

BBDD

Bases de Datos – Programación de bases de datos



Objetivos

- Definir y utilizar guiones para la automatización de tareas.
- Identificar los recursos de programación (Gestión de variables, comentarios, estructuras de control)
- Identificar y diferenciar los distintos tipos de encapsulamiento de código y su aplicación: bloques anónimos, procedimientos almacenados, funciones de usuario, eventos, Triggers.
- Desarrollar procedimientos almacenados, evaluando y utilizando las sentencias del lenguaje incorporado en el SGBD.
- Desarrollar funciones de usuario, evaluando y utilizando las sentencias del lenguaje incorporado en el SGBD.



Contextualización

La **programación de Bases de Datos** está planificada para desarrollarse durante la 3ª evaluación y no antes dada la interdisciplinaridad del módulo con otro módulo (**Programación**).

Se considera que el alumnado ha podido adquirir ya los fundamentos básicos de programación impartidos en el módulo **Programación** tales como:

- Encapsulamiento de código en clases y métodos
- Gestión de variables, operadores y tipos elementales de datos



Interdisciplinaridad

Durante el desarrollo de las UD's 11 y 12, **programación de Bases de Datos**, realizaremos prácticas transversales a los módulos de Programación y Bases de Datos, que serán tenidas en cuenta en la evaluación de ambos módulos.



Programación



Bases de Datos



- Los **guiones** son un conjunto de instrucciones empleando el lenguaje de programación que incorpora el gestor relacional y que tiene como principal diferencia con respecto a otros lenguajes, que vamos a poder emplear **órdenes SQL como parte de su código**.
- Todas las instrucciones serán ejecutadas secuencialmente en un gestor de bases de datos relacional, en nuestro caso, MySQL.
- En un guion, entre otras instrucciones, podemos incluir:
 - Consultas
 - Procedimientos
 - Funciones
 - Operaciones de manipulación de datos
 - Órdenes SQL asociadas a los permisos y seguridad.
- En general, un guion siempre se distribuirá en un fichero de texto **con extensión .sql** editable desde cualquier editor (ultraedit, Notepad ++, bloc de notas, etc).
- A este conjunto de instrucciones se le suele denominar también **archivos de procesamiento por lotes** y se emplean para distribuir bases de datos tanto estructura como datos, encapsulamientos de código como procedimientos, funciones, triggers, eventos, etc.

Ejemplo de guion MySQL

```
DROP SCHEMA IF EXISTS sakila2;
CREATE SCHEMA sakila2;
USE sakila2;
CREATE TABLE actor ( actor_id SMALLINT
UNSIGNED NOT NULL AUTO_INCREMENT,
first_name VARCHAR(45) NOT NULL, last_name
VARCHAR(45) NOT NULL, last_update TIMESTAMP
NOT NULL PRIMARY KEY (actor_id));

SET AUTOCOMMIT=0;
INSERT INTO actor VALUES
(1,'PENELOPE','GUINNESS','2006-02-15 04:34:33');
INSERT INTO actor VALUES
(2,'NICK','WAHLBERG','2006-02-15 04:34:33');
COMMIT;

CREATE TRIGGER rental_date BEFORE INSERT ON
rental      FOR EACH ROW SET NEW.rental_date
= NOW();
```

- Sintaxis de ejecución de un guion:

`mysql < guion.sql`

`source guion.sql`



Guiones: Actividad coevaluuable

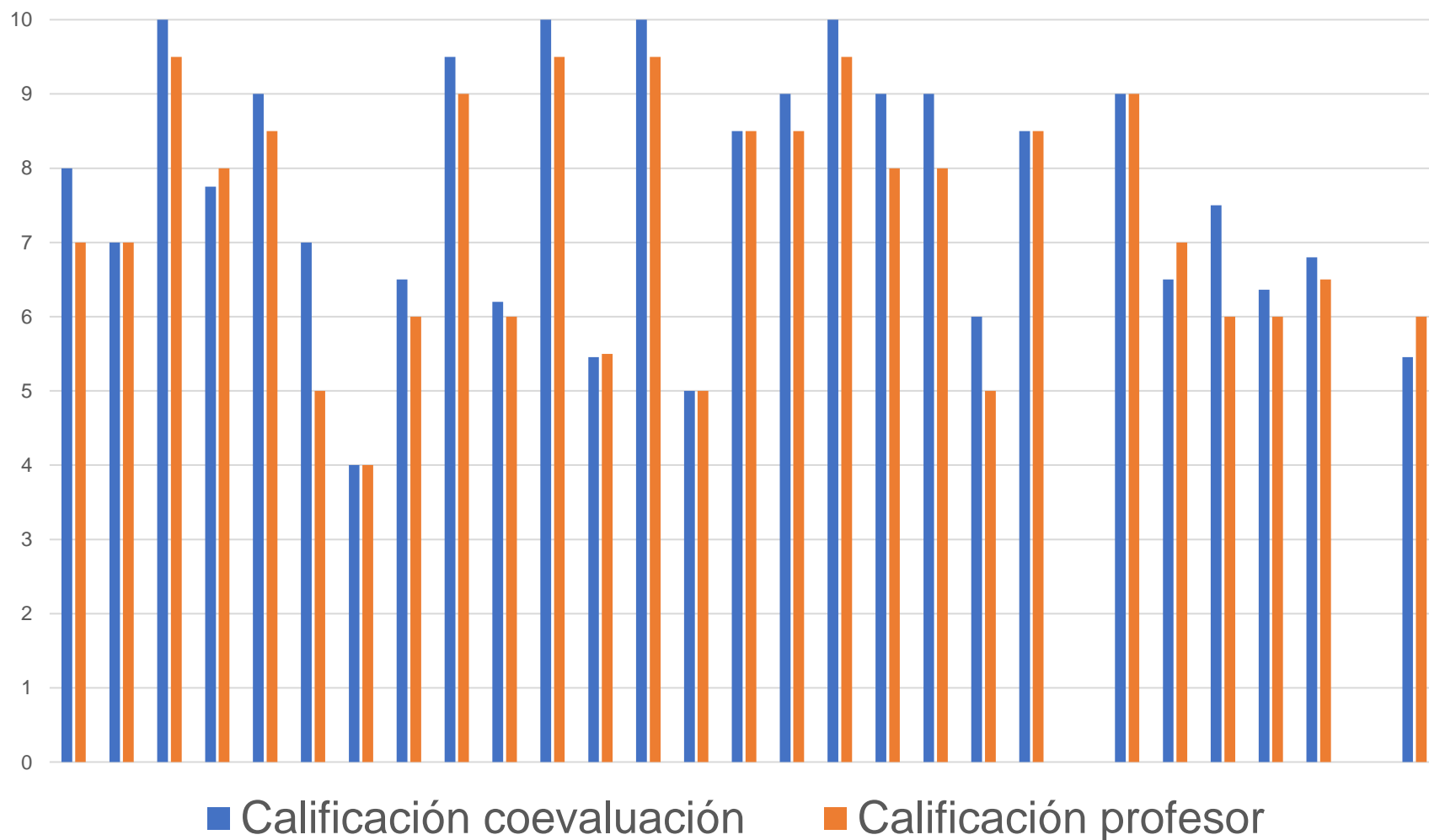
Desarrolla un guion que cumpla con los siguientes requisitos:

1. Debe crear una base de datos del contexto que prefieras. Asegúrate de borrarla previamente en caso de existir por si te da por “reejecutar” el guion.
2. La base de datos debe tener al menos dos tablas relacionadas entre sí (PK, FK) en las que utilices restricciones de borrado y actualización (ON DELETE, ON UPDATE...)
3. Las tablas de la base de datos deben tener varias restricciones semánticas (check, default, unique, enumeraciones,...)
4. Los campos de las tablas debe tener tipología diferente (números, cadenas de texto, fechas, booleanos...)
5. Modifica una de las tablas para crear un nuevo índice.
6. Inserta al menos 2 registros en cada tabla.
7. Crea un usuario que se pueda conectar desde cualquier ubicación y asígnale una contraseña segura.
8. Asígnale permisos a dicho usuario para que pueda leer, modificar, añadir y borrar registros en todas las tablas de la base de datos.
9. Asígnale también permisos a dicho usuario para que pueda crear rutinas y ejecutarlas en dicha base de datos.
10. Añade comentarios aclaratorios antes de cada sentencia previa.
11. Guárdalo como nomApe1.sql siendo nom, tu nombre y ape1, tu primer apellido y **dáselo a tu compañero@ para que lo evalúe.**



Guiones: Actividad coevaluuable - resultados

UD12A01



Desviación calificaciones coevaluación – profesor

Encapsulamiento de código

- Los SGBDs relacionales más utilizados, tales como ORACLE, SQL Server, PostgreSQL o MySQL incorporan una extensión del lenguaje SQL para añadir la opción de realizar **programación procedimental**.

ORACLE ➡ PL-SQL: Procedural Language – Structured Query Language

SQL SERVER ➡ T-SQL: Transact-SQL

PostgreSQL ➡ PgSQL: Postgress SQL

MYSQL ➡ Conjunto de instrucciones propias 🤗



- Con la **programación procedimental** podemos:
 - Encapsular bloques de código
 - Utilizar variables
 - Utilizar estructuras condicionales
 - Utilizar estructuras repetitivas (bucles)
 - Controlar errores y excepciones
- En general, con la programación procedimental disponible en las bases de datos se puede implementar:
 - Bloques anónimos
 - Procedimientos almacenados**
 - Funciones de usuario**
 - Triggers**
 - Eventos**

Recursos: variables

En MySQL, los tipos de variables más habituales son:

- **Variables definidas por el usuario** (con prefijo @): Puedes acceder a ellas sin necesidad de declararlas o inicializarlas previamente. Las no inicializadas tendrán valor NULL y tipo string. Son específicas de cada sesión.
- **Variables locales** (Sin prefijo): Las variables locales necesitan ser declaradas con la cláusula **DECLARE** antes de poder acceder a ellas. La inicialización se hace con la cláusula **DEFAULT**, sino tendrá valor NULL. El ámbito de dichas variables será dentro del bloque **BEGIN ... END** donde se declare (Procedimiento, Función o Trigger). Por ejemplo:

```
DELIMITER $$
CREATE PROCEDURE procedimiento_ejemplo()
BEGIN
    DECLARE start INT unsigned DEFAULT 1;
    DECLARE finish INT unsigned DEFAULT 10;

    SELECT * FROM places WHERE place BETWEEN start AND finish;
END $$
DELIMITER ;
```

- **Variables del Sistema** (con prefijo @@): Pueden ser de tipo GLOBAL, SESSION o ambas. El ámbito de las variables globales es el SGBD mientras que el ámbito de session afecta únicamente a la sesión individual de la conexión del cliente. Ej: @@autocommit, @@transaction_isolation, @@error_count, @@version ...

Recursos: variables

Dentro de nuestros procedimientos, funciones y triggers podemos hacer uso de **variables**.

- **Declaración (solo variables locales):** `declare nombre_variable <tipo>;`
- **Asignación:** `SET @nombre_variable = valor;` ó `SET @nombre_variable := valor;`
`SET nombre_variable = valor` (previa declaración de la variable local)
- **Visualización de valores:** `SELECT @nombre_variable;`
`SELECT nombre_variable;` (previa declaración de la variable local)
- **Visualización de valores con Alias:** `SELECT @nombre_variable AS alias;` (previa inicialización de la variable)

Las variables solo almacenan un valor y se usan generalmente:

- Para recuperar un valor a partir de una sentencia `SELECT` contra nuestras tablas de BD:

```
SET @nombre_variable = (SELECT columna FROM tabla WHERE <condición>);
```

```
SELECT columna INTO @nombre_variable FROM tabla WHERE <condición>;
```

- Para evaluar una condición de una sentencia SQL sobre tablas de BD:

```
SET @nombre_variable = valor;  
SELECT * FROM tabla WHERE campo = @nombre_variable;  
INSERT INTO tabla VALUES(@nombre_variable);
```



Recursos: delimitadores

- A la hora de encapsular código en MySQL bajo un procedimiento almacenado, función, trigger o evento es necesario modificar temporalmente el **carácter separador** que se utiliza para delimitar las sentencias SQL (;).
- El motivo es para que el SGBD no ejecute las sentencias contenidas en el encapsulamiento de código durante su creación.

`delimiter <char(s) elegido(s)>`

- Muchos autores utilizan como delimitadores: `$$` o `//`
- Debes restaurar el delimitador del sistema (;) una vez completes la sentencia de creación del encapsulamiento de código correspondiente.

```
delimiter $$
create procedure nombre_procedimiento()
begin
    INSTRUCCIONES;
end $$
delimiter ;
```

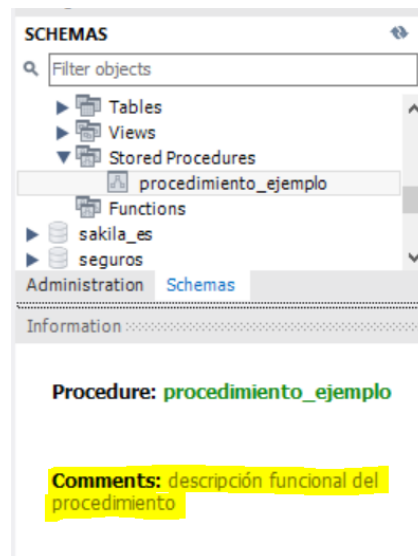
```
delimiter $$
create function nombre_funcion() returns tipoVbleX
begin
    INSTRUCCIONES;
    return vbleX;
end $$
delimiter ;
```

```
delimiter $$
CREATE TRIGGER nombre_trigger {BEFORE | AFTER}
{INSERT | UPDATE | DELETE} ON nombre_tabla
FOR EACH ROW
BEGIN
    INSTRUCCIONES;
END$$
delimiter ;
```

```
delimiter $$
CREATE EVENT nombre_evento ON SCHEDULE AT X
DO
BEGIN
    INSTRUCCIONES;
END$$
delimiter ;
```

Recursos: comentarios

- Los comentarios nos sirven para describir el código, y así que sea más fácil de entender.
- También usamos comentarios para ignorar una cierta parte del código mientras analizamos las consultas SQL (no permita que se ejecute esa parte del código).
 - Desde un carácter ' # ' hasta el fin de la línea.
 - Desde una secuencia ' -- ' hasta el final de la línea.
 - Desde una secuencia ' /* ' hasta la próxima secuencia ' */ '.
- **Como novedad, MySQL dispone de una extensión que permite añadir un comentario que describa la funcionalidad de una rutina almacenada.**



```
DELIMITER $$  
CREATE PROCEDURE procedimiento_ejemplo()  
COMMENT 'descripción funcional del procedimiento'  
BEGIN  
    INSTRUCCIONES;  
END $$  
DELIMITER ;
```



Procedimientos almacenados

- Los **procedimientos almacenados** son objetos que contienen un conjunto de sentencias SQL, que se crean en una BD mediante la sentencia **CREATE PROCEDURE** y que se invocan con la sentencia **CALL**.
- Dichos procedimientos pueden tener **parámetros de entrada** (su valor no puede cambiar dentro del procedimiento), **parámetros de salida** (su valor cambia dentro del procedimiento), **parámetros de entrada y salida** (combinación de los anteriores) o incluso no tener parámetros.
- La sintaxis básica para crear un procedimiento es:

```
delimiter $$
create procedure nombre_proc( IN a tipo1, OUT b tipo2,
INOUT c tipo3 ...)
COMMENT 'descripción funcional del procedimiento'
begin
    INSTRUCCIONES;
end $$
delimiter ;
```
- Sintaxis de invocación de un procedimiento:

```
CALL nombre_proc(<listado de parámetros>);
```
- Sintaxis para el borrado de un procedimiento:

```
DROP <IF EXISTS> PROCEDURE nom_proc;
```

```
CREATE
[DEFINER = user]
PROCEDURE [IF NOT EXISTS] sp_name ([proc_parameter[,...]])
[characteristic ...] routine_body
```

```
CREATE
[DEFINER = user]
FUNCTION [IF NOT EXISTS] sp_name ([func_parameter[,...]])
RETURNS type
[characteristic ...] routine_body
```

```
proc_parameter:
[ IN | OUT | INOUT ] param_name type
```

```
func_parameter:
param_name type
```

```
type:
Any valid MySQL data type

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

routine_body:
Valid SQL routine statement
```

<https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html>

Nota: Para cambiar el código de un procedimiento es necesario borrarlo y volver a crear.

Procedimientos almacenados

Los procedimientos almacenados aportan los siguientes **beneficios**:

1. Se **reduce el tráfico de la red**: un procedimiento almacenado puede encapsular numerosas sentencias SQL. Invocar el procedimiento requiere el envío mínimo de información frente a la sentencias SQL.
2. **Simplifica el mantenimiento** del código: Los procedimientos son reusables entre aplicaciones. Cualquier cambio necesario, solo se lleva a cabo una única vez.
3. **Aportan seguridad**: Se pueden dar permisos de ejecución sobre procedimientos que modifiquen la información de tablas sin necesidad de dar permisos sobre esas tablas.

Para poder ejecutar un procedimiento, los usuarios deben disponer de un privilegio especial llamado **execute**.

Para poder crear un procedimiento, los usuarios deben disponer de un privilegio especial llamado **create routine**.

```
CREATE
[DEFINER = user]
PROCEDURE [IF NOT EXISTS] sp_name ([proc_parameter[,...]])
[characteristic ...] routine_body
```

```
CREATE
[DEFINER = user]
FUNCTION [IF NOT EXISTS] sp_name ([func_parameter[,...]])
RETURNS type
[characteristic ...] routine_body
```

```
proc_parameter:
[ IN | OUT | INOUT ] param_name type
```

```
func_parameter:
param_name type
```

```
type:
Any valid MySQL data type

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

routine_body:
Valid SQL routine statement
```

<https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html>



Recursos: estructuras de control

Instrucciones condicionales

IF-THEN-ELSE

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

CASE

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

ó

CASE

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

Instrucciones repetitivas o bucles

LOOP

```
[begin_label:] LOOP
  statement_list
  ...
  LEAVE | ITERATE label
  ...
END LOOP [end_label] ;
```

Nota: *Iterate* fuerza iteración sin ejecutar resto código del bucle. *Leave* te saca del bucle.

REPEAT

```
[begin_label:] REPEAT
  statement_list
UNTIL search_condition
END REPEAT [end_label] ;
```

WHILE

```
[begin_label:] WHILE search_condition DO
  statement_list
END WHILE [end_label] ;
```

Funciones

- Las **funciones** son objetos que contienen un conjunto de sentencias SQL, que se crean mediante la sentencia **CREATE FUNCTION** y que se invocan con la sentencia **SELECT** o dentro de una expresión.
- Dichas funciones SOLO pueden tener parámetros de entrada (su valor no puede cambiar dentro del procedimiento) por lo que no se indica 'IN' y SIEMPRE devuelven un valor, asociado al nombre de la función.
- Para definir una función es necesario, al igual que en los procedimientos almacenados, modificar temporalmente el carácter separador que se utiliza para delimitar las sentencias SQL (;): **delimiter <char(s) elegido(s)>**
- Además, se debe indicar en la cabecera, el tipo devuelto con la palabra reservada **RETURNS** y en el cuerpo de la función, se debe indicar el valor retornado con la palabra reservada **RETURN**.
- La sintaxis básica para crear una función es:

```
delimiter $$  
CREATE FUNCTION nombre_func (a tipo 1, b tipo2, ...) RETURNS tipoX  
...  
BEGIN  
  INSTRUCCIONES;  
  RETURN vble;  
END$$  
delimiter ;
```

```
CREATE  
  [DEFINER = user]  
  PROCEDURE [IF NOT EXISTS] sp_name ([proc_parameter[,...]])  
  [characteristic ...] routine_body
```

```
CREATE  
  [DEFINER = user]  
  FUNCTION [IF NOT EXISTS] sp_name ([func_parameter[,...]])  
  RETURNS type  
  [characteristic ...] routine_body
```

```
proc_parameter:  
  [ IN | OUT | INOUT ] param_name type
```

```
func_parameter:  
  param_name type
```

```
type:  
  Any valid MySQL data type  
  
characteristic: {  
  COMMENT 'string'  
  | LANGUAGE SQL  
  | [NOT] DETERMINISTIC  
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }  
  | SQL SECURITY { DEFINER | INVOKER }  
}  
  
routine_body:  
  Valid SQL routine statement
```

<https://dev.mysql.com/doc/refman/8.0/en/create-function.html>

- Ejemplos de invocación de una función:

```
Select nombre_func(<valores de parámetros>);
```

```
SET @num= nombre_func(<valores de parámetros>);
```


Funciones

En la definición de las **funciones**, además de indicar el tipo devuelto en la cabecera de las mismas, se debe indicar las características de la función. Las opciones disponibles son las siguientes:

- **DETERMINISTIC**: Indica que la función siempre devuelve el mismo resultado cuando se utilizan los mismos parámetros de entrada.
- **NOT DETERMINISTIC** (por defecto): Indica que la función no siempre devuelve el mismo resultado, aunque se utilicen los mismos parámetros de entrada. Esta es la opción que se selecciona por defecto cuando no se indica una característica de forma explícita.

- **CONTAINS SQL** (por defecto): Indica que la función contiene sentencias SQL (variables, funciones...), pero en ningún caso serán de escritura o lectura de datos.
- **NO SQL**: Indica que la función NO contiene sentencias SQL.

- **READS SQL DATA**: Indica que la función NO modifica los datos de la base de datos y que contiene sentencias de lectura de datos, como la sentencia SELECT.
- **MODIFIES SQL DATA**: Indica que la función SI modifica los datos de la base de datos y que contiene sentencias como INSERT, UPDATE o DELETE.

Las **funciones** pueden ser borradas usando la sentencia:

`DROP <IF EXISTS> FUNCTION < nombre_func >`

Para poder crear una función “segura” en MySQL es necesario indicar al menos una de estas tres características:

- DETERMINISTIC
- NO SQL
- READS SQL DATA


Error Code 1418

```
SET GLOBAL  
log_bin_trust_function_creators = 1;
```

La variable **log_bin_trust_function_creators** controla si se debe confiar o no en la creación de funciones “inseguras”.

Dicha variable volverá a su valor original al cerrar sesión salvo que se actualice la configuración (my.ini) bajo la etiqueta [mysqld]:

```
log_bin_trust_function_creators = 1
```

Y se reinicie el servidor de MySQL  Windows

Las funciones en MySQL **NO soportan la recursividad**. Tampoco los Triggers. Los procedimientos almacenados sí, pero por defecto está deshabilitado.

```
set @@max_sp_recursion_depth = x;  
siendo x > 0
```

Bloques anónimos

