.- Crear función.

En la sección izquierda aparecen contenedores para organizar de forma gráfica diferentes objetos de la base de datos: tablas, procedimientos almacenados, funciones, vistas, ...

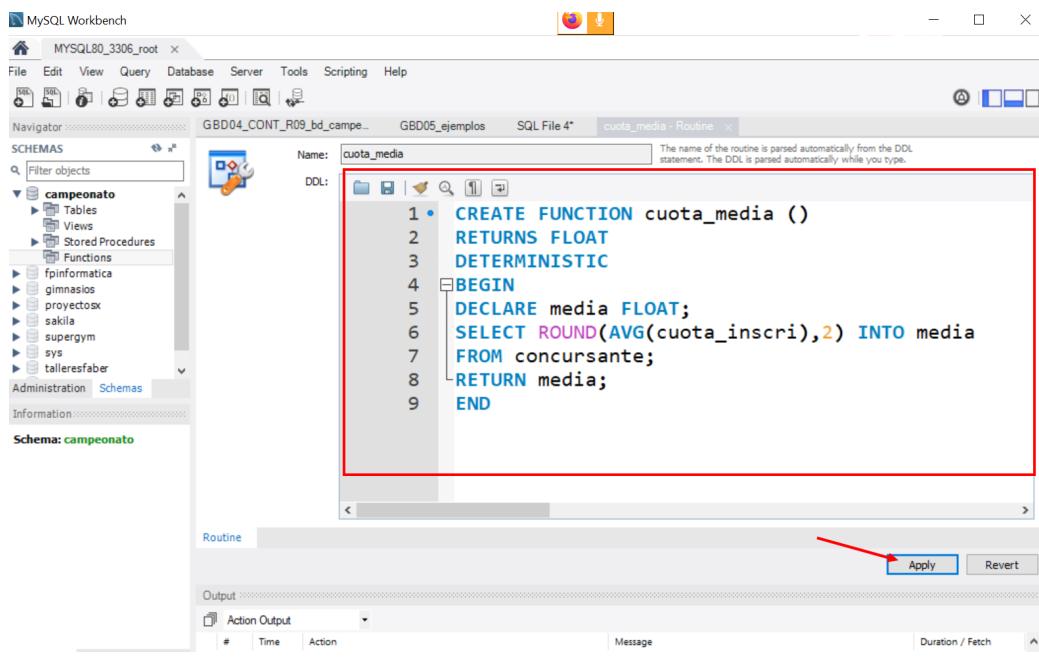
Haz clic derecho sobre el contenedor de funciones almacenados y después pulsas en la opción "Crear función".

En esa estructura básica que te aparece en la ventana derecha debes incluir el código de la función.

Workbench (Elaboración propia)

7.- Crear función.

Un vez que hayas escrito el código apropiado para la función debes pulsar en aplicar.



The screenshot shows the MySQL Workbench interface. The left sidebar displays the 'campeonato' schema with its tables, views, stored procedures, and functions. The main area is titled 'cuota_media - Routine'. A red box highlights the SQL code in the DDL pane:

```
1 • CREATE FUNCTION cuota_media ()  
2 RETURNS FLOAT  
3 DETERMINISTIC  
4 BEGIN  
5     DECLARE media FLOAT;  
6     SELECT ROUND(AVG(cuota_inscri),2) INTO media  
7     FROM concursante;  
8     RETURN media;  
9 END
```

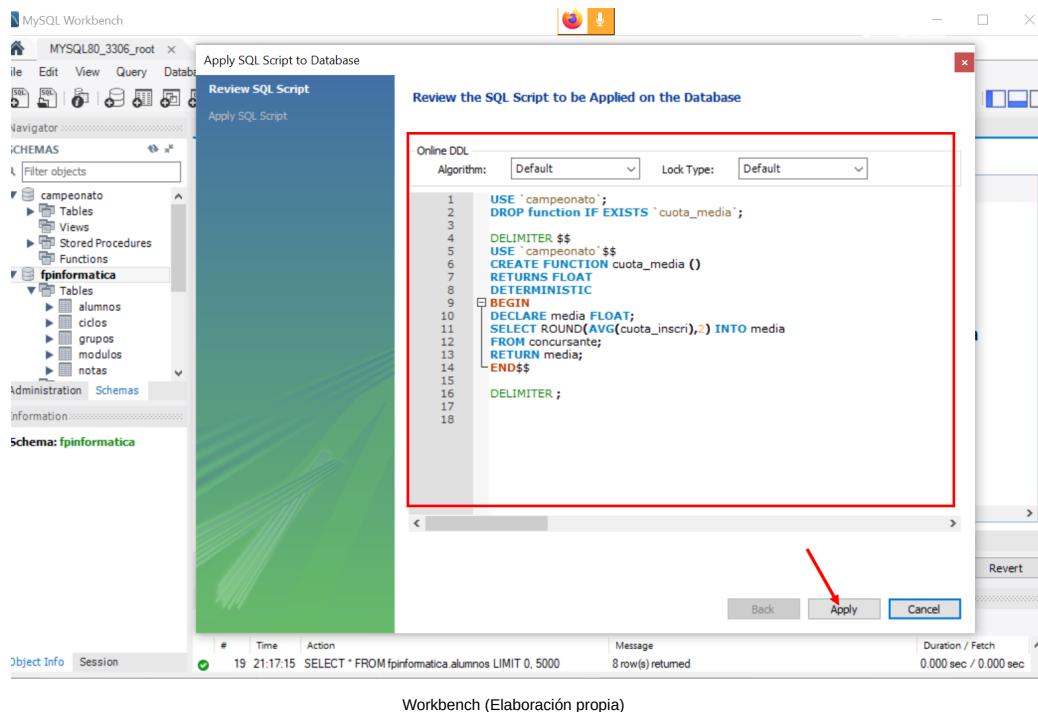
An arrow points from the bottom right of the red box to the 'Apply' button at the bottom of the DDL pane.

Workbench (Elaboración propia)

8.- Crear función.

Aparece en una nueva ventana el código generado por MySQL. Ese será el formato que seguiremos nosotros para escribir mediante SQL/PSM las funciones almacenadas.

Pulsamos aplicar y, si todo es correcto y se ejecuta sin errores, nos permite finalizar. La función ya se ha creado.
Si al pulsar el botón de aplicar se producen errores, habrá que corregirlos y volver a intentarlo.



9.- Ejecutar función.

Una vez creada la función, éste se habrá almacenado en la base de datos. Realmente se almacena en una de las tablas del sistema, pero la herramienta Workbench organiza los objetos de la base de datos en esos contenedores para facilitar su tratamiento visual o gráfico.

Para ejecutarla:

Pulsa sobre el rayo que aparece sobre el nombre de la función y mostrará el resultado directamente, pues en este caso no hay que suministrar datos para que se ejecute.

MySQL Workbench

File Edit View Query Database Server Tools Scripting Help

Navigator GBD04_CONT_R09_bd_camp... GBD05_ejemplos SQL File 5 cuota_media x

SCHEMAS Filter objects campeonato Tables Views Stored Procedures Functions cuota_media f() fpinformatica Tables Views Stored Procedures Functions gimnasios Administration Schemas

Schema: campeonato

Result Grid Filter Rows: Export: Wrap Cell Content: Result 1 x Output: Action Output # Time Action Message Duration / Fetch

Object Info Session 20 21:20:19 Apply changes to cuota_media Changes applied Read Only

The screenshot shows the MySQL Workbench interface. In the top-left, the schema 'campeonato' is selected. In the central query editor, the following SQL code is run:

```
1 • select campeonato.cuota_media();  
2
```

The result grid displays the output of the function call:

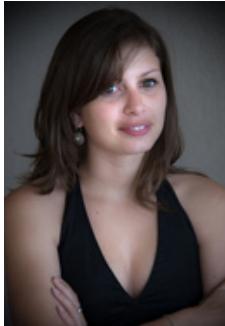
campeonato.cuota_media()
66.31999969482422

The entire code block and the result grid are highlighted with red boxes. A red arrow points from the left margin towards the 'cuota_media' entry in the 'Functions' section of the Navigator pane.

Workbench (Elaboración propia.)

3.- Variables de usuario.

Caso práctico



Para diseñar procedimientos, funciones o triggers, **Noiba** tiene que empezar por revisar los elementos que estas rutinas pueden incluir. En unidades anteriores ha tratado los tipos de datos que emplea MySQL y algunos aspectos relativos a la estructura del lenguaje, como: tratamiento de palabras reservadas, valores literales, escritura de comentarios, nombres de objetos, etc. Es necesario tener presente todos estos temas, pero ahora tiene que detenerse en algunos elementos que forman parte de la estructura del lenguaje, que no ha tratado hasta ahora: las **variables** y las **estructuras de control**. En este apartado, vamos a seguirla en el estudio de las reglas que hay que cumplir para definir y utilizar variables en un guión.

Alain Bachellier (CC BY-NC-SA)

Variables de usuario o de sesión.

Una variable de usuario, también denominada variable de sesión, permite almacenar un valor y referirnos a él más tarde. También pueden pasarse de una sentencia a otra. Las variables 'de usuario' no pueden ser vistas por otros usuarios y desaparecen cuando la conexión se cierra. Para **crear** una variable de usuario:

SET @NombreVariable1 = Expresión [, @NombreVariable2 = Expresión] ...

Podemos utilizar como **operador de asignación** tanto el signo = como el signo := cuando las definimos con **SET**. Si la variable recibe su valor de otras sentencias el operador debe ser := para diferenciarlo del **operador de comparación**.

Para utilizar una variable de usuario:

Podemos utilizar las variables de usuario en cualquier sitio donde se puedan usar expresiones, siempre y cuando no sea necesario que sea un valor literal. Por ejemplo, la cláusula LIMIT en una SELECT tiene que contener un valor literal que refleje el número de filas devueltas por la consulta.

No se debe asignar un valor a una variable de usuario en una parte de una sentencia y usarla en otra parte de la misma sentencia. Por ejemplo: no se debe asignar un valor a una variable en SELECT y hacer referencia a ella en HAVING, GROUP BY u ORDER BY. Puede dar resultados inesperados.

Si hacemos referencia a una variable sin inicializar con ningún valor, su valor es NULL y de tipo cadena.

Autoevaluación

Queremos obtener un listado con dos columnas: **Marcas de vehículos y el número de vehículos de cada marca que nos ha visitado. El listado estará ordenado por el número de vehículos.** ¿Cuál de las siguientes consultas nos muestra el listado correcto?

- SELECT Marca, @NUMERO:=count(Matricula) AS 'Número de vehículos por marca'
FROM VEHICULOS GROUP BY marca ORDER BY @NUMERO;
- SELECT Marca, @NUMERO:=count(Matricula) AS 'Número de vehículos por marca'
FROM VEHICULOS GROUP BY marca ORDER BY 'Número de vehículos por marca';

- SELECT Marca, @NUMERO:=count(Matricula) AS 'Número de vehículos por marca'
FROM VEHICULOS GROUP BY marca ORDER BY count(Matricula);
- SELECT Marca, @NUMERO:=count(Matricula) AS 'Número de vehículos por marca'
FROM VEHICULOS GROUP BY marca ORDER BY Marca;

Incorrecto. No puede definirse una variable y usarla en otra parte de la misma sentencia.

La ordenación es incorrecta. El alias no puede utilizarse en la cláusula ORDER BY.

Correcto. El listado se muestra ordenado correctamente.

No es la respuesta correcta. El criterio de ordenación no es el solicitado.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta
4. Incorrecto

Para saber más

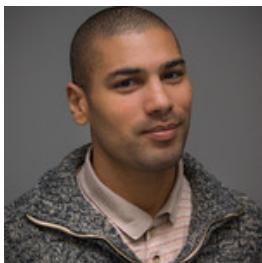
Aunque ya hemos tratado en unidades anteriores el resto de elementos que forman parte de la estructura del lenguaje SQL en MySQL, es aconsejable que los recuerdes como paso previo a la escritura de procedimientos y funciones.

Para recordar las reglas sobre cómo escribir: **literales, nombres de objetos** de la base de datos, **comentarios, variables de usuario** y tratamiento de **palabras reservadas**, puedes acceder al manual de MySQL en el siguiente enlace:

[Reglas para escribir sentencias en SQL](#).

4.- Procedimientos.

Caso práctico



[Alain Bachellier \(CC BY-NC-SA\)](#)

Vindio le comenta a **Noiba** y **Naroba** que considera muy interesante poder **almacenar** en el servidor de la base de datos TalleresFaber algunos **bloques de instrucciones que se repiten frecuentemente**. El cree que esto va a resultar, en primer lugar, más rápido porque no habrá que enviar las consultas de una en una al servidor, y en segundo lugar más seguro. Si el código es correcto, no tendrán que estar pendientes de posibles errores.

Por otra parte, todas las personas que acceden a la base de datos con autorización podrán ejecutar estos procedimientos, aunque se trate de tareas complejas.

Parece interesante ¿no crees?

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

Analizaremos en primer lugar, las ventajas de utilizar procedimientos:

Mejoran el rendimiento en la comunicación entre el cliente y el servidor. Dado que un procedimiento consta de varias instrucciones y está almacenado en el servidor, desde el cliente MySQL (Workbench o cliente en modo texto) no hay que enviar todas esas instrucciones una a una, sino simplemente se envía la llamada al procedimiento. Por tanto, con los procedimientos **se agiliza el envío de instrucciones** desde los clientes al servidor, no recibiendo este tantas peticiones de tareas. Como contrapartida, el servidor tiene una **mayor carga de trabajo** al tener que buscar y decodificar los procedimientos almacenados cuando son invocados desde los clientes.

Cuando las aplicaciones cliente trabajan con **diferentes lenguajes y plataformas**, los procedimientos son muy útiles.

Proporcionan **mayor seguridad** en dos sentidos: por una parte, las aplicaciones y los usuarios no acceden directamente a las tablas, sino que sólo pueden ejecutar algunos procedimientos y no tendrán que construir esos procesos sobre la base de las sentencias que los forman, con el posible riesgo de alteraciones no deseadas de los datos por operaciones indebidamente realizadas; y por otra parte, se pueden establecer autorizaciones de ejecución diferentes, para los usuarios sobre los procedimientos y funciones que se deseen.

Resumir un proceso con varias instrucciones SQL complejas. **Se ejecuta más rápido** como un procedimiento almacenado que si se trata de un programa instalado en el cliente que envía y recibe consultas SQL al servidor.

Los **permisos asociados** a las distintas operaciones que se pueden realizar con procedimientos almacenados son: CREATE ROUTINE (creación), EXECUTE (ejecución), ALTER ROUTINE (borrado). Por ejemplo un usuario puede tener permiso para ejecutar un procedimiento (EXECUTE), pero en cambio puede no tener permiso para crear nuevos procedimientos (CREATE ROUTINE).

Hemos dicho que los procedimientos se almacenan en el SGBD, pero ¿dónde se almacenan?

En MySQL los procedimientos **se almacenan en**:

La tabla proc de la base de datos del sistema mysql. (hasta la versión MySQL 5.7). Esta tabla se crea durante la instalación de MySQL.

La tabla routines de la base de datos de los metadatos information_schema (versiones MySQL 8.x). Esta tabla se crea durante la instalación de MySQL.

Puedes ejecutar las siguientes sentencias para ver los procedimientos en tu servidor MySQL:

```
/* Versiones hasta MySQL 5.7 -- en base de datos del sistema mysql */
use mysql;
show tables;
select * from proc;

/* Versiones MySQL 8.0 -- en base de datos de los metadatos information_schema */
use information_schema;
show tables;
select * from routines;
```

Workbench (Elaboración propia)

Autoevaluación

Con relación a los procedimientos señala la afirmación INCORRECTA:

- Reducen el tráfico de solicitudes del cliente al servidor.
- Controlan las operaciones que los clientes pueden realizar.
- Agilizan la carga de trabajo del servidor.
- Simplifican procesos complejos.

No es la respuesta correcta. Los procedimientos sí reducen el número de solicitudes del cliente.

Incorrecta. Los procedimientos permiten controlar qué instrucciones puede ejecutar un usuario.

Respuesta correcta. La carga de trabajo del servidor aumenta.

No es correcta. Los procedimientos resumen procesos de varias sentencias complejas.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta
4. Incorrecto

4.1.- Creación de procedimientos I.

Un procedimiento almacenado consta, usualmente, de:

- Un nombre.
- Una lista de parámetros.
- Sentencias SQL.
- Sentencias repetitivas.
- Sentencias alternativas.



Para crear **procedimientos** se utiliza la sintaxis:

Everaldo Coelho and
YellowIcon (GNU/GPL)

```
CREATE PROCEDURE NombreProcedimiento ([parámetro1 [, parámetro2,...]])<br />
[características ...] <br /> BEGIN<br />
```

```
CuerpoDelProcedimiento
```

```
END
```

Los **parámetros** permiten al procedimiento recibir y/o devolver información. Y pueden ser:

- De entrada: IN
- De salida: OUT
- De entrada/salida: INOUT

Para definir cada **parámetro** se sigue el siguiente formato:

```
[IN | OUT | INOUT] NombreParámetro Tipo
```

IN: el procedimiento recibe el parámetro y no lo modifica. Lo usa para consultar y utilizar su valor.

OUT: el procedimiento únicamente puede escribir en el parámetro, no puede consultararlo.

INOUT: el procedimiento recibe el parámetro y puede consultararlo y modificarlo.

Los parámetros OUT o INOUT se usan cuando se desea que un procedimiento nos devuelva valores en determinadas variables, esto es, los parámetros de salida permiten simular que el procedimiento devuelve valores.

Es obligatorio escribir una lista de parámetros, aunque sea una lista vacía, reflejada con ().
Por defecto cada parámetro es de tipo IN. Si queremos especificar otro tipo se escribe delante del nombre del parámetro.

Tipo: Es el Tipo de datos devuelto, puede ser cualquier tipo de datos válido de MySQL(INT, DATE, CHAR, VARCHAR, etc).

Ejemplo 1.

En la base de datos TalleresFaber creamos un procedimiento que obtenga un listado con todos los recambios que existen.

```
USE talleresfaber;

CREATE PROCEDURE listado_recambios()
SELECT *
FROM recambios;
```

¿Y dónde está el BEGIN y el END del formato de sintaxis?

Cuando el procedimiento consta de una sola sentencia SQL se pueden omitir el BEGIN y END, pero lo normal es que un procedimiento incluya varias sentencias, y entonces si son obligatorios.

En la **sección de características** se puede especificar la siguiente información:

LANGUAGE SQL

```
| [NOT] DETERMINISTIC<br /> | { CONTAINS SQL | NO SQL | READS SQL DATA |  
MODIFIES SQL DATA }<br /> | SQL SECURITY { DEFINER | INVOKER }<br /> |  
COMMENT 'CadenaComentario'
```

LANGUAGE SQL significa que el cuerpo del procedimiento está escrito en SQL. Por defecto se tiene esa característica para prever la posible construcción de procedimientos almacenados con otros lenguajes como Java.

DETERMINISTIC / NOT DETERMINISTIC. El procedimiento se considera “determinista” si siempre produce el mismo resultado para los mismos parámetros de entrada, y “no determinista” si no es así. Por defecto es NOT DETERMINISTIC.

SQL SECURITY sirve especificar si el procedimiento es llamado, usando los permisos del usuario que lo creó (DEFINER, que es el valor por defecto), o usando los permisos del usuario que está haciendo la llamada (INVOKER).

COMMENT se usa para escribir el comentario que aparecerá cuando se ejecute una sentencia para ver el contenido de un procedimiento o de una función con:

SHOW CREATE PROCEDURE o SHOW CREATE FUNCTION

CuerpoDelProcedimiento: Casi todas las sentencias SQL válidas. El cuerpo del procedimiento empieza con la sentencia BEGIN y termina con la sentencia END y consta de varias instrucciones. Cada una termina con punto y coma (;).

Las cláusulas de la sección de características tienen como **valores predeterminados** los siguientes:

```
LANGUAGE SQL NOT DETERMINISTIC SQL SECURITY DEFINER COMMENT ''
```

Debes conocer

DELIMITER

En MySQL las sentencias se ejecutan después de escribir el punto y coma (;). Para poder escribir el procedimiento completo evitando que se ejecute al encontrar el símbolo (;) tenemos que asignar la función de delimitador a otro carácter, como por ejemplo la barra (\), o bien los símbolos \$\$ o //, como veremos en el siguiente ejemplo. Al finalizar asignaremos al punto y coma su función habitual.

Para **almacenar los resultados de una consulta** directamente **en variables** usamos la sentencia:

```
<b>SELECT NombreColumnas [, ...] INTO NombreVariables [, ...] </b>
```

De esta manera dejamos los valores devueltos en las variables indicadas en lugar de mostrarlos en pantalla. Para hacer esto la consulta debe devolver una sola fila porque no se puede asignar a una variable una lista de contenidos.

Ejemplo 2.

En la base de datos TalleresFaber creamos un procedimiento que obtenga un listado con todas las reparaciones realizadas en un determinado año. Además, debe dejar en un parámetro de salida el total de reparaciones realizadas en ese año.

Observa el uso de DELIMITER.

```
DELIMITER //
CREATE PROCEDURE total_reparaciones_anio(IN anio YEAR, OUT total INT)
BEGIN
SELECT *
FROM reparaciones
WHERE YEAR(fechasalida) = anio;

SELECT count(*) INTO total
FROM reparaciones
WHERE YEAR(fechasalida) = anio;

END//
DELIMITER ;
```

Ejercicio resuelto

Obtener un listado de todos los clientes y otro de todos los vehículos de TalleresFaber. (La base de datos deberá estar abierta).

[Mostrar retroalimentación](#)

```
USE talleresfaber;

DELIMITER |
CREATE PROCEDURE Listados()
BEGIN

    SELECT * FROM CLIENTES;
    SELECT * FROM VEHICULOS;

END |
DELIMITER ;
```

Crea un procedimiento en TalleresFaber que liste a todos los empleados y en una parámetro de salida deje el total de empleados que hay.

[Mostrar retroalimentación](#)

```
USE talleresfaber;

DELIMITER $$

CREATE PROCEDURE empleados_y_total (out total int)
BEGIN
SELECT count(*) into total
FROM empleados;

SELECT *
FROM empleados;

END$$

DELIMITER ;
```

4.2.- Creación de procedimientos II.

Por defecto, al crear un procedimiento queda almacenado en el servidor y se asocia a la base de datos actual. Si se quiere **asociar a una base de datos específica** se escribe el nombre de la base de datos antes del procedimiento:

NombreBaseDatos.NombreProcedimiento

Cuando se crea un procedimiento, el servidor MySQL nos devolverá indicaciones sobre los errores que pueda tener el procedimiento. Cuando la sintaxis es correcta, el servidor almacena el procedimiento.

Everaldo Coelho and
YellowIcon (GNU/GPL)

Los procedimientos pueden ser llamados por cualquier usuario con autorización para ello y por el usuario que los creó. Al ejecutar un procedimiento, el servidor ejecuta automáticamente una sentencia USE BaseDatos.

Para ejecutar un procedimiento, una vez almacenado, usaremos la sentencia:

```
<b>CALL [NombreBaseDatos.]NombreProcedimiento (parámetros_pasados);</b>
```

En el **Ejemplo 1** anterior ejecutaremos:

```
CALL listado_recambios();
```

Cuando un procedimiento utiliza un parámetro de salida (OUT) o un parámetro de entrada/salida (INOUT), para llamar al procedimiento, es necesario pasar una variable que cargue el dato devuelto por el procedimiento. Para este fin utilizaremos las variables de usuario o sesión vistas anteriormente.

Para definir una variable desde la línea de comandos, que será una variable temporal de sistema, la variable debe tener el nombre precedido del carácter @ y se usa la sentencia:

SET @NombreVariable=Valor;

Ejemplos:

```
SET @Edad=23;  
SET @Fecha='2011-03-16';  
SET @Nombre='Alejandra';
```

También se puede crear una variable de sistema especificándola como parámetro en la llamada a un procedimiento, tanto si ya se tenía creada la anterior variable @Num como si se crea en la propia llamada al procedimiento:

CALL NombreProcedimiento(@Num);

Una vez cargada ese valor @Num se puede usar en cualquier sentencia como por ejemplo:

SELECT @Num;

En el **Ejemplo 2** anterior la llamada la procedimiento se haría de la siguiente forma:

```
-- Para invocar al procedimiento con:  
-- primer parámetro de entrada y el segundo es de salida (una variable de sesión)
```

```
CALL total_reparaciones_anio(2011,@tot);
```

```
-- Hay que consultar el valor que ha dejado el procedimiento en @tot
```

```
SELECT @tot;
```

Ejercicio resuelto

En la base de datos TalleresFaber crea un procedimiento que reciba como parámetro de entrada la matrícula de un vehículo y nos muestre las características del vehículo, y como parámetro de salida el número de reparaciones que ha sufrido ese vehículo.

Obtener un listado de los vehículos que igualen o superen el número de reparaciones anterior.

[Mostrar retroalimentación](#)

```
<b></b>

USE talleresfaber;

DELIMITER |
CREATE PROCEDURE NumReparaciones (IN Matri varchar(8), OUT NumRep INT)
BEGIN

    SELECT * FROM VEHICULOS WHERE Matricula=Matri;
    SELECT COUNT(Matricula) INTO NumRep FROM REPARACIONES WHERE Matricula=Matri;

END |
DELIMITER ;
```

Para ejecutar el procedimiento:

```
Call NumReparaciones('1313 DEF', @NumRep);
```

Para ver el valor devuelto:

```
<b></b>
```

```
SELECT @NumRep AS 'Número de reparaciones';
```

```
<b> </b>
```

Para ver los vehículos que igualen o superen el número de reparaciones:

```
<b></b>
```

```
SELECT VEHICULOS.Matricula, Marca, Modelo, Count(*)
FROM VEHICULOS
INNER JOIN REPARACIONES ON VEHICULOS.Matricula=REPARACIONES.Matricula GROUP BY VEHICULOS.Matricula
HAVING Count(*) >=@NumRep;
```

```
<b> </b>
```

Ejercicio Resuelto Procedimientos

Utilizando la base de datos campeonato realiza las siguientes tareas:

- 1.- Crea un procedimiento que recibe el código de un concursante y muestra un listado con los juegos en los que participa. En un parámetro de salida deja ese total.
- 2.- Crea un procedimiento que muestra un listado de todos los juegos con dificultad la que se pasa como parámetro y deja en un parámetro de salida la media de los megusta de esos juegos.
- 3.- Genera un listado con los juegos que tienen un megusta superior a la media obtenida en apartado anterior.

[Mostrar retroalimentación](#)

Te resolveremos el apartado 1.-

Intenta hacer tú los apartados 2.- y 3.-

```
USE campeonato;
DELIMITER //
CREATE PROCEDURE juegos_concursante(IN codigoc CHAR(3), OUT total INT )
BEGIN

SELECT c.cdconcur, c.nombre, p.cdjuego, j.nombre, j.dificultad, j.megusta, p.puntos
FROM concursante c
INNER JOIN participa p ON p.cdconcur=c.cdconcur
INNER JOIN juego j ON j.cdjuego=p.cdjuego
WHERE c.cdconcur = codigoc;

SELECT COUNT(*) INTO total
FROM concursante c
INNER JOIN participa p ON p.cdconcur=c.cdconcur
INNER JOIN juego j ON j.cdjuego=p.cdjuego
WHERE c.cdconcur = codigoc;

END//
DELIMITER ;
```

Un ejemplo de llamada al procedimiento sería:

```
CALL juegos_concursante('C01',@tot);
```

Se consulta el valor que ha dejado el procedimiento en el parámetro de salida:

```
SELECT @tot 'es el total de juegos del consusante';
```

4.3.- Declaración de variables locales.



Everaldo Coelho and
YellowIcon (GNU/GPL)

Cuando creamos una rutina, (procedimiento, función o trigger) puede ser necesario que se tenga que almacenar temporalmente algún valor en variables. Para eso se usarán las variables locales. Veamos como se declaran y se usan.

Declaración de variables con DECLARE:

Dentro de cada procedimiento se pueden definir variables locales, es decir, que sólo existen mientras se ejecuta el procedimiento y después se destruyen. Las variables locales únicamente son visibles dentro del bloque BEGIN ... END donde estén declaradas, y deben estar al comienzo de este bloque, antes de cualquier sentencia.

Para declarar variables locales se utiliza la sintaxis:

```
<b>DECLARE NombreVariable [, ...] Tipo [DEFAULT Valor]</b>
```

Para proporcionar un valor inicial a la variable declarada se utiliza **DEFAULT**. El valor inicial puede ser una constante o una expresión. En caso de no emplear **DEFAULT** el valor por defecto es **NULL**. En cuanto al tipo, puede utilizarse cualquier tipo válido en MySQL.

Para cada variable que se declara es necesario utilizar una sentencia **DECLARE** distinta.

Sentencia SET

Para asignar un valor a cualquier variable (local, global o pasada como parámetro) se utiliza la sentencia **SET**. Las variables que se asignan con **SET** pueden declararse dentro de una rutina o como variables globales de servidor.

Para modificar el valor de una variable o de un parámetro utilizando una **asignación** debe utilizarse la sentencia:

```
<b>SET NombreVariable1=Expresión1 [, NombreVariable2=Expresión2] ... ;</b>
```

En **Expresión** puede haber una constante, una función, una variable, una operación entre ellas o incluso una sentencia **SELECT** que devuelva un solo resultado.

Ejercicio resuelto

Realizar un procedimiento que liste el vehículo o los vehículos que más reparaciones han sufrido durante el mes que se indique:

Mostrar retroalimentación

```
DELIMITER //
CREATE PROCEDURE MasReparaciones(IN Mes INT)
BEGIN
```

```
DECLARE a INT;
SELECT COUNT(*) INTO a
FROM REPARACIONES
WHERE MONTH(FechaEntrada)=Mes
GROUP BY Matricula
ORDER BY COUNT(*) DESC
LIMIT 1;

SELECT VEHICULOS.Matricula, Marca, Modelo
FROM REPARACIONES
INNER JOIN VEHICULOS ON REPARACIONES.Matricula=VEHICULOS.Matricula WHERE MONTH(FechaEntrada)=Mes
GROUP BY VEHICULOS.Matricula
HAVING COUNT(*)=a;

END //
DELIMITER ;
```


Para ver los vehículos más reparados en Enero:

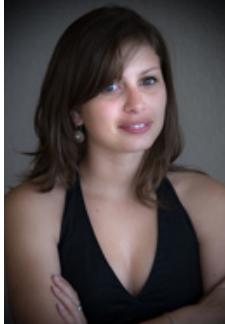

```
CALL MasReparaciones(1);
```

Para llamar al procedimiento usando el mes actual:


```
CALL MasReparaciones(MONTH(CURDATE()));
```


5.- Funciones.

Caso práctico



Aunque el uso de procedimientos almacenados ha supuesto para **Noiba y Naroba** la posibilidad de ejecutar con una sola llamada una serie de sentencias, con las ventajas que hemos visto, los procedimientos **no permiten** algo que ellas consideran muy útil: poder **ser llamados dentro de una sentencia SQL**. Para poder llamar a un conjunto de instrucciones dentro de una sentencia **SQL** necesitará utilizar funciones, que además devuelven siempre un valor.

En unidades anteriores **Noiba** ha incorporado funciones propias de **SQL** para obtener los resultados requeridos en las consultas que ha realizado en TalleresFaber. **¿Podrá crear ella sus propias funciones?**

Alain Bachellier (CC BY-NC-SA)

Otra de las rutinas que podemos crear y almacenar en el servidor de base de datos son las funciones, que serán similares a las funciones **SQL** que proporciona el SGBD.

Un función es conjunto de instrucciones **SQL** que después de ejecutarse devuelven un valor.

La forma de crear una función es similar a la de creación de un procedimiento. Las funciones, una vez creadas, quedan almacenadas en el servidor y pueden ser invocadas en cualquier momento por cualquier cliente MySQL.

Aunque las funciones comparten muchas características con respecto a los procedimientos, presentan también algunas **diferencias**:

Las funciones devuelven siempre un dato a través de una sentencia **<code>RETURN</code>**. El dato se corresponde con un tipo de dato declarado para la función.

Las funciones no pueden trabajar con parámetros **<code>OUT</code>** o **<code>INOUT</code>**, sino únicamente con parámetros de entrada **IN**, por eso no se especifica el tipo.

En las funciones no se pueden usar sentencias que devuelvan filas de resultados (**<code>SELECT</code>**, **<code>SHOW</code>**, **<code>DESC</code>**). Se pueden usar sentencias **<code>SELECT</code>** que devuelvan una fila siempre que los datos devueltos se carguen en variables, para sentencias que devuelvan varias filas se utilizan los cursos.

Las funciones son llamadas a ejecución, al igual que las funciones internas de MySQL, escribiendo su nombre y la lista de parámetros pasados a la función encerrados entre paréntesis. Por tanto no usa una sentencia de llamada como la sentencia **<code>CALL</code>** en el caso de los procedimientos.

Las funciones podrán ser llamadas desde cualquier sentencia **SQL** como **<code>SELECT</code>**, **<code>UPDATE</code>**, **<code>INSERT</code>**, **<code>DELETE</code>**. Los procedimientos nunca pueden ser llamados a ejecución dentro de otra sentencia.

Las funciones se llaman usando: **<code>NombreFunción(parámetros)</code>**.

Las funciones definidas por el usuario están disponibles en versiones anteriores de MySQL. Actualmente se soportan junto con los procedimientos almacenados. En el futuro será posible un marco para procedimientos almacenados externos en lenguajes distintos a SQL, por ejemplo **PHP**.

Los **permisos asociados** a las distintas operaciones que se pueden realizar con funciones almacenadas son: **CREATE ROUTINE** (creación), **EXECUTE** (ejecución), **ALTER ROUTINE** (borrado).

Hemos dicho que las funciones se almacenan en el **SGBD**, pero ¿dónde se almacenan?

En MySQL las funciones **se almacenan en**:

La tabla func de la base de datos del sistema mysql. (hasta la versión MySQL 5.7). Esta tabla se crea durante la instalación de MySQL.

La tabla routines de la base de datos de los metadatos information_schema (versiones MySQL 8.x). Esta tabla se crea durante la instalación de MySQL.

Workbench (Elaboración propia)

Ejercicio resuelto

Escribe una función que recibe una cadena de caracteres y devuelve un mensaje de saludo.

[Mostrar retroalimentación](#)

```
DELIMITER |
CREATE FUNCTION Saludo (S CHAR(20))
RETURNS CHAR(50)
DETERMINISTIC
BEGIN

    DECLARE A CHAR(50) DEFAULT 'HOLA, ';
    SET A=CONCAT(A, S, '!');
    RETURN A;

END |
DELIMITER ;
```

Una vez creada la función podemos llamarla desde sentencias SQL, por ejemplo desde **SELECT**:

```
SELECT Saludo ('Grupo de ASIR');
```

Con la función anterior, saluda a los clientes de TalleresFaber.

[Mostrar retroalimentación](#)

```
<b></b>
```

```
USE talleresfaber;
SELECT Saludo(Nombre) FROM CLIENTES;
```

5.1.- Creación de funciones.

La forma de crear una función es similar a la de creación de un procedimiento. Las funciones, una vez creadas, quedan almacenadas en el servidor y pueden ser invocadas en cualquier momento por cualquier cliente

Vamos a describir detalladamente la sintaxis para crear **funciones**:

[Everaldo Coelho and
YellowIcon \(GNU/GPL\)](#)

```
<b>CREATE FUNCTION NombreFuncion ([Parametro1 [, Parametro2,...]])<br /> RETURNS  
Tipo<br /> [características ...] <br /> BEGIN</b>  
  
<b>CuerpoDeLaFuncion</b>  
  
<b>END</b>
```

Aunque las funciones comparten muchas características con respecto a los procedimientos, presentan también algunas **diferencias**:

Como hemos dicho en el punto anterior con relación a los parámetros, no es necesario especificar el tipo porque no trabajan con parámetros **OUT** o **INOUT**.

La cláusula **RETURNS** es obligatoria e indica el tipo de retorno que nos va a devolver la función.

En el cuerpo de la función debe incluirse un comando **RETURN valor o expresión**, debiendo ser expresión del mismo tipo que la función. Generalmente la sentencia **RETURN** para devolver un resultado es la última del cuerpo de la función.

En las funciones, en las sentencias de su cuerpo, no se pueden usar sentencias que devuelvan filas de resultados (**SELECT**, **SHOW**, **DESC**). Se pueden usar sentencias **SELECT** que devuelvan una fila siempre que los datos devueltos se carguen en variables; para sentencias que devuelvan varias filas trataremos los cursos.

Tipo: El Tipo de datos devuelto puede ser cualquier tipo de datos válido de MySQL.

Las funciones son llamadas a ejecución, al igual que las funciones internas de MySQL, escribiendo su nombre y la lista de parámetros pasados a la función encerrados entre paréntesis. Por tanto no usa una sentencia de llamada como en el caso de los procedimientos.

Las funciones podrán ser llamadas desde cualquier sentencia SQL como **SELECT**, **UPDATE**, **INSERT**, **DELETE**. Los procedimientos nunca pueden ser llamados a ejecución dentro de otra sentencia.

Las funciones se invocan usando: **NombreFunción(parámetros)**.

Las **características**:

```
<b>LANGUAGE SQL<br /> | [NOT] DETERMINISTIC<br /> | { CONTAINS SQL | NO SQL |  
READS SQL DATA | MODIFIES SQL DATA }<br /> | SQL SECURITY { DEFINER | INVOKER }<br /> | COMMENT 'CadenaComentario' </b>
```

La descripción de las características que puede incorporar una función es similar a la que hemos visto para los procedimientos.

Como por ejemplo:

Function **DETERMINISTIC** es la que no realiza cambios en los datos, o para los mismos parámetros de entrada produce los mismos resultados, en otro caso es **NOT DETERMINISTIC**. Por defecto MySQL supone que son **NO DETERMINISTIC**. En nuestro caso, la mayoría de las funciones que diseñemos serán **DETERMINISTIC**.

CuerpoDeLaRutina:

Contiene el código ejecutable de la función entre las sentencias **BEGIN** y **END**.

Ejemplo.

En la base de datos campeonato crea una función que se encargue de devolver la cuota media de los concursantes inscritos en un determinado año.

```
USE campeonato;

DELIMITER //
CREATE FUNCTION cuota_media_concursante( anio YEAR)
RETURNS FLOAT
DETERMINISTIC
BEGIN
DECLARE media FLOAT DEFAULT 0;

SELECT AVG(cuota_inscri) INTO media
FROM concursante
WHERE YEAR(fecha_inscri)=anio;

RETURN media;

END//
DELIMITER ;
```

Para probar la función:

```
SELECT cuota_media_concursante(2020) 'es la cuota media';
```

Ejercicio resuelto

Escribe una función que reciba una fecha y devuelva el año correspondiente a esa fecha. (cómo si esa función no existiera). Los pasos a seguir son:

Convertir la fecha en una cadena de caracteres (función **CONVERT**).
Extraer los cuatro caracteres correspondientes al año (función **LEFT**).
Convertir esa cadena en un número entero sin signo (función **CONVERT**).

Probar la función obteniendo el año correspondiente a tu fecha de nacimiento. Probar la función con el valor que tendrá dentro de un año.

[Mostrar retroalimentación](#)

```
<b></b><b></b>
<b></b>

DELIMITER |
CREATE FUNCTION año (fecha DATE)
RETURNS INT
BEGIN
```

```
DECLARE an CHAR(10);
DECLARE a INT;
SET an=CONVERT(fecha,CHAR);
SET an=LEFT(an,4);
SET a=CONVERT(an,UNSIGNED);
RETURN a;

END|
DELIMITER ;
```

Para probar la función:

```
SELECT año('1985-11-06') AS 'Año de mi nacimiento';

SELECT año (CURDATE())+1 AS 'Año próximo';
```


5.2.- Funciones de librerías básicas disponibles.

Todo SGBD permite que los usuarios creen sus propias funciones y procedimientos. Además, cualquier SGBD incluye un conjunto de funciones en la distribución. Sean funciones de la distribución o funciones desarrolladas por el usuario, pueden usarse en sentencias SQL para que estas sentencias traten los resultados que devuelven las funciones.

MySQL incluye un numeroso conjunto de funciones. En esta unidad se expone la sintaxis de utilización de cada función y se agrupan las funciones por el tipo de datos que manipulan o por el tipo de proceso que llevan a cabo.

Everaldo Coelho and
YellowIcon (GNU/GPL)

Normalmente las funciones operan con las columnas de una tabla, se utilizan en las expresiones que indican los datos que se muestran en una consulta, en las condiciones **WHERE**, en las expresiones **SET** para obtener los datos con los que se modifica una columna, en los valores que se insertan mediante **INSERT**, etc.

Las funciones se clasifican en:

- Funciones matemáticas.
- Funciones de cadenas de caracteres.
- Funciones de fecha y hora.
- Funciones de control de flujo.
- Funciones de búsqueda sobre índices **FULLTEXT**.
- Funciones de conversión.
- Funciones de agregado.
- Otras funciones.

Debes conocer

Puedes consultar un resumen de las funciones disponibles en MySQL en el siguiente enlace. Obviamente no se trata de aprenderlas de memoria, sino de saber que existen para utilizarlas cuando se necesiten.

[Resumen algunas de las funciones disponibles en MySQL](#) (pdf - 0,49 MB)

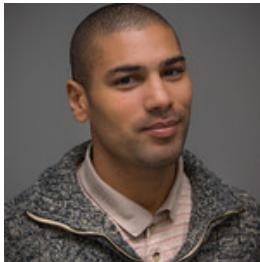
Para saber más

Si necesitas consultar el significado de alguna función y ver algún ejemplo de su aplicación acude al manual de MySQL en los siguientes enlaces:

[Funciones en MySQL](#)

6.- Modificar y borrar procedimientos y funciones.

Caso práctico



[Alain Bachellier \(CC BY-NC-SA\)](#)

Noiba y Vindio están revisando los procedimientos y funciones que han creado hasta ahora para la base de datos TalleresFaber, han visto que hay algún procedimiento que deben modificar para incluir nuevas funcionalidades y hay otros que deben eliminar. ¿Cómo hacerlo?

Precisamente eso es lo que vamos a tratar en este apartado.

Modificar rutinas.

Para modificar el código de un procedimiento o función almacenada tendrás que volver a escribir su código y crear de nuevo esa rutina, pues lo único que se puede cambiar o modificar son sus características.

Para **cambiar** las características de un procedimiento o de una función se utiliza la sentencia **<code>ALTER</code>**, cuya sintaxis es:

<code>ALTER {PROCEDURE | FUNCTION} Nombre [Características ...]</code>

Y recuerda que esas **características** son:

**<code>{CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA}
 | SQL SECURITY {DEFINER | INVOKER}
 | COMMENT 'Cadena'</code>**

Eliminar rutinas.

Para **borrar** un procedimiento o una función almacenada utilizamos la sentencia:

<code>DROP {PROCEDURE | FUNCTION} [IF EXISTS] Nombre</code>

Utilizamos la cláusula **<code>IF EXISTS</code>** para evitar que si el procedimiento o la función ya existen, la sentencia devuelva un error.

Ejemplo.

Para eliminar la función `cuota_media_concursante()` creada anteriormente en la base de datos campeonato, sería:

```
USE campeonato;  
  
DROP FUNCTION IF EXISTS cuota_media_concursante;
```

Observa que no hay que poner los paréntesis, solo el nombre de la rutina.

Otras sentencias útiles:

Esta sentencia muestra algunas **características del procedimiento o de la función** como el nombre de la base de datos, el tipo de rutina: procedimiento o función, el creador, fecha de creación y modificación, etc.

```
<b>SHOW CREATE {PROCEDURE | FUNCTION} Nombre</b>
```

Esta sentencia devuelve la cadena exacta que corresponde a la creación de la rutina.

```
<b>SHOW {PROCEDURE | FUNCTION} STATUS [LIKE 'patrón'] Nombre</b>
```

Si no se incluye un patrón, la información se muestra para todos los procedimientos y funciones.

```
SHOW {PROCEDURE | FUNCTION} STATUS;
```

Ejemplo.

Para mostrar el código del procedimiento creado anteriormente en la base de datos talleresfaber y de nombre total_reparaciones_anio() sería:

```
USE talleresfaber;  
  
SHOW CREATE PROCEDURE total_reparaciones_anio;
```

Para mostrar ciertas características de ese mismo procedimiento y todos los que tengan un nombre que se ajuste a ese patrón, sería:

```
SHOW PROCEDURE STATUS LIKE '%reparaciones_anio%';
```

Para mostrar cierta información de todos los procedimientos del servidor, sería:

```
SHOW PROCEDURE STATUS;
```

Ejercicio resuelto

- 1.- Diseñar una función que calcule los días que pasa un vehículo en el taller de reparaciones. Comprobar que la función no exista.
- 2.- Probar la función en una consulta SELECT para el vehículo de matrícula 1313 DEF.
- 3.- Mostrar la sentencia que describe las características de autor, fecha, etc. de todas las funciones que empiecen por d.

[Mostrar retroalimentación](#)

```
-- 1

DELIMITER |
DROP FUNCTION IF EXISTS dias|
CREATE FUNCTION dias (FechaE date, FechaS date)
RETURNS INT
BEGIN

DECLARE NDIAS INT;
SET NDIAS=FechaS-FechaE;
RETURN NDIAS;

END |
DELIMITER ;

-- 2

SELECT IdReparacion, Matricula, dias (FechaEntrada, FechaSalida) AS 'Dias en el taller'
FROM reparaciones
WHERE matricula='1313 DEF';

-- 3
SHOW FUNCTION STATUS LIKE 'd%';
```

7.- Estructuras de control.

Caso práctico

A medida que van avanzando en la utilización de los procedimientos y de las funciones, a **Naroba y Noiba** se les van abriendo nuevas perspectivas y están muy contentas de sus prácticas de FCT en **BK Sistemas Informáticos**. Muchas de las sentencias que antes tenían que repetir periódicamente, las tienen ahora almacenadas como **procedimientos que pueden ejecutar como una única sentencia**; algunas operaciones complejas que antes tenían que ejecutar en una consulta ahora están recogidas como una **función propia**. Como suele ocurrir, una vez que han empezado a dominar estos nuevos recursos se les van ocurriendo otras aplicaciones para su base de datos. ¿Y si pudieran elegir la sentencia que se ejecuta en función de una condición o del valor de un dato? ¿No se podría repetir un proceso un número determinado de veces hasta que se alcance una condición? **Naroba y Noiba están echando en falta** dos tipos de estructuras que no hemos visto hasta ahora y que van a abrir nuevas posibilidades a sus procedimientos y funciones: **las estructuras de control**.



Alain Bachelier (CC BY-NC-SA)

En el SQL de MySQL disponemos de varias estructuras o sentencias de control de flujo. Estas sentencias sirven para codificar estructuras de decisión y repetitivas en una función o en un procedimiento.

Las sentencias de control de flujo disponibles en MySQL son:

```
<b>IF</b>
<b>CASE</b><b></b></b>
<b>WHILE</b>
<b>REPEAT</b>
<b>LOOP</b><b></b></b>
<b>ITERATE</b>
<b>LEAVE</b>
```

En otros SGBD se utiliza también la sentencia **FOR** que en caso de MySQL no se admite.

Estas estructuras pueden contener una sola sentencia o un bloque de ellas usando los comandos **BEGIN ... END**. Unas estructuras pueden estar anidadas dentro de otras.

A continuación vemos cada una de ellas.

Para saber más

Puedes consultar el siguiente enlace para más de detalle sobre las sentencias de control de flujo en MySQL.

[Sentencias de control de flujo MySQL](#)

7.1.- La sentencia IF.

A veces, puede interesar realizar acciones diferentes en base a que se cumpla o no cierta condición. Esto lo permiten las sentencias alternativas, condicionales o de decisión.

Estas alternativas pueden ser simples, múltiples y anidadas.

La sentencia IF es un tipo de alternativa simple, que permite elegir qué sentencias se ejecutarán y cuáles no, dependiendo de que una condición sea verdadera o falsa.

La sintaxis de la sentencia IF es el siguiente:

```
IF Condición1 THEN Sentencias1  
[ELSEIF Condición2 THEN Sentencias2]  
...  
[ELSE Sentencias_Else]  
END IF;
```

Si la Condición1 del IF se cumple (es verdadera), se ejecutan las sentencias correspondientes (Sentencias1). Si no es verdadera, se evalúa la Condición2 y si es verdadera se ejecutan las sentencias asociadas (Sentencias2), y así sucesivamente con todos los ELSEIF.

Si ninguna de las condiciones es verdadera, en caso de que haya ELSE, se ejecutan las sentencias asociadas a ELSE (Sentencias3), si no hay ELSE no se ejecuta ninguna sentencia.

Ejemplo.

Procedimiento que recibe 2 números y muestra un mensaje indicando el de mayor valor o son iguales.

```
CREATE DATABASE rutinas; -- base de datos para los ejemplos ilustrativos  
USE rutinas;  
  
DELIMITER //  
CREATE PROCEDURE numero_mayor(IN num1 INTEGER, IN num2 INTEGER)  
BEGIN  
IF num1 > num2 THEN  
    SELECT 'El mayor es: ', num1;  
ELSEIF num2 > num1 THEN  
    SELECT 'El mayor es: ', num2;  
ELSE  
    SELECT 'Los dos números son iguales ';  
END IF;  
END//  
DELIMITER ;  
  
-- comprobación  
CALL numero_mayor(4,5);  
  
CALL numero_mayor(5,5);
```

Ejercicio Resuelto

Realizar una función que reciba una fecha y devuelva el nombre del día de la semana que le corresponde.

[Mostrar retroalimentación](#)

```
DROP FUNCTION IF EXISTS diasemana;

DELIMITER |

CREATE FUNCTION DiaSemana(d DATE)

RETURNS VARCHAR(10)

DETERMINISTIC

BEGIN

DECLARE n INT;

SET n=DAYOFWEEK(d);

IF n=1 THEN RETURN 'Domingo';

ELSEIF n=2 THEN RETURN 'Lunes';

ELSEIF n=3 THEN RETURN 'Martes';

ELSEIF n=4 THEN RETURN 'Miercoles';

ELSEIF n=5 THEN RETURN 'Jueves';

ELSEIF n=6 THEN RETURN 'Viernes';

ELSE RETURN 'Sabado';

END IF;

END |

DELIMITER ;
```

Para probar la función:

```
SELECT DiaSemana(CURDATE());
```

Debe mostrar el nombre de día actual.

7.2.- La sentencia CASE.

La sentencia **CASE** se utiliza para establecer opciones múltiples a partir de una expresión.

Estas opciones múltiples se pueden resolver también mediante IF - ELSE anidados, pero el uso de CASE hace más legible el código.

Esta sentencia CASE presenta dos variantes.

[Everaldo Coelho and
YellowIcon \(GNU/GPL\)](#)

El **primer formato** o sintaxis es la siguiente:

```
<b>CASE Expresión </b>
    <b>WHEN Valor1 THEN Sentencias1<br /> [WHEN Valor2 THEN Sentencias2]<br />
        ....<br /> [WHEN ValorN THEN SentenciasN] </b>
    <b>[ELSE Sentencias_Else]</b>
<b>END CASE;</b>
```

Se evalúa la expresión y se ejecutan las sentencias correspondientes al primer valor igual al valor de la expresión.

Si ningún valor es igual, se ejecutan las sentencias que hay dentro de **ELSE**, caso de que hubiera **ELSE**.

Ejemplo.

Procedimiento que recibe un número entero comprendido entre 1 y 5, y muestra un mensaje literal indicando el número recibido.

```
USE rutinas;
DELIMITER //
CREATE PROCEDURE numero_pulsado(IN num INTEGER)
BEGIN
CASE num
    WHEN 1 THEN SELECT 'Has pulsado 1';
    WHEN 2 THEN SELECT 'Has pulsado 2';
    WHEN 3 THEN SELECT 'Has pulsado 3';
    WHEN 4 THEN SELECT 'Has pulsado 4';
    WHEN 5 THEN SELECT 'Has pulsado 5';
ELSE
    SELECT 'Número no valido, debe ser un entero del 1 al 5';
END CASE;

END//
DELIMITER ;

-- comprobación

CALL numero_pulsado(3);

CALL numero_pulsado(8);
```

Ejercicio resuelto

Realizar la función que obtiene el día de la semana correspondiente a una fecha con la sentencia CASE.

[Mostrar retroalimentación](#)

```
USE rutinas;

DROP FUNCTION IF EXISTS DiaSemana;

DELIMITER |

CREATE FUNCTION DiaSemana(d DATE)
RETURNS VARCHAR(10)
DETERMINISTIC
BEGIN
    DECLARE dia VARCHAR(10);
    DECLARE n INT;
    SET n=DAYOFWEEK(d);

    CASE n

        WHEN 1 THEN SET dia= 'Domingo';
        WHEN 2 THEN SET dia= 'Lunes';
        WHEN 3 THEN SET dia= 'Martes';
        WHEN 4 THEN SET dia= 'Miércoles';
        WHEN 5 THEN SET dia= 'Jueves';
        WHEN 6 THEN SET dia= 'Viernes';
        WHEN 7 THEN SET dia= 'Sábado';

    END CASE;

    RETURN dia;
END |

DELIMITER ;

-- Comprobación

SELECT DiaSemana(CURDATE());
```

El **segundo formato** o sintaxis válida para **CASE** es el siguiente:

CASE

 WHEN Condición1 THEN Sentencias1
 [WHEN Condición2THEN
Sentencias2]

 [WHEN CondiciónN THEN

SentenciasN]
 [ELSE Sentencias_Else]

END CASE;

Si una condición se evalúa como verdadera, se ejecutan las sentencias correspondientes.
Si no se cumple ninguna condición, se ejecutan las sentencias de la cláusula ELSE en caso de que la haya.

Ejercicio resuelto

Realizar una función que devuelve la calificación (texto) correspondiente a una calificación numérica con decimales. Deben tratarse errores.

Mostrar retroalimentación

```
USE rutinas;
DROP FUNCTION IF EXISTS Calificacion;
DELIMITER |
CREATE FUNCTION Calificacion (c FLOAT)
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN

    DECLARE Ctexto VARCHAR(20);
    CASE
        WHEN c>=0 AND c<5 THEN SET Ctexto='Suspens' ;
        WHEN c>=5 AND c<6 THEN SET Ctexto='Suficiente';
        WHEN c>=6 AND c<7 THEN SET Ctexto='Bien';
        WHEN c>=7 AND c<9 THEN SET Ctexto='Notable';
        WHEN c>=9 AND c<=10 THEN SET Ctexto='Sobresaliente';
        ELSE SET Ctexto='Calificación errónea';
    END CASE;
    RETURN Ctexto;

END|
DELIMITER ;
```

-- comprobación

```
SELECT Calificacion(7.5);
SELECT Calificacion(11);
```

No hay que confundir las estructuras IF o CASE para **procedimientos almacenados** que acabamos de ver, con las funciones IF() o CASE() ya que se trata de **funciones de control de flujo** y en el caso de CASE su sintaxis es también diferente.

7.3.- La sentencia REPEAT.

La sentencia **REPEAT** permite representar una estructura repetitiva del tipo **repetir ... hasta**, muy común en la mayoría de los lenguajes de programación.

En esta estructura se empieza ejecutando las sentencias que están dentro de **REPEAT**. Al final se evalúa si se cumple o no una condición. Si la condición se cumple, se sale del bucle en caso contrario se vuelve al comienzo y se repite una nueva iteración. Las sentencias continúan repitiéndose hasta que la condición es cierta.

[Everaldo Coelho \(YellowIcon\) \(GNU/GPL\)](#)

La condición de parada del bucle se especifica detrás de la cláusula UNTIL.

El formato o sintaxis de esta sentencia es el siguiente:

```
<b>[Etiqueta_Inicio:] REPEAT </b>
<b>Sentencias;</b>
<b>UNTIL Condición<br /> END REPEAT [Etiqueta_Fin];</b>
```

Ejemplo.

Para ilustrar el funcionamiento de REPEAT vamos a ver un ejemplo sencillo.

En la base de datos rutinas creamos un procedimiento que recibe un número entero y muestra los números anteriores hasta llegar al uno. (El número debe ser mayor que cero).

```
USE rutinas;

DELIMITER //
CREATE PROCEDURE pa_numeros_menores_repeat(IN num INT)
BEGIN
    IF num <= 0 THEN
        SELECT 'El valor introducido debe ser positivo';
    ELSE
        REPEAT -- REPITE las siguientes sentencias
            SELECT num, ' ';
            SET num = num-1;
        UNTIL num<1 -- hasta que la condición es cierta
        END REPEAT;
    END IF;
END //
DELIMITER ;

-- llamada
CALL pa_numeros_menores_repeat(0);
CALL pa_numeros_menores_repeat(10);
```

Ejercicio Resuelto

Realizar una función que recibe una cadena de texto, cadeA, y una letra para devolver una cadena que sustituye con caracteres subrayados todos los de cadeA excepto los que son iguales a la letra pasada.

[Mostrar retroalimentación](#)

```
USE rutinas;

DELIMITER //
CREATE FUNCTION palabra0culto(cadeA VARCHAR(20), letra CHAR(1))
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
    DECLARE oculta VARCHAR(20);
    DECLARE c VARCHAR(20);
    DECLARE pos INT DEFAULT 1;
    SET oculta='';

    REPEAT
        SET c=substring(cadeA, pos, 1);
        IF c= letra THEN
            SET oculta=concat(oculta,letra);
        ELSE
            SET oculta=concat(oculta, '_');
        END IF;
        SET pos=pos+1;
    UNTIL pos > char_length(cadeA)
    END REPEAT;
    RETURN oculta;

END// 
DELIMITER ; 

-- comprobación
SELECT palabra0culto('Anaconda', 'a');
```

7.4.- La sentencia WHILE.

La sentencia **WHILE** permite representar una estructura repetitiva del tipo conocido como **Mientras**...

En este bucle la condición se evalúa al iniciar el bucle, si la condición se cumple, se ejecutan las sentencias que están dentro del bucle, y cuando se llega a la última sentencia se vuelve a evaluar la condición de **WHILE**, repitiéndose este proceso de nuevo si la condición se cumple. La salida del bucle se produce cuando la condición es falsa.

[Everaldo Coelho \(YellowIcon\) \(GNU/GPL\)](#)

La sintaxis o formato es el siguiente:

```
<b>[Etiqueta_Inicio:] WHILE Condición DO</b>
```

```
<b>Sentencias</b>
```

```
<b>END WHILE [Etiqueta_Fin];</b>
```

Ejemplo.

Para ilustrar el funcionamiento de WHILE vamos a ver un ejemplo sencillo.

En la base de datos rutinas creamos un procedimiento que recibe un número entero y muestra los números anteriores hasta llegar al uno. (El número debe ser mayor que cero).

```
USE rutinas;

DELIMITER //
CREATE PROCEDURE pa_numeros_menores_while(IN num INT)
BEGIN
    IF num <= 0 THEN
        SELECT 'El valor introducido debe ser positivo';
    ELSE -- n > 0
        WHILE num >= 1 DO -- Mientras la condición es cierta HAZ las siguientes sentencias
            SELECT num, ' ';
            SET num = num-1;
        END WHILE;
    END IF;
END //
DELIMITER ;

-- llamada
CALL pa_numeros_menores_while(0);
CALL pa_numeros_menores_while(10);
```

Ejercicio resuelto

Realizar un procedimiento que carga en una tabla, FECHAS, con una sola columna F de tipo DATE, las fechas en la que no se han realizado reparaciones desde la fecha pasada hasta la

fecha actual.

[Mostrar retroalimentación](#)

En primer lugar creamos la tabla Fechas en la base de datos talleresfaber:


```
USE talleresfaber;
CREATE TABLE Fechas (Fecha1 DATE);
```

 A continuación creamos el procedimiento:

```
DELIMITER //
CREATE PROCEDURE malasFECHAS( IN dia DATE)
BEGIN
    DECLARE fi DATE;
    DECLARE n INT;
    IF dia>curdate() THEN
        SELECT 'Fecha indicada es incorrecta';
    ELSE
        SET fi=dia;
        WHILE fi<curdate() DO
            SELECT count(*) INTO n
            FROM REPARACIONES
            WHERE FechaEntrada=fi;
            IF n=0 THEN
                INSERT INTO fechas values(fi);
            END IF;
            SET fi= adddate(fi, INTERVAL 1 DAY);
        END WHILE;
    END IF;
END//
```

DELIMITER ;

Para ver los resultados:

```
CALL malasFECHAS('2019-03-16');
SELECT * FROM Fechas;
```


Crear un procedimiento que añada una línea en la tabla Incluyen para añadir un nuevo recambio en una reparación, y que posteriormente actualice el stock de recambios disponible. Comprobar antes de añadir la fila que haya unidades suficientes en stock de ese recambio.

Comprobar el procedimiento con los siguientes datos: IdRecambio= 'BJ_111_666', IdReparacion=2, Unidades=1

[Mostrar retroalimentación](#)


```
DELIMITER //
CREATE PROCEDURE InsertarIncluyen (IN IdRec char(10), IN IdRepar INT , IN Unid SMALLINT)
BEGIN
DECLARE Num INT;
SELECT Stock INTO Num
FROM RECAMBIOS
WHERE IdRecambio = IdRec;

IF Num >= Unid THEN
INSERT INTO Incluyen VALUES (IdRec, IdRepar, Unid);
UPDATE RECAMBIOS
SET Stock=Stock-Unid
WHERE IdRecambio = IdRec;
END IF;

END///
DELIMITER ;

-- comprobación

CALL InsertarIncluyen('BJ_111_666',2,1);

<b> </b>
```

7.5.- Sentencias LOOP, LEAVE e ITERATE.

La sentencia LOOP se utiliza para implementar un bucle sin condición de parada, por lo que será un bucle infinito. **
**

Al inicio y al final de un bucle de este tipo suele aparecer una **etiqueta** (label), aunque no es obligatorio. Si tras END LOOP se coloca una etiqueta, entonces la misma etiqueta debe aparecer al principio del bucle.

Everaldo Coelho and
YellowIcon (GNU/GPL)

En combinación con la sentencia LOOP se puede utilizar la sentencia LEAVE para abandonar el bucle (exit loop), mientras que ITERATE se utiliza para volver a comenzar una iteración (start the loop again).

El formato o sintaxis de LOOP es el siguiente:

**[Etiqueta_Inicio:] LOOP
 Sentencias
 END LOOP [Etiqueta_Fin];**

LOOP Permite realizar un bucle repetitivo que no tiene ninguna condición de salida. Para salir de un bucle LOOP es necesario incluir una sentencia de salida forzada: LEAVE.

La sintaxis o formato de la sentencia LEAVE es la siguiente:

LEAVE Etiqueta

Como ves, es muy sencillo. Este comando se utiliza para abandonar cualquier control de flujo etiquetado. Puede usarse con BEGIN ... END o con bucles.

La sintaxis o formato de la sentencia ITERATE es también muy sencillo:

ITERATE Etiqueta

La sentencia ITERATE sólo puede usarse dentro de LOOP, REPEAT y WHILE. Esta sentencia provoca un salto para reiniciarse de nuevo el bucle desde la primera sentencia. Para ello, es necesario que el bucle correspondiente esté marcado con una etiqueta.

Ejemplo.

Como ejemplo ilustrativo del funcionamiento de este bucle vamos a realizar un procedimiento en la base de datos rutinas que recibe un número entero y muestra los números anteriores hasta llegar al 1. (Controla que el número debe ser mayor que cero).

```
USE rutinas;

DELIMITER //
CREATE PROCEDURE pa_numeros_menores_loop(IN num INT)
BEGIN
    IF num <= 0 THEN
        SELECT 'El valor introducido debe ser positivo';
    ELSE
        b1: LOOP -- REPITE las siguientes sentencias
            SELECT num, ' ';
            SET num = num-1;
    END LOOP b1;
```

```

IF num < 1 THEN -- si la condición es cierta
    LEAVE b1; -- salir del bucle etiquetado b1
ELSE
    ITERATE b1; -- comenzar el bucle
END IF;
END LOOP b1; -- punto final del bucle
END IF;
END // 
DELIMITER ;

```

-- llamada y comprobación

```

CALL pa_numeros_menores_loop(0);
CALL pa_numeros_menores_loop(10);

```

Ejercicio resuelto

Ejemplo de utilización de las sentencias LEAVE e ITERATE dentro de un bucle LOOP.

[Mostrar retroalimentación](#)

```

<b></b>

DELIMITER //
CREATE PROCEDURE BucleLoop (P1 INT, OUT P2 INT, OUT P3 INT)
BEGIN
SET P2=0;
Etiqueta1:LOOP

    SET P1=P1+1;
    SET P2=P2+1;
    IF P1<10 THEN
        ITERATE Etiqueta1;
    END IF;
    LEAVE Etiqueta1;

END LOOP Etiqueta1;
SET P3=P1;
END// 
DELIMITER ;

```

-- comprobación

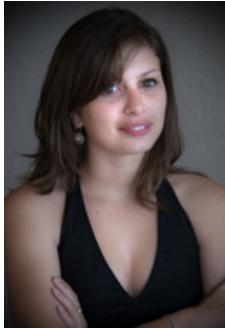
```

CALL BucleLoop(7,@n, @m);
SELECT @n, @m; -- los valores mostrados son 3 y 10

```

8.- Manipuladores de error I.

Caso práctico



Alain Bachellier (CC BY-NC-SA)

Noiba y sus compañeros han observado que, al ejecutar procedimientos o funciones éstos **se interrumpen porque ha ocurrido algún error**, del cual MySQL sólo devuelve un código. En muchos casos a Noiba le interesa que el procedimiento continúe su ejecución o que se lleve a cabo una acción determinada, por eso se dispone ahora a repasar **cómo manipular los errores en MySQL**, y las posibilidades que ofrece para cambiar la respuesta del servidor ante un error en una rutina almacenada.

Cuando se ejecuta una sentencia SQL, el servidor devuelve un **código de error** (numérico) relativo a esa sentencia. Por ejemplo, se devuelve un código de error, cuando se ejecuta una sentencia que trata de insertar una fila con un valor que ya existe en la columna que es clave primaria.

Si la sentencia no ha generado ningún error, entonces devuelve 0 como código de error.

Si una sentencia forma parte de un procedimiento o función y genera un error, entonces se termina automáticamente la ejecución del procedimiento o función.

Esto, generalmente, es un grave problema ya que es normal que, aunque una sentencia de una rutina produzca error, se desee procesar las siguientes sentencias de la rutina. Para solucionar este problema, MySQL permite usar **manipuladores de errores o handler** que sirven para indicar como debe responder el servidor MySQL, en procedimientos y funciones, a situaciones de error.

Además del **código de error**, MySQL también devuelve un **estado de error**, que es algo más genérico que el propio código de error. **Un estado de error engloba varios códigos de errores**. El estado de error suele aparecer encerrado entre paréntesis.

Por ejemplo, si intentamos insertar una fila en una tabla y esa fila no se puede insertar porque el valor de la clave primaria ya existe, MySQL devuelve una indicación de error cuyo **estado de error** o `SQLSTATE es '23000'` y cuyo **código de error MySQL** es 1062 (los valores o códigos de estado devueltos para las sentencias ejecutadas son cadenas de caracteres mientras que los códigos de error son números enteros).

Ejemplo.

En la base de datos campeonato ya existe un equipo en la tabla equipo con el código '01',(la columna cdequipo es la clave primaria de la tabla equipo), luego si se intenta insertar otro equipo con ese mismo código, MySQL devuelve el error 1062 y el estado de error '23000':

```
<b>DECLARE NombreCondición CONDITION FOR {SQLSTATE} ValorEstado |  
CódigoErrorMySQL};</b>
```

Así por ejemplo, para asignar un nombre a una **condición de error**, lo podemos hacer de dos formas:

En función del código de error de 1062

```
DECLARE ClaveRepe CONDITION FOR 1062';
```

En función del código de estado SQLSTATE '23000'

```
DECLARE ClaveRepe CONDITION FOR SQLSTATE '23000';
```

Declaración de un manipulador de errores.

Para declarar un manipulador de errores o de excepción hay que usar la sintaxis:

```
<b>DECLARE TipoManipulador HANDLER FOR ValorCondicion[...] sentencia;</b>  
<b>TipoManipulador:</b>  
    <b>CONTINUE<br /> | EXIT</b>  
<b>ValorCondicion:</b>  
    <b>SQLSTATE ValorEstado<br /> | NombreCondicion<br /> | SQLWARNING<br /> |  
    NOT FOUND<br /> | SQLEXCEPTION<br /> | CódigoErrorMySQL </b>
```

Si se produce cualquiera de las condiciones de error declaradas en el manipulador, se ejecutará la sentencia especificada para el manipulador. Si es más de una sentencia las que se debe ejecutar, éstas deben ir entre BEGIN ... END.

TipoManipulador: puede ser EXIT o CONTINUE.

Un manipulador de tipo **CONTINUE** hace que prosiga la ejecución de la siguiente sentencia a aquella donde se ha producido un error controlado por el manipulador.

Un manipulador de tipo **EXIT** hace que se termine el bloque en el que se encuentra la sentencia que ha producido el error controlado por el manipulador y, por tanto, termine la rutina en la que se encuentra.

ValorCondicion:

SQLWARNING se usa para referenciar a todos los valores de estado que comienzan por 01.

NOT FOUND se usa para referenciar a todos los valores de estado que comienzan por 02.

SQLEXCEPTION se usa para referenciar a todos los valores de estado que no son controlados por **SQLWARNING** y por **NOT FOUND**.

Para saber más

En el siguiente enlace puedes ver la documentación oficial MySQL de Manejadores de error.

[Manajadores de error MySQL](#)

En el siguiente enlace tienes todos los códigos de error que nos puede devolver MySQL:

[Códigos de error de MySQL](#)

8.1.- Manipuladores de error II.

Veamos algunos ejemplos, simplemente ilustrativos, para entender mejor como funcionan los manipuladores de error.

Ejemplo.

En la base de datos talleresfaber, supongamos que tenemos el siguiente procedimiento:

[Everaldo Coelho \(YellowIcon\)](#)
(GNU/GPL)

```
DELIMITER |
CREATE PROCEDURE Manipuladores()
BEGIN

    BEGIN

        INSERT INTO VEHICULOS (Matricula, Marca, Modelo) VALUES ('1234BMY','audi','A6');

        SELECT * FROM CLIENTES;
        SELECT COUNT(*) FROM CLIENTES;

    END;
    SELECT * FROM REPARACIONES;

END |
DELIMITER ;
```

Suponiendo que hacemos la llamada al procedimiento y la matrícula a insertar ya existe en la tabla, no se ejecutará ninguna de las sentencias que siguen a la sentencia **INSERT** que da error.

```
CALL Manipuladores()

<b>ERROR 1062 (23000): Duplicate entry '1234BMY' for key 1</b>
```

Sabiendo que un error de inserción por clave duplicada da el valor de estado '23000' y el código de error 1062, modificaríamos el procedimiento anterior para que controlase ese error de una forma similar a la siguiente:

```
DELIMITER |
CREATE PROCEDURE Manipuladores()
BEGIN

    DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SELECT 'Se ha producido un error';
    BEGIN
        INSERT INTO VEHICULOS (Matricula,Marca, Modelo) VALUES ('2233 ABC','Ford','Mondeo');
        SELECT * FROM CLIENTES;
        SELECT COUNT(*) FROM CLIENTES;
    END;
    SELECT * FROM REPARACIONES;

END|
DELIMITER ;
```

Ahora, al ejecutar el procedimiento, se ejecuta la sentencia asociada al manipulador cuando se produce el error, es decir, se escribe el mensaje **se ha producido un error** y se continua (tipo **CONTINUE**) con la sentencia que hay a continuación de la sentencia **INSERT** que produjo error. Si el

manipulador fuese tipo **<code>EXIT</code>** se ejecutaría la sentencia asociada al manipulador, pero no las siguientes y se iría al final del procedimiento.

La declaración del manipulador podríamos haberla hecho usando el código de error y no el valor de estado de la sentencia. En su lugar podríamos haber puesto:

```
DECLARE CONTINUE HANDLER FOR 1062 SELECT 'Se ha producido un error';
```

O incluso podríamos haber definido previamente una condición de error llamada **<code>Respuesta</code>**, para después declarar un manipulador para esa condición de error:

```
DECLARE Respuesta CONDITION FOR 1062;
DECLARE CONTINUE HANDLER FOR Respuesta SELECT 'Se ha producido un error';
```

Ejercicio resuelto

Partiendo del ejercicio resuelto del apartado 4.2, añade un Handler que evite que las consultas se ejecuten cuando la matrícula que se reciba no exista en TalleresFaber:

[Mostrar retroalimentación](#)

```
USE talleresfaber;
DROP PROCEDURE IF EXISTS NumReparaciones;
DELIMITER |
CREATE PROCEDURE NumReparaciones (IN Matri varchar(8), OUT NumRep INT)
BEGIN
DECLARE m VARCHAR(8);
DECLARE EXIT HANDLER FOR SQLSTATE '02000' SELECT 'La matricula no existe';
SELECT MATRICULA into m
FROM VEHICULOS
WHERE Matricula=Matri;
BEGIN

SELECT *
FROM VEHICULOS
WHERE Matricula=Matri;

SELECT COUNT(Matricula) INTO NumRep
FROM REPARACIONES
WHERE Matricula=Matri;

END ;
END |
DELIMITER ;

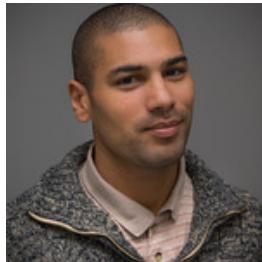
-- comprobación con matrícula que no existe

Call NumReparaciones('2020 DEF', @NumRep);
SELECT @NumRep;
```

< b > </ b >

9.- Manejo de cursos I.

Caso práctico



[Alain Bachellier \(CC BY-NC-SA\)](#)

Vindio, Noiba y Naroba están consiguiendo sacar mucho partido a las funciones y procedimientos que tiene almacenados en la base de datos; no sólo para sus propias consultas sino que le permiten ocultar al resto de usuarios de TalleresFaber operaciones complejas y manejar las acciones que deben llevarse a cabo cuando se produce algún error.

Pero aún encuentran algunas limitaciones importantes, como por ejemplo que el resultado de las consultas tratadas en sus funciones y procedimientos tenga que devolver una única fila. **¿No podría utilizarse algún elemento que le permita tratar varias filas?**

Efectivamente, se trata de los **cursos** y a continuación aprenderás a usarlos en tus procedimientos y funciones.

Un cursor es una consulta declarada que provoca que el servidor, cuando se realiza la operación de **abrir cursor**, cargue en memoria los resultados de la consulta en una tabla interna. Teniendo abierto el cursor, es posible, mediante una sentencia **<code>FETCH</code>**, leer una a una las filas correspondientes al cursor y, por tanto, correspondientes a la consulta definida. Los cursos deben declararse después de las variables locales.

Para hacer **uso de un cursor**, tendremos que:

- Declarar el cursor (después de las variables locales).
- Abrir el cursor.
- Asignar las filas al cursor según tarea a realizar.
- Cerrar el cursor una vez finalizada la tarea.

Declarar el cursor.

Un cursor se declara siguiendo la sintaxis:

```
<code>DECLARE NombreCursor CURSOR FOR SELECT .....;</code>
```

En la sentencia **<code>SELECT</code>** de declaración del cursor puede haber cualquier cláusula utilizada dentro de una **<code>SELECT</code>**, excepto la cláusula **<code>INTO</code>**. En un procedimiento o en una función podemos definir tantos cursos como necesitemos.

Abrir un cursor.

Para abrir el cursor o, lo que es lo mismo, hacer que los resultados de la consulta asociada al cursor queden cargados en memoria, se usa la sentencia:

```
<code>OPEN NombreCursor;</code>
```

La primera vez que se lea sobre el cursor se leerá la primera fila de la consulta, la segunda vez se leerá sólo la segunda fila y así sucesivamente. Si quisieramos volver a leer desde la primera, tendríamos que cerrar el cursor y abrirlo nuevamente.

Asignar las filas de un cursor.

Para leer la fila actualmente disponible en el cursor, se debe usar la sintaxis:

```
<b>FETCH NombreCursor INTO var1 [, var2] ...;</b>
```

Esta sentencia asigna los valores devueltos de la fila que se está leyendo sobre las variables indicadas tras `INTO`. Debe haber una variable por cada valor que devuelve el cursor (por cada valor seleccionado en la `SELECT`). Un cursor se comporta como un puntero que inicialmente apunta a los datos de la primera fila y, cuando se lee, el puntero se incrementa para apuntar a la siguiente fila y así sucesivamente hasta que el puntero llega al final tomando el valor nulo.

Cerrar un cursor.

Todo cursor abierto debe ser cerrado. No es necesario haber consultado todas las filas controladas por el cursor para cerrar el cursor, se puede cerrar en cualquier momento. Si no se cierra, se cerrará al final del comando en el que se ha declarado.

La sintaxis para cerrar un cursor es:

```
CLOSE NombreCursor;
```

Ejemplo.

En el siguiente ejemplo se crea un procedimiento que obtiene en una variable S la suma de los Precios correspondientes a los 3 Recambios más caros obtenidos en una consulta:

```
USE talleresfaber;
DROP PROCEDURE IF EXISTS SumaPrecios;

DELIMITER $$

CREATE PROCEDURE SumaPrecios (OUT S INT)
BEGIN

    DECLARE prec INT;
    DECLARE cur_1 CURSOR FOR
        SELECT PrecioReferencia
        FROM RECAMBIOS
        ORDER BY PrecioReferencia DESC;

    OPEN cur_1;
    SET S=0;
    FETCH cur_1 INTO prec;
    SET S=S+prec;
    FETCH cur_1 INTO prec;
    SET S=S+prec;
    FETCH cur_1 INTO prec;
    SET S=S+prec;
    CLOSE cur_1;

END $$

DELIMITER ;
```


La llamada al procedimiento sería:

```
CALL SumaPrecios(@suma);
SELECT @suma;
```

Ejercicio resuelto

El ejemplo anterior de cursor, impleméntalo mediante un bucle REPEAT y un bucle WHILE.

[Mostrar retroalimentación](#)

Bucle REPEAT

```
USE talleresfaber;

DROP PROCEDURE IF EXISTS SumaPrecios_repeat;

DELIMITER $$
CREATE PROCEDURE SumaPrecios_repeat (OUT S INT)
BEGIN

    DECLARE prec INT;
    DECLARE n INT DEFAULT 1;
    DECLARE cur_1 CURSOR FOR
        SELECT PrecioReferencia
        FROM RECAMBIOS
        ORDER BY PrecioReferencia DESC;

    OPEN cur_1;
    SET S=0;
    REPEAT
        FETCH cur_1 INTO prec;
        SET S=S+prec;
        SET n=n+1;
    UNTIL n>3
    END REPEAT;
    CLOSE cur_1;
END $$
DELIMITER ;

-- comprobación
CALL SumaPrecios_repeat(@suma);
SELECT @suma;
```

Bucle WHILE

```
USE talleresfaber;
DROP PROCEDURE IF EXISTS SumaPrecios_while;
```

```
DELIMITER $$  
CREATE PROCEDURE SumaPrecios_while (OUT S INT)  
BEGIN  
  
    DECLARE prec INT;  
    DECLARE n INT DEFAULT 1;  
    DECLARE cur_1 CURSOR FOR  
        SELECT PrecioReferencia  
        FROM RECAMBIOS  
        ORDER BY PrecioReferencia DESC;  
  
    OPEN cur_1;  
    SET S=0;  
    WHILE n<= 3 DO  
        FETCH cur_1 INTO prec;  
        SET S=S+prec;  
        SET n=n+1;  
    END WHILE;  
    CLOSE cur_1;  
END $$  
DELIMITER ;  
  
-- comprobación  
CALL SumaPrecios_while(@suma);  
SELECT @suma;
```

Recomendación

Te recomendamos que visites la siguiente página para que veas más información de cursos.

[Cursos en MySQL](#)

9.1.- Manejo de cursos II.

A continuación se muestra un **ejemplo** de una función que obtiene usando cursos la suma de los kilómetros de los automóviles en reparación de la marca que se pase a esa función.

Si la marca no existe se controla el error devuelto por una **SELECT** nula ('02000') asignando un valor 0 a la variable Existe. Esto hace que cuando se llegue a no poder leer una fila con la condición dada también se pueda salir del bucle.

[Everaldo Coelho and YellowIcon \(GNU/GPL\)](#)

```
USE talleresfaber;

DELIMITER //
CREATE FUNCTION SumaKmMarca(m char(15))
RETURNS INT
DETERMINISTIC
BEGIN

DECLARE Existe INT DEFAULT 1;
DECLARE Tot, k INT;
DECLARE Cur_1 CURSOR FOR
SELECT Km
FROM VEHICULOS
INNER JOIN REPARACIONES ON VEHICULOS.Matricula=REPARACIONES.Matricula
WHERE Marca=m;
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET Existe = 0;
SET Tot=0;
OPEN Cur_1;
FETCH Cur_1 INTO k ;
WHILE Existe=1 DO
    SET Tot=Tot+k;
    FETCH Cur_1 INTO k;
END WHILE;
CLOSE Cur_1;
RETURN Tot;
END//
DELIMITER ;

-- comprobación

SELECT SumaKmMarca('Seat');
```


Ejercicio resuelto

Crear una función que reciba una Ciudad y nos devuelva en una fila los nombres de los clientes de esa ciudad separados por comas, utilizando cursos.

[Mostrar retroalimentación](#)

```

USE talleresfaber;

DROP FUNCTION IF EXISTS ListadoClientes;

DELIMITER $$

CREATE FUNCTION ListadoClientes(Ciudad VARCHAR(20))
RETURNS VARCHAR(200)
DETERMINISTIC
BEGIN
DECLARE ultima_fila INT DEFAULT 0;
DECLARE SNombre VARCHAR(50) default '';
DECLARE Resultado VARCHAR(200) default '';
DECLARE LONGITUD INT default CHAR_LENGTH(Ciudad);
DECLARE cursor1 CURSOR FOR
SELECT Nombre
FROM Clientes WHERE Right(Direccion, Longitud)=Ciudad;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET ultima_fila=1;
OPEN cursor1;
c1_loop: LOOP
    FETCH cursor1 INTO SNombre;
    IF (ultima_fila=1) THEN
        LEAVE c1_loop;
    END IF;
    IF Resultado ='' THEN
        SET Resultado = TRIM(SNombre);
    ELSE
        SET Resultado = CONCAT(Resultado , ', ',TRIM(SNombre));
    END IF;
END LOOP c1_loop;
CLOSE cursor1;

RETURN Resultado;

END$$
DELIMITER ;

```

-- comprobación

```

SELECT ListadoClientes('Santander'); -- salen:'Carlos,Carmen,María Luisa'

SELECT ListadoClientes('Almería'); -- ninguno

```


Autoevaluación

Indica si la siguiente afirmación es verdadera o falsa

Los cursos solo se pueden usar en el código de un procedimiento y no en el de una función.

- Verdadero Falso

Falso

Un cursor se puede usar tanto el código de procedimiento como en el de una función, pues el cursor nos muestra el resultado de la consulta SELECT, sino que la deja en una tabla temporar para operar con ella.

10.- Disparadores o triggers.

Caso práctico

María ha se ha reunido hoy con el equipo de **Vindio** y han estado hablando sobre el proyecto de TalleresFaber.

Aunque **muchos procesos rutinarios** que se llevan a cabo en TalleresFaber **se han simplificado**, todavía en muchos casos es necesario que las personas recuerden que deben realizar determinadas operaciones. Es el caso, por ejemplo, del registro de la disminución de las unidades en stock de un recambio cuando se incorpora a una reparación.

Para evitar despistes durante el registro de operaciones en TalleresFaber, **Nooiba, Vindio y Naroba** consideran que la solución sería que algunas **rutinas se ejecutaran automáticamente**, es decir que cada vez que se incorpora un recambio a la reparación de

un vehículo la cantidad en stock de ese recambio disminuyera automáticamente sin necesidad de que la encargada de recambios tenga que hacer esa modificación o lanzar el procedimiento que la realiza. Es hora de que conozcamos con ellos la utilidad de los **disparadores o triggers**.

[Nate Steiner \(Dominio público\)](#)

En este apartado vamos a tratar una herramienta muy potente para programar nuestra base de datos que son los disparadores, desencadenadores o triggers en inglés.

Un **trigger o disparador** es una rutina asociada con una tabla, que se activa o ejecuta automáticamente cuando se produce algún evento sobre la tabla.

Es necesario tener en cuenta:

- Un trigger siempre se invoca antes o después de que una fila se inserta, modifica o elimina.
- Un trigger siempre está asociado con una tabla de la base de datos.
- Cada base de datos puede o no, tener uno o más triggers.
- Un trigger se ejecuta como parte de la transacción que lo activó.

Los triggers pueden utilizarse para:

- Implementar restricciones** definidas en el diseño de la base de datos.
- Automatizar acciones críticas** suministrando avisos y sugerencias cuando haya que reparar alguna acción.
- Actualizar los valores de una tabla**, insertar registros en una tabla o llamar a otros procedimientos almacenados.

La sintaxis de **creación de un trigger** es la siguiente:

```
<b></b>  
CREATE TRIGGER trigger_nombre  
{ BEFORE | AFTER } { INSERT | UPDATE | DELETE }  
ON NombreTabla  
  
FOR EACH ROW  
[ { FOLLOWS | PRECEDES } otro_trigger ]
```

```
<b></b>  
BEGIN  
<b></b>  
Sentencias;  
<b></b>  
END  
<b></b>
```

trigger_nombre: nombre que recibe el desencadenador, disparador o trigger.
BEFORE|AFTER: determina si las instrucciones asociadas al trigger se ejecutarán antes (BEFORE) o después (AFTER) de la operación o evento que lo activó.
INSERT|UPDATE|DELETE: es el evento u operación sobre la tabla que activará al desencadenador.
tabla: nombre de la tabla con la que está asociado el desencadenador o trigger.
FOLLOWS/PRECEDES: determina el orden en el que se ejecutará ese trigger en relación al trigger otro_trigger (trigger asociado a la misma tabla, el mismo evento y mismo momento de disparo)
Entre **BEGIN** y **END** van las sentencias asociadas al desencadenador.

Para acceder a los valores de las columnas de la tabla asociada al trigger, se utilizan **OLD** y **NEW**:

En un trigger INSERT sólo se pueden usar los valores NEW.columna.

En un trigger DELETE sólo se pueden usar los valores OLD.columna.

En un trigger UPDATE se puede acceder a los valores NEW.columna que son los nuevos valores que se dan a las columnas y OLD.columna que son los antiguos valores de las columnas antes de la actualización.

Por **ejemplo** el valor de la columna nombre que se ha insertado con un INSERT que ha disparado un trigger, se representa como NEW.Nombre.

Debes tener en cuenta que:

No se puede asociar un trigger a una tabla temporal o a una Vista (VIEW)

No puede haber dos triggers o más para una misma tabla, que respondan al **mismo evento** y en el **mismo momento** de disparo, en versiones anteriores a MySQL 8.0.x.

Si puede haber dos o más triggers asociados a una tabla, respondiendo al mismo evento y que se disparen en el mismo momento a partir de la versión MySQL 8.0.x, y en este caso hay que gestionar el orden o prioridad de ejecución de cada uno de ellos mediante las opciones FOLLOW/PRECED. En otro caso, se activan según el orden de creación..

No pueden incluirse sentencias de control de transacción, tales como COMMIT y ROLLBACK en el código de un trigger.

Dado que los triggers son rutinas, podemos usar para ellos las mismas sentencias que en los procedimientos y en las funciones.

Para crear y ejecutar TRIGGERS se necesita el privilegio TRIGGER.

Ejemplo.

Creamos un trigger sobre la base de datos campeonato y asociado a la tabla juego controlando que si se intenta poner como nuevo valor de la columna megusta un **NULL**, se deja con el antiguo valor que tuviese.

```
USE campeonato;  
  
DELIMITER //  
CREATE TRIGGER comprueba_juego  
BEFORE UPDATE ON juego  
FOR EACH ROW  
BEGIN  
IF NEW.megusta IS NULL THEN
```

```
SET NEW.megusta=OLD.megusta;
END IF;
END// 
DELIMITER ;

-- comprobación
SELECT * FROM juego; -- vemos los juegos

UPDATE juego
SET megusta=NULL
WHERE cdjuego='ELV'; -- modificamos y después volvemos a consultar para ver que no se ha cambiado
```

La sintaxis para **eliminar un trigger** es la siguiente:

```
DROP TRIGGER NombreTabla.NombreTrigger;
```

La sentencia para ver los **triggers de la base de datos en uso** es:

```
SHOW TRIGGERS;
```

Autoevaluación

Indica si la siguiente afirmación es verdadera o falsa.

En un disparador asociado al evento **DELETE** se pueden usar los modificadores **OLD** y **NEW** para referirnos a las columnas de la tabla asociada al trigger.

- Verdadero Falso

Falso

En un disparador **DELETE** solo se puede usar **OLD**.

10.1.- Gestión de errores con SIGNAL.

En este apartado veremos algunos ejemplos más de triggers y como utilizar la sentencia SIGNAL para lanzar condiciones de error dentro de rutinas almacenadas.

Veamos previamente **otro ejemplo de trigger**.

Realizar un trigger que antes de insertar una nueva fila en la tabla facturas modifique el estado de la reparación del vehículo (REPARADO=1) y la fecha de salida de la tabla REPARACIONES sustituyéndola por la fecha actual de emisión de la factura en caso de que no se hubiera modificado anteriormente.

[Everaldo Coelho and YellowIcon \(GNU/GPL\)](#)

```
USE talleresfaber;
DROP TRIGGER IF EXISTS ActualizarReparacion;

DELIMITER $$

CREATE TRIGGER ActualizarReparacion
BEFORE INSERT ON FACTURAS FOR EACH ROW
BEGIN
DECLARE Fechas DATE;
DECLARE Rep INT(1) default 0;
SELECT REPARADO, FechaSalida INTO Rep, Fechas
FROM REPARACIONES
WHERE IdReparacion=New.Idreparacion;

IF REP=0 OR Fechas IS NULL
THEN
UPDATE REPARACIONES
SET REPARADO=1, FechaSalida=curdate()
WHERE IdReparacion=New.IdReparacion;
END IF;
END $$

DELIMITER ;
```

Puedes utilizar la sentencia SIGNAL para devolver de manera personalizada el error que se produce en un procedimiento almacenado, función almacenada o disparador. La sentencia SIGNAL te proporciona el control sobre qué información devolver al generarse el error, tales como el valor devuelto y el mensaje SQLSTATE.

El formato de la sentencia SIGNAL es el siguiente:


```
SIGNAL SQLSTATE | nombre_condicion;
SET item_informacion = valor_1,
item_informacion = value_2, ... ;
```

Un valor genérico de SQLSTATE para gestionar errores es el valor '45000'.

Si se utiliza nombre_condicion, debe estar previamente declarado con un valor de SQLSTATE.

El item_informacion puede ser entre otros un MESSAGE_TEXT.

Ejemplo con SIGNAL.

En la base de datos campeonato vamos a crear un trigger que al intentar eliminar un equipo no deje eliminarlo si su año de fundación es posterior a 2018.

```
USE campeonato;

DELIMITER //
CREATE TRIGGER control_elimina_equipo
BEFORE DELETE ON equipo
FOR EACH ROW
BEGIN
IF OLD.anio_funda > 2018 THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'No puede ser elimiinar un equipo de año > 2018';
END IF;
END///
DELIMITER ;

-- comprobación
DELETE FROM equipo
WHERE cdequipo='07'; -- se mostrará el mensaje personalizado
```

Ejercicio resuelto

Realiza un trigger que cuando modifiquemos la tabla REPARACIONES, si el valor de la columna REPARADO es 1 añada la factura correspondiente insertando una fila en la tabla FACTURAS.

NOTA: Para ejecutar este trigger deberás eliminar previamente el anterior para evitar que se activen mutuamente.

[Mostrar retroalimentación](#)

```
USE talleresfaber;

DROP TRIGGER IF EXISTS EmitirFactura;

DELIMITER $$
CREATE TRIGGER EmitirFactura
AFTER UPDATE ON REPARACIONES FOR EACH ROW
BEGIN
DECLARE Codcli CHAR(5);
DECLARE Nfact INT(4) default 0;
DECLARE Rep INT(1) default 0;
SELECT REPARADO INTO Rep
FROM REPARACIONES
WHERE IdReparacion=New.Idreparacion;
SELECT IdFactura INTO Nfact
FROM FACTURAS WHERE IdReparacion=New.Idreparacion;

IF Rep=1 AND NFact=0
```

```

THEN
SELECT Clientes.CodCliente into Codcli
FROM CLIENTES, VEHICULOS, REPARACIONES
WHERE CLIENTES.CodCliente=VEHICULOS.CodCliente
AND REPARACIONES.Matricula=VEHICULOS.Matricula and IdReparacion>New.Idreparacion;
SELECT IdFactura INTO Nfact
FROM FACTURAS
ORDER BY IdFactura DESC
limit 1;
SET Nfact=Nfact+1;
INSERT INTO FACTURAS VALUES (Nfact, curdate(),Codcli, New.IdReparacion);

END IF;

END $$

DELIMITER ;

```

Ejercicio Resuelto con SIGNAL

En la base de datos campeonato debes crear un trigger para controlar las inserciones de consursantes de la siguiente forma:

- No se puede insertar un consursante sin un valor para cuota de inscripción y fecha de inscripción, se avisa con un mensaje de error.
- La cuota de inscripción debe estar en el rango de 10 a 100€, si no es así, se muestra un mensaje de error.
- La fecha de inscripción no puede ser posterior a la fecha actual.

Indica las sentencias de comprobación que ejecutarías.

[Mostrar retroalimentación](#)

El trigger sería:

```

USE campeonato;
DROP TRIGGER IF EXISTS control_inserta_concursante;

DELIMITER //
CREATE TRIGGER control_inserta_concursante
BEFORE INSERT ON concursante
FOR EACH ROW
BEGIN
IF NEW.cuota_inscri IS NULL OR NEW.cuota_inscri NOT BETWEEN 10 AND 100
THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'La cuota debe estar entre 10 y 100 euros';
END IF;
IF NEW.fecha_inscri IS NULL OR NEW.fecha_inscri > curdate() THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'La fecha de inscripción no puede ser nula ni posterior a la actual';
END IF;
END//
```

DELIMITER ;

Las sentencias de comprobación de todos los casos, serían:

```
-- COMPROBACIÓN
-- Da ERROR, con fecha nula
INSERT INTO concursante (cdconcur, nombre, fecha_inscri, cuota_inscri)
VALUES('A05', 'Ana Cruz',null,50);
-- Da ERROR, con fecha posterior a la actual
INSERT INTO concursante (cdconcur, nombre, fecha_inscri, cuota_inscri)
VALUES('A05', 'Ana Cruz',curdate()+1,50);
-- Da ERROR con cuota nula
INSERT INTO concursante (cdconcur, nombre, fecha_inscri, cuota_inscri)
VALUES('A05', 'Ana Cruz',curdate(),null);
-- da ERROR con cuota fuera de rango
INSERT INTO concursante (cdconcur, nombre, fecha_inscri, cuota_inscri)
VALUES('A05', 'Ana Cruz',curdate(),500.50);
-- SIN ERROR
INSERT INTO concursante (cdconcur, nombre, fecha_inscri, cuota_inscri)
VALUES('A05', 'Ana Cruz',curdate(),50);
```

Para saber más

Para ver mas opciones de la sentencia SIGNAL puedes consultar el siguiente enlace:

[Sentencias SIGNAL en MySQL](#)

Condiciones y términos de uso de los materiales

Materiales desarrollados inicialmente por el Ministerio de Educación, Cultura y Deporte y actualizados por el profesorado de la Junta de Andalucía bajo licencia Creative Commons BY-NC-SA.

Antes de cualquier uso leer detenidamente el siguiente [Aviso legal](#)

Historial de actualizaciones

Versión: 01.00.01	Fecha de actualización: 10/03/22
Actualización de materiales y correcciones menores.	
Versión: 01.00.00	Fecha de actualización: 23/07/20
Versión inicial de los materiales.	

