PROGRAMACIÓN DE BD

Tabla de Contenidos

1.- Introducción

Hasta ahora hemos trabajado con la base de datos de forma interactiva y hemos podido comprobar la potencia de SQL para la realización de consultas y manipulaciones de datos. Sin embargo, esta forma de trabajar no resulta operativa en un entorno de producción ya que esto supondría que todos los usuarios conocen y manejan SQL. Además, a menudo es necesario realizar operaciones complejas y automatizar tareas en la base de datos.

En este caso una solución es la de realizar un programa utilizando cualquier lenguaje compatible y realizar desde este programa los comandos SQL adecuados para realizar el proceso.

Los modernos SGBD ofrecen una solución alternativa que pasa por proporcionar ellos mismos un lenguaje en el que se pueden realizar procesos más complejos que los permitidos utilizando simple SQL pero combinado con la potencia de un lenguaje de programación. A este tipo de programas incrustados y ejecutados por el propio SGBD se les denomina guiones y son el objeto del presente tema.

2.- Lenguaje de programación

Existe una gran variedad de lenguajes de programación de guiones, puesto que esta característica está totalmente fuera de lo definido en los estándares, por lo que cada SGBD lo ha implementado siguiendo su propio camino. Así tenemos PL/SQL (Procedural Language/Structured Query Language) para Oracle, T - SQL (Transact-SQL) para Microsft SQL Server y PL/pgSQL (Procedural Language/PostgreSQL) para PostgreSQL entre otros.

En esta sección y las siguientes se describirán las características comunes a los lenguajes de guiones, sin referenciar un SGBD en concreto.

2.1.- Tipos de guiones

Según el motor de SGBD se pueden tener los siguientes tipos de guiones:

- **Procedimientos almacenados**

  Estos procedimientos se pueden invocar como si fueran sentencias del lenguaje. El usuario los invoca explícitamente utilizando su nombre y se le pueden pasar parámetros y devolver resultados. Se pueden tomar como sentencias escritas por el usuario en contraposición a las que vienen de serie con el SGBD.

- **Funciones de usuario**

  Estas funciones se pueden utilizar desde sentencias del lenguaje como si fueran funciones de las que vienen predefinidas por el SGBD. Por ejemplo, un usuario podría definir una función `MEDIANA` que calculara la mediana de un grupo de valores y esta función podría utilizarse en sentencias `SELECT` o `INSERT`.

- **Procesos disparados**

  Estos procedimientos se invocan de forma automática por parte del SGBD cuando ocurre un evento o eventos determinados. Por ejemplo, se puede escribir un guion que es invocado cada vez que se va a eliminar una fila de determinada tabla. En este guion se podría realizar la "limpieza" de datos de otras tablas relacionadas con la fila que se va a eliminar.

2.2.- Características generales del lenguaje

Los guiones utilizan un lenguaje que suele ser el mismo SQL soportado por el SGBD, con la adición de:

- **Variables**

  Permiten almacenar información durante el proceso del script.

- **Módulos**

  Permiten crear scripts complejos a partir de scripts más pequeños utilizando la filosofía "divide y vencerás" para la creación de scripts.

- **Estructuras de control de flujo**

  Para poder tomar decisiones y realizar iteraciones.

- **Control de excepciones**

  Para poder decidir qué se hace si se produce algún problema durante el proceso de forma que se mantenga la integridad de los datos.

2.3.- Tipos de datos

Los tipos de datos definen qué puede contener una variable de un guion (los valores válidos de una variable) y qué operaciones se pueden realizar sobre ella.

Usualmente los tipos de datos disponibles para las variables de un SGBD coinciden con los tipos de datos disponibles para las columnas de las tablas de dicho SGBD, es decir, si la base de datos dispone de un tipo `INTEGER` para definir columnas de tipo entero, también se podrán crear variables de tipo `INTEGER`, que podrán contener valores enteros con las mismas restricciones que una columna de tipo `INTEGER`.

Asimismo, al igual que ocurre con las columnas, a una variable se le puede asignar el valor especial `NULL`.

2.4.- Identificadores

Las variables se identifican siempre por un nombre que define el usuario. Este nombre se denomina identificador. Aunque el nombre lo define el programador a su gusto, cada SGBD impone una serie de normas que deben cumplir los identificadores a fin de evitar problemas de sintaxis. Por ejemplo, los SGBDs suelen prohibir que se utilicen identificadores que coincidan con palabras clave del lenguaje SQL como SELECT o GROUP.

Es recomendable seguir las siguientes reglas básicas:

- Utilizar sólo letras inglesas (nada de eñes o vocales acentuadas), números y subrayado (_).
- Comenzar siempre los identificadores por letra (los números suelen estar prohibidos como primer carácter de un identificador y algunos sistemas utilizan el subrayado como primer carácter de variables especiales de sistema).
- Suponer siempre que el sistema no distingue entre mayúsculas y minúsculas, esto es, que para el sistema Resultado, resultado, y ReSuLtAdO son la misma variable. Si no lo son no se habrá perdido nada. Si lo son, se habrá evitado un problema difícil de identificar.

2.5.- Declaración de variables

Los SGBDs suelen requerir que las variables se declaren antes de ser utilizadas. La declaración tiene como efecto que el sistema reserve espacio de almacenamiento para la variable y se le asigne un tipo de forma que se pueda conocer qué valores se pueden almacenar en dicho espacio y qué operaciones se permiten sobre dicha variable.

La declaración usualmente se realiza con una palabra clave reservada al efecto y una especificación del identificador y tipo de la variable. Opcionalmente también se suele permitir asignar a la variable un valor inicial.

2.6.- Constantes y literales

En la programación de guiones se hace frecuentemente necesario el uso de valores constantes.

Como viene siendo la tónica habitual, los SGBDs normalmente utilizan la misma sintaxis para los valores constantes o literales que utilizan en las sentencias SQL. Entre otros existen los siguientes tipos de literales:

- Números enteros
- Números reales en notación de coma fija
- Números reales en notación científica
- Cadenas y caracteres
- Tiempos (fechas, horas, fechas / horas)

2.7.- Operadores

Los operadores que se pueden utilizar en la programación de guiones son los mismos que se pueden utilizar en las sentencias SQL. Haciendo un rápido repaso estos son:

- **Operadores aritméticos**

  Se emplean para realizar cálculos aritméticos sobre cantidades numéricas:

  - Suma
  - Resta
  - Multiplicación
  - División
  - División entera
  - Resto de la división entera (módulo)
  - Exponenciación

- **Operadores de relación o comparación**

  Para comparar dos valores. Devuelven un resultado lógico (VERDADERO o FALSO).

  - Mayor que
  - Menor que
  - Mayor o igual que
  - Menor o igual que
  - Igual a
  - Distinto de

- **Operadores lógicos**

  Para combinar valores lógicos y obtener expresiones para las sentencias de toma de decisiones. Operan sobre valores lógicos y devuelven valores lógicos:

  - Y (AND)
  - O (OR)
  - No (NOT)
  - O exclusivo (XOR)
  - No O (NOR)
  - No Y (NAND)

2.8.- Funciones predefinidas

Los operadores vistos en el apartado anterior permiten la realización de muchas operaciones pero no son un juego completo. Habitualmente se necesitan realizar operaciones sobre los datos que, o bien son muy difíciles de implementar utilizando los operadores básicos o bien son totalmente imposibles de realizar utilizando los mismos.

Para esta eventualidad, los SGBDs ofrecen librerías más o menos amplias de *funciones predefinidas*. Una función, en un lenguaje de programación de guiones, es una construcción que toma ninguno, uno o más de un valor de entrada y devuelve un único valor de salida, que depende de los valores de entrada.

El conjunto de funciones predefinidas es fuertemente dependiente del SGBD en que se trate, por lo que sería inútil intentar dar una lista aquí. Sin embargo, todos los SGBDs tienen en común que proporcionan funciones encuadradas en las siguientes categorías:

- **Funciones de cadena**

  Permiten realizar operaciones sobre cadenas de caracteres tales como buscar texto en una cadena, realizar conversiones (mayúsculas / minúsculas,...) o componer cadenas a partir de otras.

- **Funciones numéricas**

  Permiten realizar cálculos con cantidades enteras o reales.

- **Funciones temporales**

  Permiten manipular fechas, horas y duraciones.

- **Funciones de conversión**

  Permiten realizar conversiones entre los distintos tipos de datos (de carácter a número, de fecha a texto,...).

- **Funciones varias**

  Permiten realizar funciones que no encajan en una categoría concreta como obtener información del sistema, conocer el estado de una transacción,…

2.9.- Estructuras de control

Las estructuras de control determinan el orden en que se van a ejecutar las sentencias que componen el guion. Las secuencias se corresponden con las secuencias clásicas de la programación estructurada, aunque algunos SGBDs también incluyen características de orientación a objetos como encapsulación, herencia y polimorfismo, que no se tratarán aquí. Las estructuras que seguro se podrán encontrar en todos los lenguajes de guiones son:

- **Secuencia**

  Es el flujo "natural" del guion. Consta de varias sentencias, una a continuación de la otra. El sistema las ejecuta en el mismo orden en que las encuentra.

- **Condición simple**

  La condición simple bifurca la secuencia de ejecución en dos caminos, dependiendo del valor de una condición. Si la condición produce un valor VERDADERO se sigue un camino de ejecución, mientras que seguirá otro si se produce un valor FALSO. Sea cual sea el camino elegido, al final del mismo la ejecución sigue por un único camino al finalizar la estructura.

- **Condición múltiple**

  La condición múltiple elige entre uno de varios caminos a partir de una condición que produce un resultado con más de un valor (en contraposición a un valor VERDADERO / FALSO, como la condición simple). El camino elegido se ejecuta y se continúa por el final de la estructura, como en la condición simple.

- **Ciclos**

  Los ciclos son estructuras que repiten la ejecución de la estructura que contienen en su interior. El número de veces que se repite la ejecución depende del tipo de ciclo:

  - **Ciclo tipo MIENTRAS**

    Un ciclo tipo MIENTRAS está gobernado por una condición que se evalúa al inicio de dicho ciclo. Si la condición produce un valor VERDADERO se ejecuta el contenido del ciclo. Si el valor es FALSO, se acaba el ciclo, su contenido no se ejecuta y se continúa por la siguiente estructura. Después de cada ejecución del contenido del ciclo se vuelve a evaluar la condición y el proceso se repite. Si no se es cuidadoso y se programa el ciclo de forma que la condición cambie eventualmente se puede producir un ciclo infinito sin salida.

  - **Ciclo tipo REPETIR**

    Un ciclo tipo REPETIR es parecido al tipo MIENTRAS, en el sentido de que depende de una condición pero en este caso la condición se comprueba **al final** de la ejecución del ciclo y cuando el contenido de este se ha ejecutado al menos una vez.

  -

2.10.- Subprogramas

Los programas simples son fáciles de realizar pero cuando los guiones se vuelven más complejos se hace necesaria la presencia de mecanismos que permitan lidiar con dicha complejidad. El mecanismo ofrecido es el de subprogramas.

Un subprograma es una parte de programa que realiza una función específica y que puede ser utilizada desde otra parte del programa. De esta forma los programas se construyen de forma *modular*.

Normalmente, un subprograma realiza una tarea concreta y usualmente necesita, para poder realizar su labor, el recibir información desde la parte del programa que requiere sus servicios y a veces también devuelve información como resultado. Otras veces no devuelve información ninguna o la información procesada se almacena en la base de datos.

A los módulos que no devuelven información se les denomina procedimientos y a los que sí devuelven información, funciones.

2.10.1.- Procedimientos

Un procedimiento es un subprograma que realiza una tarea pero no devuelve información directamente. Usualmente se emplean para realizar tareas en lugar del programa que los utiliza o para implementar una respuesta a un disparador.

EJEMPLO

Supongamos el caso de que se tiene una base de datos que se utiliza para almacenar información de un servicio de Internet (un foro por ejemplo). Una de las tablas que tendría esta hipotética base de datos sería una tabla de usuarios. Supongamos que se decide, como política del sistema, que aquellas cuentas que no han sido accedidas en los últimos seis meses deben ser eliminadas junto con los mensajes enviados desde dichas cuentas.

Ésta sería una tarea apropiada para un procedimiento ya que no devuelve información (directamente al menos, ya que la base de datos si se ve modificada como resultado de su acción).

2.10.2.- Funciones

Una función es un subprograma que realiza un procesado de información y devuelve un único valor. Las funciones pueden realizarse tanto para ser utilizadas desde otros programas como para servir en sentencias SQL como funciones definidas por el usuario, sirviendo efectivamente como una forma de ampliar la librería de funciones predefinidas del SGBD con funciones no disponibles pero que son necesarias o convenientes para un usuario u organización.

EJEMPLO

Supongamos que se tiene una base de datos y, para ciertas tareas estadísticas, se desea calcular la moda de la edad de los usuarios (la moda de una distribución aleatoria es el valor que más veces aparece en la muestra). Esta función no suele ser proporcionada por defecto por los SGBD por lo que debe ser calculada a base de consultas.

Con las funciones definidas por el usuario, un programador puede crear una función MODA() a la que se le proporcione un conjunto de valores y devuelva el valor moda.

Esta sería una tarea apropiada para una función ya que se recibe un conjunto de valores y se devuelve un único resultado.

2.11.- Tratamiento de errores (excepciones)

Un programa robusto debe ocuparse no sólo de realizar la tarea que tiene encomendada sin problemas cuando todo va bien sino saber enfrentarse a los problemas o situaciones excepcionales cuando se presentan.

Por ejemplo, supongamos que se crea un procedimiento al que se le pasa un ID de usuario y elimina todo rastro de dicho usuario de la base de datos. En condiciones normales, el usuario del que se proporciona el ID existirá y se podrá realizar la operación pero, ¿qué ocurre si el usuario no existe? En este caso estamos ante una excepción y el programa debe tratarla de forma que el programa no falle

estrepitosamente sino que se adapte a la nueva situación lo mejor posible e intente realizar lo máximo de la tarea que tiene encomendada o, en caso de que ésto no sea posible, deshacer los cambios realizados y, sobre todo, dejar los datos en un estado consistente siempre.

A este tipo de condiciones excepcionales que ocurren se les denominan *excepciones* y un lenguaje de guiones robusto tiene soporte tanto para generarlas (nuestro programa la generará cuando se encuentre en circunstancias que le impiden continuar) como para responder a ellas (realizando las labores de recuperación que se pueda antes de terminar).

3.- Cursores

Muchas veces se hace necesario el procesamiento de una o más tablas fila a fila dentro de un guion, realizando operaciones complejas a partir de los valores de dicha fila. Para simplificar el desarrollo de dichas operaciones, muchos SGBDs disponen de una estructura de control denominada *cursor*. El uso de los cursores se hace necesario cuando el procesamiento de las filas no se puede hacer en conjunto sino de forma individual para cada fila.

Un cursor se declara como si fuera una variable (de hecho lo es, pero de un tipo más complejo) junto con la consulta que realiza.

Una sentencia (`OPEN`) abre el cursor, evalúa la consulta y hace los resultados accesibles. El cursor se sitúa en la primera fila del resultado. Otra sentencia (`FETCH`) obtiene los valores de las columnas de la fila actual en las variables especificadas y mueve el cursor a la siguiente fila. Por último otra sentencia (`CLOSE`) se encarga de liberar los recursos asociados al cursor.

4.- Disparadores

Usualmente un guion, tal y como se ha visto hasta ahora, se ejecuta a voluntad del usuario que lo debe invocar directamente, pero en ocasiones sería conveniente el disponer de guiones que se ejecutaran cuando se produjeran determinadas condiciones sobre los datos de la base de datos, a fin de responder a estos eventos o condiciones de forma adecuada. Muchos SGBDs implementan para realizar esta tarea el mecanismo conocido como *disparador* (`TRIGGER`).

Un disparador o trigger es un objeto que se asocia a una tabla y a una operación sobre la misma y se invoca cuando se realiza la operación indicada sobre la tabla. Usualmente también se permite especificar si se debe realizar el disparo *antes* o *después* de que se ejecute la operación.

Otro problema que se pueden presentar cuando se utilizan disparadores es la presencia de más de un disparador para una misma operación o tabla. Algunos SGBDs sólo permiten la presencia de un disparador por operación y tabla y la creación de otro nuevo o bien se prohíbe o bien sobreescribe el ya existente. Otros SGBDs, sin embargo, permiten encadenar distintos disparadores, de forma que se invoquen en cadena cuando se produzca la condición de disparo. Esto hace necesario arbitrar algún mecanismo que permita indicar el orden de prioridad de los distintos disparadores así como el procedimiento a seguir cuando alguno de ellos falla o provoca un error.

5.- Temporizadores

Como se ha visto, los disparadores son formas reactivas de ejecutar guiones. Se responde a una acción concreta con la ejecución de un guion.

En otras ocasiones se desean realizar tareas de forma reactiva pero lo que determina la reacción es *el paso del tiempo*. Por ejemplo, supongamos que se tiene una base de datos que soporta un servicio de Internet (digamos que un foro). Usualmente estos servicios permiten el registro de usuarios sin intervención de ningún operador. Los usuarios introducen sus datos y el sistema les registra en el mismo de forma que pueden utilizarlo. Un requisito fundamental de dichos sistemas es que el usuario proporcione una cuenta de correo válida, usualmente por motivos publicitarios. Muchos usuarios comenzaron a proporcionar cuentas falsas o inventadas para evitar molestias, pero los administradores contraatacaron añadiendo un paso más al registro. Cuando el usuario finaliza de introducir sus datos se le envía un mensaje de correo electrónico con un enlace que deben visitar si desean que la nueva cuenta se active y se les da un periodo de tiempo razonable para que realicen dicha activación. Si transcurrido este tiempo no se ha producido la activación, el pre-registro se cancela.

En este caso se tiene una tabla, la de usuarios, que contendrá un número indeterminado de registros correspondientes a usuarios que se han pre-registrado pero que nunca han finalizado el registro. Esta información "basura" no tiene ningún valor y afecta al rendimiento del sistema.

La solución más sencilla es la de chequear la tabla periódicamente y eliminar las cuentas que no estén activas y para las que haya transcurrido un periodo superior al de espera desde la pre-activación.

Esta tarea no se puede realizar con disparadores puesto que no es reactiva a modificaciones en la tabla sino que depende del tiempo.

Para disparar acciones en función del tiempo se dispone de los *temporizadores* o *eventos*.

Un temporizador es un objeto que inicia la ejecución de un programa en momentos determinados por el creador del temporizador. Los hay de dos tipos:

- **De un solo disparo (oneshot).**

  Se programan para un momento determinado en el futuro. Cuando llega ese instante se disparan y se eliminan.

- **De disparo recurrente (periodical)**

  Se programan para que se disparen a intervalos definidos. Después de cada disparo el temporizador se rearma, espera el intervalo y se vuelve a disparar. Esto se realiza continuamente hasta que se eliminan manualmente.

Apéndice A: Programación de Guiones en MariaDB

Los guiones permiten realizar procesamiento sobre los datos en la misma base de datos. En este documento se describen las particularidades de los guiones para el SGBD MariaDB, versión 10.x

1.- Comentarios

Los comentarios son bloques de texto que acompañan a los programas y que no son interpretados como instrucciones. Su función es acompañar al programa y servir de documentación de éste.

MariaDB soporta los siguientes tipos de comentarios:

- **De una sola línea**

  Si la línea comienza por un hashtag (#) o dos guiones seguidos de un espacio (`--`), el resto de la línea, hasta el final de esta, se considera un comentario.

- **De múltiples líneas**

  El comentario comienza por la combinación (`/*`) y finaliza con la combinación (`*/`).

EJEMPLO

```
# Esto es un comentario de una sola linea
-- Y esto también
/* Este comentario
abarca más
de una linea */
```

2.- Tipos de datos

Los tipos de datos que tiene MariaDB son los mismos que los tipos de datos de las columnas de las tablas.

3.- Identificadores

Se deben seguir las siguientes reglas:

- Las variables que se utilicen fuera de un procedimiento almacenado deben comenzar por el carácter arroba (@). Las que se declaren dentro de un procedimiento almacenado no lo necesitan.

- Utilizar sólo letras inglesas (nada de eñes o vocales acentuadas), números y subrayado (_).
- Comenzar siempre los identificadores por letra (los números suelen estar prohibidos como primer carácter de un identificador y algunos sistemas utilizan el subrayado como primer carácter de variables especiales de sistema).
- El sistema tiene un sistema complejo para los identificadores pero lo más seguro es utilizar siempre la misma forma (mayúsculas o minúsculas) y suponer que dos nombres con distinta combinación de mayúsculas y minúsculas pueden ser considerados iguales por el sistema.

4.- Declaración de variables

Si una variable va a ser utilizada dentro de un procedimiento almacenado, necesita ser declarada antes. Si se utiliza fuera de un procedimiento almacenado, la variable se crea la primera vez que se le asigna un valor.

Para declarar una variable se utiliza la sentencia DECLARE, que tiene la siguiente forma:

```
DECLARE identificador1,identificador2,… identificadorX tipo [DEFAULT valor]
```

donde,

- **identificador1, identificador2,… identificadorX**

  Son las variables que se van a declarar. Hay que hacer notar que si hay más de una variable en la lista todas se crearán del mismo tipo y con el mismo valor por defecto si se especificara.
- **tipo**

  Es el tipo de la variable o variables y debe ser uno de los tipos del sistema.
- **DEFAULT valor**

  Es el valor inicial opcional que tendrán las variables al ser creadas. Si no se especifica el valor será NULL.

EJEMPLO

Para declarar las variables x e y como enteras y la variable nombre como cadena de caracteres de 50 caracteres de longitud máxima, con valores por defecto 0 y la cadena vacía, respectivamente, las sentencias serían:

```
DECLARE x, y INTEGER DEFAULT 0;
DECLARE nombre VARCHAR(50) DEFAULT '';
```

5.- Constantes y literales

Las contantes y literales son iguales que las ya especificadas para las sentencias SQL de MariaDB.

6.- Operadores

Los operadores son idénticos a los ya especificados para MariaDB.

7.- Funciones predefinidas

Las funciones se deben consultar en el manual de MariaDB ya que son muchas para describirlas aquí.

8.- Asignación de valores

Para asignar un valor a una variable se utiliza la sentencia `SET` con la sintaxis:

```
SET variable = valor;
```

donde

- **`variable`** es la variable cuyo valor se desea modificar.
- **`valor`** es el nuevo valor de la variable. Puede ser una constante, expresión, otra variable, incluso una subconsulta que devuelva un único valor.

9.- Estructuras de control

9.1.- Secuencia

En MariaDB las secuencias se realizan utilizando la notación:

```
sentencia1;
sentencia2;
...
sentenciaN;
```

es decir, terminando las sentencias con punto y coma “`;`”.

Es importante porque es el marcador que determina que una sentencia ha terminado.

9.2.- Condición simple

En MariaDB la condición simple se expresa con la sintaxis:

```
IF condicion1 THEN sentencias1
   [ELSEIF condicion2 THEN sentencias2]
   [ELSE sentencias3]
END IF
```

donde:

- `condicionX`
  Es una condición que evalúa a `VERDADERO` o `FALSO`. Si la condición se evalúa a `VERDADERO` se ejecutan las sentencias tras la clausula `THEN`. Si es `FALSO` se ejecutan la clausula `ELSE` o `ELSEIF` correspondiente.
- `sentenciasX` es una secuencia.
- **`ELSEIF`**
  Se ejecuta si la condición del `IF` o `ELSEIF` correspondiente ha evaluado a `FALSO`. Se evalúa la condición que tiene si se porta como si fuera un `IF`.
- **`ELSE`**
  Se ejecuta incondicionalmente si la condición del `IF` o `ELSEIF` correspondiente ha evaluado a `FALSO`.

EJEMPLO

El siguiente trozo de código obtiene una edad y calcula la etapa educativa a la que pertenece el alumno

```
DECLARE edad INTEGER DEFAULT 0;
DECLARE etapa VARCHAR(50) DEFAULT '';
# Se lee la edad por algún método que no se describe aquí
IF edad >= 0 AND < 6 THEN
    SET etapa = 'Educacion Infantil';
ELSEIF edad >=6 AND edad < 12 THEN
    SET etapa = 'Educación Primaria';
ELSEIF edad >= 12 AND edad < 16 THEN
    SET etapa = 'Educación Secundaria Obligatoria';
# Si no es ninguna de las anteriores, suponemos que será Bachillerato
ELSE
    SET etapa = 'Bachillerato';
END IF;
```

9.3.- Condición múltiple

La condición múltiple tiene dos sintaxis distintas y con distintos significados.

La primera es:

```
CASE expresion
     WHEN valor1 THEN secuencia1
     WHEN valor2 THEN secuencia2
     ….
     WHEN valorN then secuenciaN
     ELSE secuencia_else
END CASE
```

donde,

- **expresion**
  Es una expresión que se evalúa a un único valor.
- **valor1, valor2, …, valorN**
  Son valores literales. El resultado de la expresión es comparada con cada valor. Si hay coincidencia se ejecuta la secuencia situada después de la clausula THEN correpondiente
- **secuencia1, secuencia2, …, secuenciaN**
  Secuencia que se ejecuta si el valor de la expresión coincide con el valor correspondiente.
- **secuencia_else**
  Esta secuencia se ejecuta si el valor de la expresión no coincide con ninguno de los valores indicados en las clausulas WHEN.

EJEMPLO

La misma clasificación por etapas educativas que antes pero resuelto utilizando CASE, sería:

```
DECLARE edad INTEGER DEFAULT 0;
DECLARE etapa VARCHAR(50) DEFAULT '';
...Se lee la edad por algún método...
CASE edad
# Casos de Educacion Infantil
    WHEN 0 THEN SET etapa = 'Educacion Infantil';
    WHEN 1 THEN SET etapa = 'Educacion Infantil';
    WHEN 2 THEN SET etapa = 'Educacion Infantil';
    WHEN 3 THEN SET etapa = 'Educacion Infantil';
    WHEN 4 THEN SET etapa = 'Educacion Infantil';
    WHEN 5 THEN SET etapa = 'Educacion Infantil';
```

```
# Casos de Primaria
    WHEN 6 THEN SET etapa = 'Educacion Primaria';
    WHEN 7 THEN SET etapa = 'Educacion Primaria';
    WHEN 8 THEN SET etapa = 'Educacion Primaria';
    WHEN 9 THEN SET etapa = 'Educacion Primaria';
    WHEN 10 THEN SET etapa = 'Educacion Primaria';
    WHEN 11 THEN SET etapa = 'Educacion Primaria';
# Casos de ESO
    WHEN 12 THEN SET etapa = 'Educacion Secundaria Obligatoria';
    WHEN 13 THEN SET etapa = 'Educacion Secundaria Obligatoria';
    WHEN 14 THEN SET etapa = 'Educacion Secundaria Obligatoria';
    WHEN 15 THEN SET etapa = 'Educacion Secundaria Obligatoria';
# EL resto suponemos que es Bachillerato
    ELSE SET etapa = 'Bachillerato';
END CASE;
```

La sintaxis alternativa es muy similar a la de la condición simple:

```
CASE
     WHEN condicion1 THEN secuencia1
     WHEN condicion2 THEN secuencia2
     ....
     WHEN condicionN THEN secuenciaN
     ELSE secuencia_else;
END CASE;
```

Aqui la diferencia es que no se proporciona expresión que de el valor a comparar sino que cada caso (`WHEN`) evalúa su propia condición. La primera que evalúa a `VERDADERO` ejecuta la secuencia asociada con la clausula `THEN`.

Si ninguna de las condiciones se cumple, entonces se ejecuta la secuencia asociada a la clausula `ELSE`.

EJEMPLO

El mismo ejemplo, quedaría ahora:

```
DECLARE edad INTEGER DEFAULT 0;
DECLARE etapa VARCHAR(50) DEFAULT '';
# Se lee la edad por algún método
# Se hace una línea por caso con la condición correspondiente
```

```
CASE
    WHEN edad >= 0 AND < 6 THEN
         SET etapa = 'Educacion Infantil';
    WHEN edad >=6 AND edad < 12 THEN
         SET etapa = 'Educacion Primaria';
    WHEN edad >= 12 AND edad < 16 THEN
         SET etapa = 'Educacion Secundaria Obligatoria';
    ELSE SET etapa = 'Bachillerato';
END CASE;
```

9.4.- Ciclo Mientras

Tene la siguiente sintaxis:

```
WHILE condicion DO
     secuencia
END WHILE
```

donde,

- `condicion`
  Es la condición que se evalúa para determinar el fin del ciclo. Si la condición evalúa al valor VERDADERO, se ejecutan la secuencia. Si evalúa a FALSO se sale del ciclo. La condición se evalúa antes de realizar las sentencias.
- `sentencia`
  Es una secuencia de sentencias que se ejecutará ninguna, una o varias veces, dependiendo de la condición.

EJEMPLO

Para introducir en una tabla la secuencia de los primeros 100 números se podría hacer lo siguiente:

```
# Contador para el ciclo (se inicializa a cero)
DECLARE c INTEGER DEFAULT 0;
# Hasta que el valor del contador llegue a 100
WHILE c < 100 DO
    # Se inserta el valor en la tabla secuencia
    INSERT INTO secuencia (valor) VALUES (c);
    # Se incrementa el contador al siguiente valor
    SET c = c + 1;
END WHILE;
```

9.5.- Ciclo Repetir

La sintaxis es la siguiente:

```
REPEAT
     secuencia
UNTIL condicion
END REPEAT
```

donde,

- **secuencia**
  Es la secuencia de sentencias que se ejecutará como parte del ciclo. La secuencia se ejecutará al menos una vez.
- **Condicion**
  Su valor se evalúa al final de cada iteración del ciclo. Si produce un valor VERDADERO se termina el ciclo. Si produce un valor FALSO, se realiza otra iteración.

EJEMPLO

El mismo ejemplo que para WHILE, utilizando REPEAT:

```
# Se declara el contador y se inicializa a cero
DECLARE c INTEGER DEFAULT 0;
REPEAT
    #Se inserta el valor en la tabla secuencia
    INSERT INTO secuencia (valor) VALUES (c);
    # Se incrementa el contador en uno
    SET c = c + 1;
# El valor 100 no entra como en el WHILE porque se utiliza >=
UNTIL c >= 100
END REPEAT;
```

Nótese que, dado que el ciclo se ejecuta al menos una vez y que la condición debe valer VERDADERO para *salir* del ciclo, la condición es distinta a la del ejemplo MIENTRAS. Además, el ciclo no se termina con la clausula UNTIL, como suele ser común en los lenguajes de programación de propósito general sino que necesita terminarse con END REPEAT.

10.- Subprogramas

MariaDB soporta dos tipos de subprogramas: procedimientos y funciones.

10.1.- Procedimientos

Los procedimientos se declaran con la siguiente sintaxis:

```
DELIMITER //
CREATE [OR REPLACE] PROCEDURE nombreproc (listaparametros)
BEGIN
     secuencia
END//
DELIMITER ;
```

donde,

- Las líneas **DELIMITER** se utilizan para modificar el delimitador de sentencia por defecto temporalmente a un nuevo valor distinto de `;` que se usa dentro del procedimiento. El valor temporal habitual es `//` que no se presenta en las sentencias SQL normales ni es un operador válido.
- La opción **OR REPLACE** permite que no se produzca error si el procedimiento ya existe. En este caso se reemplazaría por la nueva definición. Si no se especifica y el procedimiento ya existe se produce un error y éste queda inalterado.
- `nombreproc` es el nombre que se le va a proporcionar al nuevo procedimiento. Debe ser distinto al nombre de cualquier otro procedimiento que exista ya en la base de datos.
- `listaparametros` es la lista de parámetros del procedimiento. Puede estar vacía, en cuyo caso los paréntesis si deben ir, aunque sin nada entre ellos o puede constar de uno o más parámetros separados por comas. Cada parámetro consta de las siguientes partes:
  - **Indicador de dirección**

    Indica si el parámetro es de entrada (`IN`), de salida (`OUT`) o de entrada / salida (`INOUT`). Si un parámetro es de entrada, su valor no se puede modificar dentro del cuerpo del procedimiento, si es de salida, su valor no se puede leer pero si escribir dentro del procedimiento y si es de entrada / salida se puede tanto leer como escribir dentro del procedimiento.
  - **Nombre del parámetro**

    Rigen las mismas reglas que para los nombres de variables.
  - **Tipo del parámetro**

    Igual que si fuera una variable
- **`secuencia`** es la secuencia que compone el cuerpo del procedimiento.

EJEMPLO

Supongamos que se tiene la tabla

empleado

codigo	nombre	edad	oficina	titulo	fecha_contrato	codigo_jefe	cuota_ventas	ventas_obtenidas
101	Antonio Viguer	45	12	Representante	20/10/06	104	300000	305000
102	Alvaro Jaumes	48	21	Representante	10/12/06	108	350000	474000
103	Juan Rovira	29	12	Representante	01/03/07	104	275000	286000
104	Jose Gonzalez	33	12	Dir Ventas	19/05/07	106	200000	143000
105	Vicente Pantall	37	13	Representante	12/01/08	104	350000	368000
106	Luis Antonio	52	11	Dir General	14/06/08	NULL	275000	299000
107	Jorge Gutierrez	49	22	Representante	14/11/08	108	300000	186000
108	Ana Bustamante	62	21	Dir Ventas	12/10/09	106	350000	361000
109	Maria Sunta	31	11	Representante	12/10/16	106	300000	392000
110	Juan Victor	41	22	Representante	13/01/10	104	500000	760000

y se desea crear un procedimiento `ventas_por_oficina` que obtenga el total de ventas de una oficina, a partir del código de oficina que se le proporciona.

La definición del procedimiento sería:

```
DELIMITER //
CREATE PROCEDURE ventas_por_oficina (IN ofi INTEGER, OUT ventas INTEGER)
BEGIN
# Obtiene la suma de las ventas de la oficina indicada y almacena
# el resultado en el parametro de salida ventas
    SET ventas = (SELECT SUM(ventas_obtenidas) FROM empleado
                  WHERE oficina = ofi);
END//
DELIMITER ;
```

10.2.- Llamadas a procedimientos

Para llamar un procedimiento se utiliza la sentencia `CALL`, con la siguiente sintaxis:

```
CALL nombre_procedimento (listaparametros)
```

donde,

- `nombre_procedimiento` es el procedimiento a invocar.
- `listaparametros`

  Son los parámetros del procedimiento. Si no tiene parámetros hay que dejar la lista vacía `()`. Los parámetros de entrada pueden ser cualquier expresión pero los parámetros de salida o de entrada/salida deben ser variables ya que deben poder recibir un valor desde el procedimiento almacenado.

EJEMPLO

Para invocar el procedimiento creado en el apartado anterior para la oficina 12, se podría realizar de la siguiente manera:

```
# Invoca el procedimiento. Se pasa una nueva variable @ventas para que
# reciba el valor calculado en el procedimiento
    CALL ventas_por_oficina(12,@ventas);

# Forma "extraña" de SELECT. Si se hace SELECT y una variable se genera
# un resultado con una columna y una fila conteniendo el valor de la
# variable. Util para depuracion.
    SELECT @ventas;
```

10.3.- Eliminar un procedimiento

Un procedimiento se elimina utilizando la sintaxis:

```
DROP PROCEDURE [IF EXISTS] nombre;
```

donde **nombre** es el nombre del procedimiento a eliminar. A partir de ese momento el procedimiento ya no se puede volver a invocar.

La opción `IF EXISTS` se utiliza para que la sentencia no lance un error si el procedimiento no existe.

EJEMPLO

Para eliminar el procedimiento `ventas_por_oficina` ya creado se utilizaría:

```
DROP PROCEDURE ventas_por_oficina;
```

10.4.- Funciones

Existen dos formas de crear funciones en MariaDB:

- **Forma nativa**

  En esta forma hay que desarrollar la función utilizando un lenguaje de programación estándar (C, en la mayoría de los casos) que conecta con el servidor MariaDB para realizar el cálculo requerido. Es la opción más potente pero más compleja de programa.

- **Forma SQL**

  En esta forma se pueden utilizar el lenguaje de guiones de MariaDB para realizar la función pero el resultado es más limitado. Por ejemplo, con este método no se pueden crear funciones agregadas, esto es, funciones que tomen un grupo de valores de tamaño indefinido y devuelvan un resultado. Para ello hay que desarrollar la función en forma nativa.

En esta exposición sólo se describirán las funciones en forma SQL, que se definen de una forma muy parecida a la empleada para describir los procedimientos. Esta sintaxis es la siguiente:

```
DELIMITER //
CREATE [OR REPLACE] FUNCTION nombre_funcion (listaparametros) RETURNS tipo
[NOT] DETERMINISTIC
BEGIN
     secuencia
END//
DELIMITER ;
```

donde,

- La opción **OR REPLACE** hace que si ya existe una función con el mismo nombre se reemplace por la nueva definición sin levantar un error.
- **nombre_funcion** es el nombre de la funcion, igual que para un procedimiento.
- **listaparametros** es la lista de parámetros que se define y funciona parecida a la de los procedimientos exceptuando el detalle de que todos los parámetros son de entrada y no se puede indicar ninguno de los calificadores `IN`, `OUT` o `INOUT`.
- **tipo** es el tipo del valor que devuelve la función. Este valor sustituirá la aparición de la función dentro de una expresión.
- **DETERMINISTIC** especifica que la función es determinista, esto es, que siempre que se le proporcione una combinación dada de valores de entrada producirá siempre el mismo valor de salida. Si no cumple estas condiciones hay que declararla como **NOT DETERMINISTIC**. Esto afecta a la forma en que MariaDB utiliza las funciones en las expresiones por lo que es importante el definirlo correctamente y de manera veraz, ya que MariaDB no comprueba que la función es o no determinística sino que se fía del programador. En general, funciones que se basan en información de tablas que pueden cambiar con el tiempo o que se basan en tiempos no suelen ser deterministas. Por ejemplo, una función que devuelve el día de la semana en que se invoca no es determinista. En cambio otra que devuelve el día de la semana que corresponde a una fecha dada si que lo es.
- **secuencia**. Secuencia de sentencias.

Hay que hacer notar que la misma restricción con el uso de los delimitadores que se tenía con los procedimientos se aplica a las funciones por lo que hay que utilizar el mecanismo de sustitución temporal del delimitador con la sentencia `DELIMITER`.

Para devolver el valor de retorno de la función se emplea la sentencia `RETURN`, con la siguiente sintaxis:

```
RETURN expresion;
```

donde **expresion** es la expresión que se evalúa y su valor se devuelve como valor de retorno de la función. Si no es del tipo especificado en la clausula `RETURN` se intenta convertir a dicho tipo, produciéndose un error si esta conversión no es posible. Además se termina la ejecución de la función inmediatamente.

EJEMPLO

Supongamos la tabla ya utilizada anteriormente:

empleado

codigo	nombre	edad	oficina	titulo	fecha_contrato	codigo_jefe	cuota_ventas	ventas_obtenidas
101	Antonio Viguer	45	12	Representante	20/10/06	104	300000	305000
102	Alvaro Jaumes	48	21	Representante	10/12/06	108	350000	474000
103	Juan Rovira	29	12	Representante	01/03/07	104	275000	286000
104	Jose Gonzalez	33	12	Dir Ventas	19/05/07	106	200000	143000
105	Vicente Pantall	37	13	Representante	12/01/08	104	350000	368000
106	Luis Antonio	52	11	Dir General	14/06/08	NULL	275000	299000
107	Jorge Gutierrez	49	22	Representante	14/11/08	108	300000	186000
108	Ana Bustamante	62	21	Dir Ventas	12/10/09	106	350000	361000
109	Maria Sunta	31	11	Representante	12/10/16	106	300000	392000
110	Juan Victor	41	22	Representante	13/01/10	104	500000	760000

Imaginemos que deseamos obtener un listado en que se quiere ver el nivel de rendimiento de los empleados, clasificándolos en:

- MUY BAJO Si sus ventas no llegan al 50% de su cuota
- BAJO Si sus ventas estan entre el 50 y 100% de la cuota.
- NORMAL Si sus ventas están en el 100% de la cuota.
- BUENO Si sus ventas están entre el 100% y el 150% de su cuota
- MUY BUENO Si sus ventas son superiores al 150% de su cuota.

Para ayudar en esta tarea se va a crear una función a la que se le proporciona la cuota y las ventas y devuelve la clasificación. Esta función se llamará `rating`:

```
DELIMITER //
CREATE FUNCTION rating (cuota DECIMAL(10,2), ventas DECIMAL(10,2))
RETURNS VARCHAR(10)
DETERMINISTIC

BEGIN
# La ratio es la relación entre las ventas y la cuota.
# Se declara como variable para que sólo haya que calcularlo una vez
    DECLARE ratio DECIMAL(10,2) DEFAULT 0;

    # El resultado es el valor que se va a devolver
    DECLARE clase VARCHAR(10);

    # Se calcula el ratio
    SET ratio = ventas / cuota;
```

```
    /* Se va a utilizar una secuencia de IF/ELSEIF/ELSE.
       Se podría haber hecho tambien con CASE */
    IF ratio < 0.5 THEN
         SET clase = 'MUY BAJO';
    ELSEIF ratio >= 0.5 AND ratio < 1 THEN
         SET clase = 'BAJO';
    ELSEIF ratio = 1 THEN
         SET clase = 'NORMAL';
    ELSEIF ratio > 1 AND ratio < 1.5 THEN
         SET clase = 'BUENO';
    ELSE
         SET clase = 'MUY BUENO';
    END IF;

    # Se devuelve el resultado
    RETURN clase;
END//
DELIMITER ;
```

10.5.- Uso de funciones

Las funciones se utilizan incorporándolas en expresiones de la misma forma en que se utilizan las funciones predefinidas.

EJEMPLO

Si se desea ver el ranking actual de los empleados con su calificación, se podría utilizar la sentencia:

```
SELECT nombre, rating(cuota, ventas) AS calificacion FROM empleado;
```

que devolvería el resultado:

nombre	calificacion
Antonio Viguer	BUENO
Alvaro Jaumes	BUENO
Juan Rovira	BUENO
Jose Gonzalez	BAJO
Vicente Pantall	BUENO
Luis Antonio	BUENO
Jorge Gutierrez	BAJO
Ana Bustamante	BUENO
Maria Sunta	BUENO
Juan Victor	MUY BUENO

10.6.- Eliminar una función

Una función se elimina utilizando la sintaxis:

```
DROP FUNCTION [IF EXISTS] nombre;
```

donde **nombre** es el nombre de la función a eliminar. A partir de ese momento la función ya no se puede volver a utilizar.

La opción `IF EXISTS` se utiliza para que la sentencia no lance un error si la función no existe.

EJEMPLO

Para eliminar la función `rating` ya creada se utilizaría:

```
DROP FUNCTION rating;
```

11.- Tratamiento de excepciones

El tratamiento de excepciones en MariaDB se realiza mediante la sintaxis:

```
DECLARE accion HANDLER FOR lista_errores sentencia;
```

donde,

- **accion** es la acción que MariaDB realizará *después* de procesar el error. Puede ser una de:
  - `CONTINUE`. La ejecución continuará por la sentencia siguiente a la que produjo el error.
  - `EXIT`. Se abortará la ejecución del bloque `BEGIN...END` en que se encuentre la declaración. Usualmente esto significa salir del procedimiento o función.
- **lista_errores** es una lista de los errores que el manejador que se está declarando debe procesar. Es una lista separada por comas de las especificaciones de los errores. Estas especificaciones pueden ser:
  - Un número entero, equivalente a un código de error interno de MariaDB. Para ver los códigos de error y su significado hay que consultar el manual de MariaDB. No utilizar el código 0 nunca.
  - Un codigo `SQLSTATE`. Estos códigos están definidos en el estándar SQL y son más interoperables entre servidores. Es la opción recomendada siempre que se pueda en lugar de los códigos internos de MariaDB. El código se da en la forma `SQLSTATE 'codigo'`, donde `codigo` es un número de 5 cifras siempre. No hay que utilizar los `SQLSTATEs` que comienzan por 00 ya que indican éxito, no error.
  - Un identificador. Este identificador debe corresponder con una condición de error definida por el usuario. En este documento no se tratarán las condiciones definidas por el usuario.

- SQLWARNING. Indica que se capturarán todos los errores de tipo Advertencia (WARNING). Estos errores se corresponden con SQLSTATEs que comienzan por 01 y suelen corresponder con condiciones de error que no son críticas y que permiten continuar con la ejecución del programa, aunque avisa de que algo merece revisarse porque puede ser fuente de problemas.
- NOT FOUND (dos palabras). Indica que se capturarán los errores de cursor (ver cursores más adelante en este documento). Se corresponde con los SQLSTATEs que comienzan por 02.
- SQLEXCEPTION. Indican que se capturaran los errores graves, cuyo SQLSTATE no comienza por 00, 01 ó 02. Suelen indicar una condición de error grave y que impide la continuación de la ejecución.

Para ver una lista de los códigos de error o SQLSTATEs para la versión 10.x del servidor, se puede consultar la página: https://mariadb.com/kb/en/mariadb-error-codes/

EJEMPLO

Una forma clásica de detectar cuando se utilizan cursores que se ha llegado al final de los resultados es utilizando una variable booleana (VERDADERO / FALSO) que se indica al programa que se ha llegado al final de los resultados. Esta variable inicialmente vale FALSO y se cambia a VERDADERO desde un manejador de excepciones de la forma siguiente:

```
# Se crea la variable y se inicializa a FALSO directamente
DECLARE hecho INTEGER DEFAULT FALSE;
/* Declaración del manejador de errores. Se declara de tipo CONTINUE porque no queremos detener la ejecución y la condición de error es NOT FOUND. La única sentencia que se ejecuta es SET hecho = TRUE que modifica el valor de la variable */
DECLARE CONTINUE HANDLER FOR NOT FOUND SET hecho = TRUE;
# Codigo que abre y usa el cursor
...
# Se comprueba que no se ha llegado al final. Si es asi, termina
IF hecho THEN .....
```

12.- Cursores

Para poder utilizar un cursor en MariaDB lo primero que hay que hacer es declararlo.

Para ello se utiliza la siguiente sintaxis:

```
DECLARE nombre CURSOR FOR sentencia_select;
```

donde,

- **nombre** es el nombre que se va a dar al cursor. Este nombre se utilizará para operar sobre el cursor.
- **sentencia_select**

  Sentencia SELECT completa. Cuando se active el cursor se utilizará para recorrer los resultados de esta consulta fila a fila.

Una vez declarado ya se puede utilizar. Hay que tener en cuenta que en MariaDB los cursores no se pueden actualizar, esto es, son sólo lectura y que no tienen vuelta atrás, por tanto el cursor se mueve de una fila a otra en una sola dirección, no pudiéndose volver a una fila ya visitada.

Una vez declarado el cursor hay que *abrirlo* para comenzar a utilizarlo. El proceso de apertura realiza efectivamente la consulta asociada al cursor y coloca éste en la primera fila, si es que la consulta devuelve alguna fila. Para abrir un cursor hay que utilizar la sentencia OPEN, con la sintaxis:

```
OPEN nombre;
```

donde **nombre** es el nombre de un cursor previamente declarado. Si el cursor no está declarado o la sentencia no es válida se producirá un error.

Una vez abierto, ya se puede acceder a información utilizando el cursor. Para ello se utiliza la sentencia FETCH, con la siguiente sintaxis:

```
FETCH nombre INTO var1, var2, ..., varN;
```

donde,

- **nombre** es el nombre de un cursor declarado y abierto.
- **var1, var2, varN** son variables donde se almacenarán los contenidos de la columna 1, 2, etc de la fila actual del cursor. Se pueden especificar tantas variables como columnas estén especificadas en la sentencia SELECT asociada al cursor.

Un efecto adicional de la ejecución de FETCH es que el cursor se desplaza a la siguiente fila de la tabla después de acceder a los campos de la actual, con lo que el cursor queda listo para otra lectura.

Por último, cuando se finaliza el trabajo con un cursor hay que *cerrarlo* para liberar los recursos empleados en el mismo. Si no se realiza esta operación, el cursor se libera automáticamente al finalizar el bloque BEGIN..END en el que está declarado. De todas formas se recomienda liberarlo explícitamente utilizando la siguiente sintaxis:

```
CLOSE nombre;
```

donde **nombre** es el nombre de un cursor ya declarado y abierto. Si no se dan estas dos condiciones, se producirá un error.

Con estas sentencias se manipulan los cursores pero aún queda pendiente el detalle de finalizar el proceso del cursor, esto es, ¿cuándo se sabe que se han procesado todas las filas accedidas a través del cursor?

La respuesta ya se dio en capítulos anteriores: A través de excepciones. Mas concretamente, el sistema lanza una excepción con el `SQLSTATE '02000'` para indicar que se llegó al final del resultado accedido por el cursor. Esto puede ocurrir desde la primera ejecución de la sentencia `FETCH` si el resultado obtenido no contiene ninguna fila.

Por lo tanto hay que declarar un manejador de excepciones que capture esta excepción en concreto y utilizarlo para terminar el proceso. La técnica más usual ya medio descrita anteriormente es utilizar una variable que se utiliza como *bandera*. La variable está en un estado inicialmente que indica que el proceso puede seguir y se modifica su valor desde el manejador de excepciones, indicando al procedimiento que el proceso debe finalizar. En el siguiente ejemplo completo de uso de cursores se describe un procedimiento típico que utiliza todas estas técnicas.

EJEMPLO

Supongamos que se tiene la tabla de ejemplo:

empleado					
codigo	**nombre**	**apellido1**	**apellido2**	**email**	**puesto**
1	Marcos	Magaña	Perez	marcos@jardineria.es	Director General
3	Alberto	Soria	Carrasco	asoria@jardineria.es	Subdirector Ventas
4	Maria	Solís	Jerez	msolis@jardineria.es	Secretaria
6	Juan Carlos	Ortiz	Serrano	cortiz@jardineria.es	Representante Ventas
8	Mariano	López	Murcia	mlopez@jardineria.es	Representante Ventas
9	Lucio	Campoamor	Martín	lcampoamor@jardineria.es	Representante Ventas
11	Emmanuel	Magaña	Perez	manu@jardineria.es	Director Oficina
12	José Manuel	Martinez	De la Osa	jmmart@hotmail.es	Representante Ventas
14	Oscar	Palma	Aceituno	opalma@jardineria.es	Representante Ventas
16	Lionel	Narvaez		lnarvaez@gardening.com	Representante Ventas
19	Walter Santiago	Sanchez	Lopez	wssanchez@gardening.com	Representante Ventas
21	Marcus	Paxton		mpaxton@gardening.com	Representante Ventas
23	Nei	Nishikori		nnishikori@gardening.com	Director Oficina
25	Takuma	Nomura		tnomura@gardening.com	Representante Ventas
28	John	Walton		jwalton@gardening.com	Representante Ventas
29	Kevin	Fallmer		kfalmer@gardening.com	Director Oficina
30	Julian	Bellinelli		jbellinelli@gardening.com	Representante Ventas
31	Mariko	Kishi		mkishi@gardening.com	Representante Ventas

y se desea crear un procedimiento que obtenga una lista de correo a partir de los correos de los empleados. Una lista de correo consta de los correos separados por punto y coma ";".

La función quedaría:

```
DELIMITER //
CREATE PROCEDURE listaCorreo(OUT lista VARCHAR(5000))
BEGIN
    # Variable bandera para marcar el fin del resultado
    DECLARE fin INTEGER DEFAULT FALSE;
```

```
    # Variable para leer el correo de cada fila
    DECLARE correo VARCHAR(100);
    # Variable para saber si la lista tiene más de un elemento
    # o no (para el proceso de separador)
    DECLARE masdeuno INTEGER DEFAULT FALSE;
    # Cursor
    DECLARE empleados CURSOR FOR SELECT DISTINCT email FROM empleado;
    # Manejador de errores para detectar el fin de resultad
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin = TRUE;

    # Comienza el proceso. Se abre el cursor
    OPEN empleados;
    # Y se inicializa la lista
    SET lista = '';
    # Ciclo que se repite hasta el fin de los resultados
    REPEAT
         # Se lee la fila
         FETCH empleados INTO correo;
         # Si no se llego al final del resultad
         IF NOT fin THEN
              # Si hay mas de uno
              IF masdeuno THEN
                   # Añade el separador ";"
                   SET lista = CONCAT(lista, ';');
              ELSE
                   # Ahora si hay mas de uno
                   SET masdeuno = TRUE;
              END IF;
              # Añade el elemento
              SET lista = CONCAT(lista, correo);
         END IF;
    UNTIL fin
    END REPEAT;
```

```
    # Cierra el cursor
    CLOSE empleados;
END//
DELIMITER ;
```

13.- Disparadores

En MariaDB, los disparadores o triggers se crean utilizando la siguiente sintaxis:

```
CREATE [OR REPLACE] TRIGGER nombre momento operacion ON tabla
FOR EACH ROW [orden] sentencia;
```

donde,

- La opción **OR REPLACE**, si se especifica, indica que se reemplace el disparador si ya existe uno con el mismo nombre. Si no se especifica y hay un disparador con ese nombre, la operación levanta un error y el disparador no se modifica.
- **nombre** es el nombre del disparador. Debe ser único entre los disparadores de una misma base de datos, aunque puede haber disparadores con el mismo nombre en distintas bases de datos.
- **momento** es el momento en que se dispara relativo a la operación que provoca el disparo. Puede ser uno de:
  - BEFORE El disparador se dispara antes de que se realice la operación.
  - AFTER El disparador se dispara después de que se realice la operación.
- **operacion** es la operación que provoca el disparo. Puede ser una de:
  - INSERT Se inserta un registro en la tabla.
  - UPDATE Se modifica un registro de la tabla.
  - DELETE Se elimina un registro de la tabla.
- **tabla** es la tabla sobre la que se crea el disparador. No se pueden crear disparadores sobre varias tablas. Hay que crear uno por cada tabla que se quiera monitorizar.
- **orden** es opcional y especifica, si hay más de un disparador para la misma operación y momento, cual de ellos debe ejecutarse primero. Si no se especifica, los disparadores se ejecutan en el mismo orden en que se definieron. La sintaxis es:
  - FOLLOWS otro_disparador. Indica que este disparador se debe lanzar después del disparador con nombre otro_disparador.
  - PRECEDES otro_disparador. Indica que este disparador se debe lanzar antes del disparador con nombre otro_disparador.

- `sentencia` es la sentencia que se ejecuta cuando se lanza el disparador. Si se necesita más de una sentencia hay que utilizar un bloque `BEGIN...END` como en los cuerpos de los procedimientos y funciones. Usualmente se utiliza una sola sentencia o se llama a un procedimiento.

Dado que un disparador se invoca en el momento que se va a producir un cambio o este ya se ha producido, es importante que la sentencia que procesa el disparador pueda acceder a los valores de los campos antes y después de que se modifiquen, a fin de saber la naturaleza de los cambios y actuar en consecuencia. Para ello se deben preceder los nombres de los campos con los alias `OLD.campo` y `NEW.campo` para indicar el valor del campo antes y después de la modificación, respectivamente. Si el campo no está afectado por la actualización, `OLD.campo` y `NEW.campo` tendrán el mismo valor.

EJEMPLO

Supongamos una tabla con la siguiente definición:

```
CREATE TABLE usuario(id INTEGER PRIMARY KEY, nombre VARCHAR(50) NOT NULL, puntos INTEGER DEFAULT 0)
```

que contiene información sobre un usuario de una plataforma on-line. El usuario puede obtener puntos participando y los puede gastar. Se decide que el saldo de puntos debe estar siempre entre 0 y 100. Para asegurar esta regla se puede realizar un disparador que asegure su cumplimiento. El disparador se crearía de la siguiente forma:

```
# Para poder hacer el bloque BEGIN..END
DELIMITER //
CREATE TRIGGER comp_saldo BEFORE UPDATE ON usuario FOR EACH ROW
BEGIN
    # Si el nuevo saldo es negativo
    IF NEW.saldo < 0 THEN
         # Lo ajusta a cero
         SET NEW.saldo = 0;
    # Si no es negativo comprueba que no sea mayor que 100
    ELSEIF saldo > 100 THEN
         # Si lo es, lo recorta a 100
         SET NEW.saldo = 100;
    END IF;
    # Si el saldo no es ni negativo ni mayor que 100 no se modifica
END;//
DELIMITER ;
```

Para eliminar un disparador, la sintaxis es muy simple:

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

donde **nombre** es el nombre del disparador a eliminar. Si se especifica la opción `IF EXISTS` y no existe un disparador con el nombre dado no se levantaría un error, cosa que si ocurriría en caso de que no se especifique la opción.

EJEMPLO

Para eliminar el disparador creado en el ejemplo anterior se utilizaría:

```
DROP TRIGGER comp_saldo;
```

14.- Temporizadores

En MariaDB, los temporizadores se denominan *eventos*. Para crear un evento se emplea la siguiente sintaxis:

```
CREATE [OR REPLACE] EVENT [IF NOT EXISTS] nombre
ON SCHEDULE tiempo [ON COMPLETION PRESERVE] [activacion] DO sentencia;
```

donde,

- La opción **`OR REPLACE`** tiene el efecto habitual. NO se puede usar junto a la opción `IF NOT EXISTS`, que también tiene el efecto habitual.
- **nombre** es el nombre del evento y debe ser único entre los eventos de la base de datos.
- **tiempo** es la definición de cuándo se va a activar el temporizador.

  Se puede indicar de dos formas:

  - `AT timestamp`.

    Donde `timestamp` es una fecha y hora concretos en la que se realizará el disparo.

  - `EVERY intervalo [STARTS timestamp]`.

    Donde `intervalo` es un intervalo de tiempo que se puede expresar en unidades de tiempo desde microsegundos a años (ver https://mariadb.com/kb/en/date-and-time-units/). En este caso es necesario añadir la clausula `ON COMPLETION PRESERVE` para que el temporizador no se elimine después del primer disparo.

    Opcionalmente se puede añadir la opción `STARTS` proporcionando un valor de tipo `timestamp`. Esta opción indica el momento del primer disparo, repitiéndose ya de forma periódica a partir de ese momento. Si no se especifica, el evento comienza a repetirse a partir del momento en que es creado.

- `activacion`. Indica si el temporizador se activa inmediatamente cuando es creado (el comportamiento por defecto) o se crea desactivado (se puede activar más tarde). Puede valer:
  - `ENABLE`. Para que el temporizador se cree activado (la opción por defecto).
  - `DISABLE`. Para que el temporizador se cree desactivado.
- `sentencia`. Es la sentencia que se ejecutará cuando se dispare el evento. Del mismo modo que ocurre con los disparadores, si se desea realizar algo que necesite más de una sentencia, hay que utilizar un bloque `BEGIN..END` o utilizar llamadas a procedimientos almacenados.

Para que funcionen los eventos en MariaDB debe estar activado un módulo del SGBD denominado planificador de eventos (event scheduler). El estado del planificador se controla mediante la variable `event_scheduler`. Esto se puede realizar de dos maneras:

- **Por configuración**

  Mediante un ajuste en el archivo `my.cnf`, que controla la inicialización del SGBD (introduciendo una línea con la sintaxis: `event_scheduler=ON`)

- **Desde comandos**

  Modificando la variable utilizando la sentencia SET (`SET GLOBAL event_scheduler="ON"`)

EJEMPLO

Supongamos que la tabla `usuarios` dispone de varios campos para el nombre de usuario, contraseña, etc y que además, para tareas de control de cuentas no usadas, se dispone de un campo `ultimoAcceso`, de tipo `TIMESTAMP` que contiene la fecha y hora del último acceso del usuario. Para crear un evento que se dispare el día 1 de Enero a medianoche y que elimine las cuentas que no se han accedido en 6 meses, la sentencia sería:

```
CREATE EVENT limpia_cuentas_2018 ON SCHEDULE AT '2018-01-01 00:00:00'
DO DELETE FROM usuarios WHERE ultimoAcceso < '2017-06-30 23:59:59';
```

Si se deseara realizar esta misma tarea pero cada mes, habría que utilizar:

```
CREATE EVENT limpia_cuentas_mensual ON SCHEDULE EVERY 1 MONTH
ON COMPLETION PRESERVE DISABLE
DO DELETE FROM usuarios WHERE DATEDIFF(NOW(), ultimoAcceso) > 180;
```

En este último cambia la condición del borrado porque debe funcionar correctamente en todas las ejecuciones. Se comprueba que no hayan pasado más de 180 días desde el último acceso. Asimismo se ha creado el evento sin activar a fin de activarlo justo a comienzos de mes.

Para eliminar un temporizador de forma manual se utiliza la sentencia `DROP EVENT`, con la sintaxis:

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

donde `nombre` es el nombre de un evento. La clausula `IF EXISTS` previene que ocurra un error si el evento no existe.

EJEMPLO

Para eliminar el evento `limpia_cuentas_2018` se utilizaría la sentencia:

```
DROP EVENT IF EXISTS limpia_cuentas_2018;
```

Por último, para modificar un evento existente, la sintaxis es:

```
ALTER EVENT nombre [ON SCHEDULE tiempo] [RENAME TO nuevo_nombre] [activacion]
[DO sentencia];
```

donde,

- `nombre` es el nombre del evento existente que se desea modificar.
- `tiempo` es el nuevo momento o periodo de activación.
- `nuevo_nombre` es el nuevo nombre del evento. No debe existir un evento con ese nombre en la base de datos.
- `activacion`. Nuevo estado de la activación del evento.
- `sentencia`. Nuevo comando del evento.

Todos los apartados son opcionales. Si alguno no se especifica, su valor no se modifica. Los parámetros `tiempo`, `activacion` y `sentencia` se especifican como se describió en la sentencia `CREATE EVENT`.

EJEMPLO

Supongamos que se desea activar ya el evento `limpia_cuentas_mensual`.

Se haría con la sentencia:

```
ALTER EVENT limpia_cuentas_mensual ENABLE;
```