

3. Programación de bases de datos

3.1. Programación de bases de datos

Una práctica habitual en el manejo de los datos almacenados en las bases de datos es tener diseñadas aplicaciones con una interfaz fácil de manejar para ser utilizadas por usuarios que no tengan conocimientos suficientes de SQL. El diseño de las aplicaciones y de la base de datos se puede hacer en paralelo.

Por ejemplo, un sitio Web que permite ver información de las actividades culturales de una ciudad y que tiene una base de datos para almacenar la información relativa a los usuarios que pueden acceder a los contenidos restringidos de la web, a las salas de exposición, teatros, y cines de la ciudad, así como a la información del programa de actividades. Los usuarios manejan la información a través de formularios HTML, con un botón de envío que cuando se pulsa hace una petición al servidor de la base de datos, para ejecutar una sentencia INSERT, UPDATE, DELETE o SELECT. Las aplicaciones que utilizan los usuarios estarían escritas en lenguaje son como PHP o Java.

Casi todos los SGBDR incorporan utilidades que permiten ampliar el lenguaje SQL para poder crear y almacenar en el servidor pequeños programas que utilizan instrucciones propias de la programación procedimental (manejo de variables, instrucciones condicionales, bucles, etc.). De esta forma, el administrador de la base de datos puede crear pequeños programas para automatizar algunas tareas sobre las bases de datos, que pueden ser utilizados por los usuarios, o por los programadores de aplicaciones que los llaman desde sus programas.

MySQL incorpora un conjunto de sentencias con las funciones propias de los lenguajes procedimentales (manejo de variables, instrucciones condicionales, bucles, manejo de excepciones, ...).

3.1.1. Programas almacenados y rutinas almacenadas

Las ampliaciones del lenguaje SQL permiten crear y almacenar en el lado del servidor de bases de datos varios tipos de objetos:

- Procedimientos almacenados. Conjunto de sentencias que permiten automatizar tareas que pueden hacer cálculos, o generar conjuntos de resultados.
- Funciones definidas por los usuarios (UDF). Conjunto de sentencias que devuelven siempre el resultado de un cálculo, y pueden utilizarse en expresiones, al igual que las funciones propias de SQL.
- Disparadores (triggers). Conjunto de sentencias que se ejecutan de forma automática cuando se hace una determinada operación de manipulación de datos.
- Eventos. Permiten la ejecución diferida de un conjunto de sentencias, teniendo en cuenta un calendario establecido.

Algunos conceptos que debemos conocer:

Programas almacenados: Este término hace referencia a todos los tipos de objetos almacenados en el servidor: procedimientos, funciones, disparadores, y eventos.

Rutinas almacenadas: Este término sólo hace referencia a los procedimientos almacenados y las funciones. Estos objetos se definen con un sintaxis muy similar, y su ejecu-

ción no es automática como sucede con los disparadores y los eventos.

Subrutina: Este término se emplea para hacer referencia a una rutina utilizada dentro de un programa almacenado.

3.1.2. Ventajas de utilizar programas almacenados en el servidor de bases de datos

- **Seguridad:** Los procedimientos ocultan los nombres de las tablas a usuarios que no tengan privilegios para manipular los datos. Estos usuarios simplemente llaman a los procedimientos sin conocer la estructura de la base de datos. Los bancos, por ejemplo, utilizan los procedimientos almacenados para todas las operaciones estándar, y no dejan el acceso a las tablas en manos de los programadores de las aplicaciones.
- **Estándares de código:** Un equipo de programadores puede compartir programas almacenados en el servidor de bases de datos. En el caso de que cada programador haya creado su propio código para realizar una misma tarea, podrían existir problemas de integridad, además de la pérdida de tiempo.
- **Velocidad:** Los programas almacenados se almacenan en el lado del servidor, de manera que el código necesario para definirlos se envía a la red sólo cuando se crea el programa y no cuando se ejecuta, lo que reduce la carga en la red, aunque aumenta la carga en el servidor de bases de datos.
- **Control de errores:** Proporcionan un mecanismo para control de errores gestionado por el administrador de la base de datos. Permiten al administrador llevar un registro de errores y de acceso a los datos, distinto del que proporcionan los archivos de registro (log) del servidor.

La principal desventaja de la utilización de programas almacenados en SQL es que los lenguajes procedimentales de distintos fabricantes no son compatibles entre ellos.

3.1.3. Creación de programas almacenados

Para crear un programa almacenado en SQL es necesario escribir un guión (*script*) de sentencias SQL utilizando un editor y guardarlo en un fichero con extensión .sql.

La ejecución del guión de sentencias SQL realiza la compilación del programa almacenado y la creación del objeto correspondiente (procedimiento, función, disparador, o evento) en el servidor.

Las sentencias para crear, borrar y modificar programas almacenados son:

CREATE PROCEDURE	DROP PROCEDURE	ALTER PROCEDURE
CREATE FUNCTION	DROP FUNCTION	ALTER FUNCTION
CREATE TRIGGER	DROP TRIGGER	
CREATE EVENT	DROP EVENT	ALTER EVENT

3.1.4. Bloques de programación

Los programas almacenados pueden contener una o más sentencias para hacer su trabajo. En el caso de contener varias sentencias, estas se pueden agrupar en bloques de programaciones, también llamados sentencias compuestas, que son un conjunto de sentencias SQL que resuelven un problema concreto, y que empiezan con una sentencia *begin*, y terminan con una sentencia *end*. Un bloque puede estar contenido en otro bloque. Sintaxis de la creación de bloques de programación:

```
[etiqueta_inicio:] BEGIN
    [lista_sentencias]
END [etiqueta_fin]
```

- La parte *lista_sentencias* es una lista de una o más sentencias SQL.
- Si el bloque o programa se compone de una única instrucción no es obligatorio utilizar las sentencias *begin* y *end*.
- Cada sentencia que forma el bloque tiene que acabar en punto y coma (;), que es el carácter delimitador que indica el final de una sentencia SQL.
- Las etiquetas de inicio y de final, que son optativas, se pueden utilizar para hacer referencia a ese bloque desde cualquier parte del programa almacenado.

3.1.5. Delimitadores de final de sentencia

Un problema del uso de sentencias compuestas es que dentro del bloque es necesario separar las sentencias con el delimitador de final de sentencia, que de forma predeterminada es el carácter punto y coma. Esto produce un conflicto cuando se crean programas almacenados ya que en el caso de que el servidor reciba el primer carácter punto y coma considera que la sentencia de creación del programa almacenado termina, y no tendría en cuenta las siguientes instrucciones que componen el programa almacenado.

Para solucionar este conflicto hay que utilizar la sentencia `DELIMITER` que permite cambiar el carácter delimitador de final de sentencia por un carácter o combinación de caracteres que no se utilice en la creación del programa almacenado (por ejemplo `//`). Una vez creado el programa se puede utilizar esta misma instrucción para volver a definir el punto y coma como delimitador de final de sentencia. Sintaxis:

```
DELIMITER carácter_final_sentencia
```

Ejemplo de guión para crear un procedimiento almacenado:

<code>-- Hola mundo, en SQL</code>	
<code>delimiter //</code>	<code># Cambia el caracter delimitador de fin de sentencia</code>
<code>create procedure holaMundo()</code>	<code># Le da nombre al procedimiento</code>
<code>begin</code>	<code># Inicia el bloque de programación</code>
<code> select 'hola mundo';</code>	<code># Sentencias del bloque de programación</code>
<code>end;</code>	<code># Finaliza el bloque de programación</code>
<code>//</code>	<code># Finaliza la sentencia create procedure</code>
<code>delimiter ;</code>	<code># Deja el delimitador de fin de sentencia como estaba</code>

3.1.6. Parámetros

Algunas rutinas almacenadas necesitan que se les proporcione algún dato para poder hacer su trabajo; estos datos se llaman parámetros de entrada. Además, puede ocurrir que la rutina devuelva algún valor una vez finalizada a la ejecución; estos datos que devuelve la rutina se llaman parámetros de salida. Para diferenciarlos de las variables y de los nombres de columnas puede ser útil ponerles nombres que empiecen por *p*. Ejemplos: *pDNI*, o *pnombre*.

3.1.7. Ejecución de programas almacenados

Una de las diferencias más importantes en los distintos tipos de programas almacenados es la forma en que se ejecutan:

- Procedimiento almacenado. Se ejecuta haciendo una llamada (*call*) al servidor indicando el nombre del procedimiento y pasándole, opcionalmente, los parámetros necesarios.
- Función definida por el usuario. Se utiliza igual que las funciones que ya existen en SQL. No se ejecuta con una llamada explícita como los procedimientos, si no que se utiliza como parte de una expresión en las sentencias SQL.
- Disparador. Está asociado siempre a una operación de manipulación de datos (inserción, modificación o borrado de filas) sobre una tabla, y se ejecuta de forma automática cuando se realiza esa operación.
- Evento. En la creación del evento se indica en qué momento se tiene que ejecutar, y esa información queda almacenada en el servidor; cuando llega ese momento, el servidor lo ejecuta de forma automática.

Ejemplo de ejecución de un procedimiento almacenado:

```
call holaMundo();
```

Resultado de la ejecución:



	hola mundo
▶	hola mundo

3.2. Sentencias básicas para la programación con MySQL

MySQL sólo permite utilizar la mayoría de las sentencias básicas de programación en la creación de programas almacenados y no lo permite fuera de este contexto. Otros SGBDR no tienen esta limitación.

La prueba de las sentencias básicas de programación con MySQL de este documento, se hará creando un procedimiento almacenado sencillo con un único bloque de programación, similar a lo que se muestra en el apartado "Creación de programas almacenados".

3.2.1. Sentencias de declaración y manejo de variables y manipuladores

DECLARE

Permiten declarar variables, condiciones de error y manipuladores de errores. Las sentencias de declaración tienen que ir siempre al inicio del bloque de programación, después de la sentencia *begin*, y antes de escribir cualquier otra sentencia. Las sentencias de declaración hay que escribirlas en este orden: primero las variables, después las condiciones, y por último los manipuladores.

Declaración de variables

Permite definir variables locales del programa. Sintaxis:

```
DECLARE nombre_variable [,...] tipo_dato [DEFAULT valor]
```

- El tipo de dato puede ser cualquiera de los utilizados en la definición de columnas en las tablas.
- La cláusula DEFAULT permite asignarle un valor a la variable en el momento de la creación.

Declaración de condiciones

Permite asignar un nombre a una condición de error relacionada con un determinado código de error, o con un estado SQL (*sqlstate*). Se utiliza para condiciones de error que precisan de un tratamiento especial. Los nombres de condición pueden utilizarse en la declaración de manipuladores. Sintaxis:

```
DECLARE nombre_condición CONDITION FOR valor_condición
```

Donde *valor_condición* puede ser:

```
SQLSTATE [VALUE] codigo_sqlstate | código_error_mysql
```

Los códigos de error y valores de estado (SQLSTATE) junto con la explicación de su significado, se pueden consultar en el siguiente enlace:

<https://dev.mysql.com/doc/refman/8.0/en/problems.html>

Ejemplo de declaración de condición:

```
-- Haciendo referencia al código de estado (entre comillas)
declare claveDuplicada condition for sqlstate '23000';
-- También podría hacer referencia al código de error MySQL (sin comillas)
declare claveDuplicada2 condition for 1062;
```

La comprobación de la sintaxis de las órdenes de declaración de condición, puede hacerse creando un procedimiento almacenado con un único bloque de programación:

```
drop procedure if exists condicion;
delimiter //
create procedure condicion()
begin
-- Haciendo referencia al código de estado (entre comillas)
declare claveDuplicada condition for sqlstate '23000';
-- También podría hacer referencia al código de error MySQL (sin comillas)
declare claveDuplicada2 condition for 1062;
end;
//
delimiter ;
```

Declaración de manipuladores. Manejo de excepciones

Indica las acciones a ejecutar en el caso que se produzca un error determinado en la ejecución de un programa almacenado. Sintaxis:

```
DECLARE acción_manipulador HANDLER FOR valor_condición [,...] lista_sentencias
```

- El valor para *acción_manipulador* puede ser: CONTINUE | EXIT
 - La opción EXIT, se da para errores graves, y produce la interrupción de la ejecución del bloque de programación en el momento que se produce el error.
 - La opción CONTINUE, se utiliza para errores leves que permiten que la ejecución pueda continuar y que en el propio código se decidan las acciones a tomar en el caso de que se produzca el error.
- El *valor_condición* especifica la condición o clase de condiciones que activan el manipulador, y puede ser: SQLSTATE [VALUE] código_sqlstate | código_error_mysql | nombre_condición | SQLWARNING | NOT FOUND | SQLEXCEPTION
- La parte *lista_sentencias* puede contener una o más sentencias SQL a ejecutar en el caso de producirse la condición de error asociada al manipulador. En el caso de tener más de una sentencia hay que utilizar BEGIN y END.

La acción asociada a un manipulador está en función de la clase de condición que tiene asociada. En el caso de SQLEXCEPTION debería ser EXIT, y en el caso de SQLWARNING y NOT FOUND debería ser CONTINUE.

La declaración de manipuladores tiene que ir siempre en un bloque de programación, situada después de la declaración de variables y de condiciones, y antes de empezar cualquier otra sentencia distinta de las sentencias de declaración.

Ejemplo de declaración de un manipulador que utiliza el nombre de condición *claveDuplicada*, como lo creado en el ejemplo anterior, que cuando se produce el error asociado a esa condición, le asigna el valor 1 a la variable *vFinal*:

```
declare vFinal bit default 0;  
declare claveDuplicada condition for sqlstate '23000';  
declare continue handler for claveDuplicada set vFinal = 1;
```

La comprobación de la sintaxis de las órdenes de declaración y el funcionamiento de los manipuladores, puede hacerse creando un procedimiento almacenado con un único bloque de programación en el que se declara la condición, el manipulador, y una variable tipo interruptor llamada *vFinal* que cuando se crea toma el valor 0, y cuando se produce la condición de error se le asigna el valor 1:

```
drop procedure if exists manipuladorDemo;  
delimiter //  
create procedure manipuladorDemo()  
begin  
declare vFinal bit default 0;
```

```

declare claveDuplicada condition for sqlstate '23000';
declare continue handler for claveDuplicada set vFinal = 1;
insert into utilidades. provincia values ('27','Lugo');
select 'Primer intento ',vFinal;
insert into utilidades. provincia values ('27','Lugo');
select 'Segundo intento ',vFinal;
end;
//
delimiter ;
call manipuladorDemo();

```

Si no existe ninguna provincia con el código 27 la variable *vFinal* toma el valor 0 y en el caso de que ya exista una provincia con ese código toma el valor 1. Resultado de la ejecución:

Result Grid	Filter Rows:	Export:
numero_intento	vFinal	
Primer intento	0	

Result Grid	Filter Rows:	Export:
numero_intento	vFinal	
Segundo intento	1	

SET

Permite asignar valores a las variables que se manejan en el procedimiento. La sintaxis es:

```
SET nombre_variable = expresión [,nombre_variable = expresión] ...
```

Tipos de variables que se pueden utilizar en los programas almacenados:

- **Variables locales.** Se presentan dentro de un bloque de código SQL con la sentencia *declare*. Su valor sólo puede verse dentro del bloque y se elimina cuando termina el bloque. Su nombre no lleva ningún carácter especial al inicio, aunque para diferenciarlas de los nombres de columnas se recomienda que empezabacien por una letra 'v'. Ejemplo: *set vNumero = 0*.
- **Variables de usuario.** Pueden crearse, verse y modificarse dentro o fuera del procedimiento y son visibles mientras no se cierre la sesión del usuario. Para crearlas basta con asignarle un valor con el orden *set*, y no es necesario declararlas previamente. Su nombre lleva delante el símbolo @. Ejemplo: *set @mes = 3*.
A este tipo de variables también se le pueden asignar valores utilizando el operador *:=* en una sentencia SELECT. Ejemplo: *select @numero := @numero + 1*
- **Variables de sistema.** Son variables que maneja el SGBD para configurar el servidor. Se crean y se les asigna un valor en el momento de iniciar el servidor. Se puede cambiar su valor a nivel *global* o *session*. Cuando se asigna el valor a nivel *global*, ese será el valor que tenga la variable para todas las sesiones que se inicien a partir de ese momento, y cuando se asigna a nivel *session* el valor sólo cambia para la sesión actual. Su nombre lleva delante dos símbolos @. Fuera de un bloque de programación no es nece-

sario poner delante los dos símbolos @ a los nombres de las variables de sistemas; se hace dentro de los programas almacenados para distinguirlas de las variables locales y de usuario. Ejemplo: *set session @@foreign_key_checks = 0*.

SELECT ... INTO

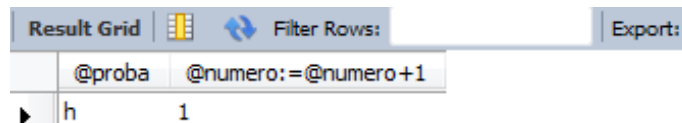
Es otra manera de asignar valores a las variables, tomando como entrada el resultado de una sentencia *select*. Permite almacenar en una variable los valores de las columnas del resultado de una consulta que sólo devuelve una fila. Sintaxis:

```
SELECT nombre_columna [,...] INTO nombre_variable [,... ]
```

Ejemplo de asignación de valores con la sentencia SELECT:

```
set @proba = 'x';
select @numero:=0;
select sexo into @proba from practicas.empleado where dni='33258458K';
select @proba,@numero:=@numero+1;
```

Resultado de la ejecución:



	@proba	@numero:=@numero+1
▶	h	1

3.2.2. Estructuras de control de flujo

SQL permite utilizar, como cualquier lenguaje de programación, una serie de sentencias de control de flujo para poder escribir rutinas que tengan cierta complejidad.

Sentencia condicional IF

Permite seleccionar qué sentencias ejecutar en función de una condición.

```
IF condicion1 THEN lista_sentencias1
  [ELSEIF condicion2 THEN lista_sentencias2] ...
  [ELSE lista_sentencias3]
END IF
```

El resultado es:

- Si la *condicion1* es verdadera, se ejecutan las sentencias contenidas en *lista_sentencias1*.
- Si la *condicion1* es falsa,
 - y existe la parte *else*, se ejecutan las sentencias contenidas en *lista_sentencias3*.

- y existe la parte *elseif*, se evaluaría la *condicion1*: si es verdadera, se ejecuta la sentencia o sentencias contenidas en *lista_sentencias2*.
- y no existe *else* ni *elseif*, se ejecuta la sentencia que aparezca después de END IF.

Está permitido el uso de sentencias *if* anidadas, es decir, una sentencia *if* puede contener otras sentencias *if*. En este caso cada sentencia *if* tiene que llevar el correspondiente *end if*.

Ejemplo que muestra si la cantidad almacenada en la variable *vNumero* es par y además múltiplo de 10, o par y además no múltiplo de 10, o es impar.

```
drop procedure if exists evaluaNumero;
delimiter //
create procedure evaluaNumero()
begin
declare vNumero integer;
set vNumero = 92;
if vNumero % 2 = 0 then
    if vNumero % 10 = 0 then select 'Número par e múltiplo de 10';
    select 'Número par que non é múltiplo de 10';
end if;
else select 'Número impar';
end if;
end ;
//
delimiter ;
call evaluaNumero();
```

Resultado de la ejecución:

Result Grid	Filter Rows:	Export:	Wrap Cell Content:
Número par que non é múltiplo de 10			

Sentencia alternativa CASE

Permite ejecutar una lista de sentencias, dependiendo del valor que toma una expresión.

```
CASE expresión
    [WHEN valor THEN lista_sentencias1] ...
    [ELSE lista_sentencias2]
END CASE
```

Evalúa el resultado que devuelve la expresión y cuando toma el valor que va después de la cláusula *when*, ejecuta las instrucciones contenidas en *lista_sentencias1*. Se pueden poner las cláusulas *when* que se quieran, y además, se puede utilizar la cláusula *else* para que se ejecute la *lista_sentencias2* en el caso de que la expresión tome un valor diferente de los relacionados en las cláusulas *when*.

Está permitido utilizar esta sentencia fuera de la creación de programas almacenados. Ejemplo de uso dentro de una consulta:

```
-- Ejemplo de uso en una consulta con sentencia select
select dni, nombre,
       case sexo
       when 'h' then 'hombre'
       when 'm' then 'muller'
       end as sexo
from empleado;
```

Ejemplo dentro de un bloque de programación:

```
-- Ejemplo de uso dentro de un bloque de programación
delimiter //
create procedure demoCase()
begin
declare vSexo char(1) default null;
set vSexo = 'm';
casi vSexo
when 'h' then select 'hombre';
when 'm' then select 'muller';
else select 'error';
end case;
end;
//
delimiter ;
call demoCase();
drop procedure demoCase;
```

Resultado de la ejecución:



Result Grid	
	muller
▶	muller

Sentencia repetitiva bucle WHILE

Repite un bloque de sentencias, mientras se cumpla una condición.

```
[etiqueta_inicio:] WHILE condición DO
    lista_de_sentencias
END WHILE [etiqueta_fin]
```

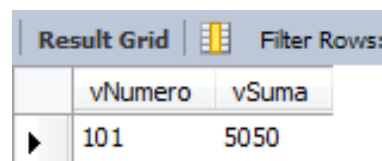
Las sentencias incluidas en *lista_de_sentencias* se van a ejecutar un número indeterminado de veces, mientras la condición que va después de WHILE sea verdadera, y dejarán de ejecutarse cuando la condición sea falsa.

Ejemplo que calcula la suma de los 100 primeros números naturales empleando un bucle *while*:

```
-- suma de los 100 primeros números naturales
drop procedure if exists suma100;
delimiter //
create procedure suma100()
begin
declare vNumero tinyint default 1;
declare vSuma smallint unsigned default 0;
bucle: while (vNumero <= 100) del
set vSuma = vSuma + vNumero;
set vNumero = vNumero+1;
end while bucle;
select vNumero, vSuma;
end;
//
delimiter ;
call suma100();
```

El bucle lleva un nombre de etiqueta para mostrar cómo es la sintaxis de etiquetas, aunque en este ejemplo no tiene ninguna utilidad. El uso de etiquetas tiene utilidad cuando se necesita hacer referencia al bucle en alguna parte del programa.

Resultado de la ejecución:



Result Grid		Filter Rows:
	vNumero	vSuma
▶	101	5050

Sentencia repetitiva bucle REPEAT

Repite una o más sentencias, y para de ejecutarlas cuando se cumple una condición.

```
[etiqueta_inicio:] REPEAT
lista_de_sentencias
UNTIL condición
END REPEAT [etiqueta_fin]
```

Las sentencias incluidas en *lista_de_sentencias* se van a ejecutar un número indeterminado de veces, hasta que la condición que va después de UNTIL sea verdadera.

Ejemplo que calcula la suma de los 100 primeros números naturales empleando un bucle *repeat..until*.

```

-- suma de los 100 primeros números naturales
drop procedure if exists suma100repeat;
delimiter //
create procedure suma100repeat()
begin
declare vNumero tinyint default 1;
declare vSuma smallint unsigned default 0;
repeat
set vSuma = vSuma + vNumero;
set vNumero = vNumero+1;
until vNumero > 100
end repeat;
select vNumero,vSuma;
end;
//
delimiter ;
call suma100repeat();

```

Resultado de la ejecución:

Result Grid		Filter Rows:
	vNumero	vSuma
▶	101	5050

3.2.3. Sentencias preparadas en SQL

La utilización de sentencias preparadas en programas almacenados y en las aplicaciones, permiten introducir parámetros en la construcción de las sentencias, que son sustituidos por los valores que se les asignan en el momento de la ejecución. Esto supone menos sobrecarga para analizar la sentencia cada vez que se ejecuta.

Normalmente las aplicaciones de bases de datos manejan muchas sentencias casi idénticas, con pequeños cambios en valores constantes o variables en cláusulas como: *where* para el caso de consultas, *set* en el caso de modificaciones, o *values* en el caso de inserciones.

La sintaxis SQL para sentencias preparadas se basa en la utilización de tres sentencias:

- La sentencia **PREPARE** permite preparar una sentencia y asignarle un nombre para hacer referencia a ella posteriormente para ejecutarla o borrarla. Sintaxis:

```
PREPARE nombre_sentencia_preparada FROM sentencia_preparada;
```

- La sentencia **EXECUTE** permite ejecutar una sentencia preparada. Sintaxis:

```
EXECUTE nombre_sentencia_preparada [USING @nombre_variable [,@nombre_variable]
...];
```

- La sentencia DEALLOCATE o DROP permite eliminar una sentencia preparada. Sintaxis:

```
{DEALLOCATE | DROP} PREPARE nombre_sentencia_preparada;
```

Ejemplo de sentencia preparada:

- Crear y ejecutar una sentencia preparada que muestre la suma de dos números que están guardados en las variables de usuario *n1* y *n2*.

```
-- Mostrar la suma de dos números contenidos en variables de usuario
```

```
use test;
```

```
set @n1=9;
```

```
set @n2=4;
```

```
prepare suma from 'select ? + ? as suma';
```

```
execute suma using @n1,@n2;
```

```
deallocate prepare suma;
```

Resultado de la ejecución:

Result Grid		Filter Rows:	Export:
	suma		
▶	13		