# U.D.4. AUTOMATIZACIÓN DE TAREAS

## 1.TRIGGERS (disparadores)

Los disparadores (triggers) son objetos relacionados a tablas y que son activados cuando sucede un determinado evento en las tablas asociadas. Estos eventos son sentencias que modifican los datos de las tablas (INSERT, DELETE, UPDATE), y las instrucciones del disparador pueden ser ejecutadas antes (BEFORE) y/o después (AFTER) de que la fila es modificada. Se pueden utilizar triggers desde la versión MySQL 5.0.2.

La sintaxis de la sentencia que permite crear un disparador es la siguiente:

**CREATE TRIGGER** *nombre_trigger*
*momento_ejecucion*
*evento_disparador*
ON *nombre_tabla* FOR EACH ROW *trigger_sentencia*

- El disparador está asociado a la tabla ***nombre_tabla***, que no puede ser una tabla temporal, ni una vista.

- ***momento_ejecución*** es indica si el disparador se activa antes (BEFORE) o después (AFTER) de ejecutar el evento que lo activa.

- ***evento_disparador*** indica el tipo de sentencia que activa el disparador. Puede ser alguna de las siguientes: INSERT, UPDATE o DELETE. Por ejemplo, poniendo BEFORE para una sentencia INSERT puede ser utilizada para verificar los valores antes de la inserción de una fila en una tabla.

- ***trigger_sentencia*** es la sentencia, o grupo de sentencias que se ejecuta cuando se activa el disparador. Los disparadores son muy parecidos a los procedimientos almacenados, de tal forma que si deseamos ejecutar múltiples acciones, en un mismo disparador, podemos encapsular estas acciones dentro de una construcción BEGIN, END.

Los disparadores tienen un par de palabras clave extra - ***OLD*** y ***NEW*** - las cuales se refieren respectivamente a los valores de las columnas antes y después de que la sentencia fuese procesada. Las sentencias INSERT únicamente permiten NEW, las sentencias UPDATE permiten ambos, NEW y OLD, y las sentencias DELETE permiten sólo OLD. La razón para esto debe ser obvia.

**Veamos un ejemplo en el que los disparadores pueden ser de utilidad:**

Queremos llevar una auditoria de los cambios efectuados en los datos de una tabla. Así podremos saber quién cambió o eliminó una fila, sin necesidad de recurrir a los ficheros de registro de MySQL.

Para hacer esto, podemos asignar un disparador a una tabla que se active después (AFTER) de una sentencia DELETE o UPDATE, y que guarde los valores del registro, así como alguna otra información de utilidad en una tabla de auditoria (log).

**Caso práctico:** tenemos la tabla de clientes con la estructura que se ve a continuación. Queremos guardar en una tabla de registro (auditoria_clientes) todos los cambios que se hagan en el límite de crédito de los clientes :

```
CREATE TABLE clientes
(
id                    INT    NOT NULL      AUTO_INCREMENT,
empresa               VARCHAR (25),
limite_credito        DECIMAL(8,2),
PRIMARY KEY(id)
) ENGINE = InnoDB;
```

/* Creación de la tabla de auditoría */

```
CREATE TABLE auditoria_clientes
(
  id                  INT NOT NULL AUTO_INCREMENT,
  empresa             VARCHAR(25),
  anterior_limite     DECIMAL(8,2),
```

```
    nuevo_limite        DECIMAL(8,2),
    usuario             VARCHAR (40),
    cambiado            DATETIME,
    accion              CHAR(1),
    PRIMARY KEY(id)
) ENGINE = InnoDB;
```

Y ahora creamos un disparador que vaya insertando filas en la tabla de auditoria cada vez que alguien ejecute una modificación sobre la tabla. Para crear triggers utilizamos la sentencia CREATE TRIGGER:

/* Creación del disparador. Se utiliza DELIMITER, para cambiar el caracter de final de sentencia */

```
DELIMITER //
CREATE TRIGGER trigger_auditoria_clientes AFTER UPDATE ON clientes
 FOR EACH ROW
  BEGIN
   INSERT INTO auditoria_clientes(empresa, anterior_limite, nuevo_limite, usuario, cambiado,accion)
   VALUES (OLD.empresa, OLD.limite_credito, NEW.limite_credito, CURRENT_USER(), NOW(),
'M');
  END
//
DELIMITER ;
```

De esta forma podremos saber quién realizó una actualización en la tabla de *clientes*, y cuando la hizo.

Para poder registrar las filas borradas, podemos crear un disparador, igual al anterior, que se dispararía con la acción AFTER DELETE. En este caso, cuando se borre una fila de la tabla de *clientes*, se inserta una fila en la tabla *auditoría_cliente*, y en la columna *accion* se le asigna la constante "B", que indica que la operación registrada corresponde a un borrado. Así podemos tener registradas en la misma tabla de auditoria tanto las modificaciones como los borrados y seleccionarlos de forma sencilla.
**Nota**: Este ejemplo fue probado en la versión 5.0.10.

Una cosa importante que hay que mencionar es que dentro de los disparadores se pueden llamar procedimientos almacenados (procedures), desde la versión 5.0.10 en adelante.

## 2.PROCEDIMIENTOS ALMACENADOS Y FUNCIONES

Un procedimiento almacenado es un conjunto de sentencias que pueden ser almacenadas en el servidor bajo un nombre único, y que las aplicaciones cliente pueden ejecutar indicando el nombre del procedimiento y pasándole, opcionalmente, los parámetros necesarios.

El conjunto de instrucciones almacenado en el servidor ya se encuentra enlazado y optimizado para su ejecución, lo que supone una mejora en el rendimiento y una disminución del tráfico en la red.

Algunas de las utilidades de los procedimientos almacenados:

- Cuando muchas aplicaciones, escritas en distintos lenguajes y plataformas, necesitan realizar un conjunto de operaciones susceptibles de almacenar en el servidor.

- Cuando la seguridad es una prioridad. Los bancos, por ejemplo, utilizan los procedimientos almacenados para todas las operaciones estándar, y no lo dejan en manos de los programadores de las aplicaciones. De esta forma se aseguran de que las operaciones son hechas y registradas correctamente. En una situación de este tipo, las aplicaciones cliente no acceden directamente a las tablas, sino que lo hacen a través de los procedimientos almacenados, que son los encargados de manejar los datos en la base de datos.

MySQL sigue la sintaxis de la norma SQL:2003 para procedimientos almacenados. Se empiezan a implementar en la versión MySQL 5.0, y todavía están en pleno desarrollo.

Los procedimientos almacenados se crean utilizando las sentencias CREATE PROCEDURE, según la siguiente sintaxis:

**CREATE PROCEDURE** *nombre_procedimiento* ( [*parámetro* [,...]])
[*característica* ...] *cuerpo_procedimiento*

***parametro***: [ IN | OUT | INOUT ] *nombre_parámetro tipo*

***tipo***: *Cualquier tipo de dato válido en MySQL*

***característica***: LANGUAGE SQL | SQL SECURITY { DEFINER | INVOKER } | COMMENT '*string*'

***cuerpo_procedimiento****: Secuencia de instrucciones SQL*

- Los paréntesis que contiene la lista de parámetros son obligatorios, y si no se necesita ninguno no se escribirá nada en ellos. El procedimiento almacenado puede devolver resultados mediante los parámetros OUT e INOUT. Si no se indica el tipo del parámetro, se considera que es de entrada (IN).

- SQL_SECURITY permite indicar si en la ejecución del procedimiento almacenado se utilizan privilegios del creador del procedimiento (DEFINER - valor por defecto) o del usuario del procedimiento (INVOKER).

- COMMENT permite introducir un comentario que se verá al ejecutar la sentencia SHOW CREATE PROCEDURE.

Los procedimientos almacenados pueden ser modificados con las sentencia ALTER PROCEDURE, y borrados con DROP PROCEDURE.

Para ejecutar un procedimiento almacenado ya creado en el servidor se utiliza la sentencia CALL, con la siguiente sintaxis:

**CALL** n*ombre_procedimiento*( [ *parámetro* [, ...] ] )

Veamos un ejemplo muy básico de creación y uso de un procedimiento almacenado:

```
DELIMITER //
CREATE   PROCEDURE sp_contar_pedidos ( pcli_codigo INT, OUT num_pedidos INT)
  BEGIN
    SELECT COUNT(*)  INTO num_pedidos
       FROM pedidos
       WHERE cli_codigo = pcli_codigo;
  END
//
DELIMITER ;
```

De esta forma se crea el procedimiento almacenado. Si queremos calcular los pedidos que tiene el cliente cuyo código es 1, ejecutaríamos la siguiente instrucción:

```
CALL sp_contar_pedidos(1, @var ) ;
SELECT @var;
+----+
|@var|
+----+
| 21 |
+----+
```

Veamos la sintaxis de algunas de las múltiples sentencias que se pueden utilizar en un procedimiento almacenado:

| | |
|---|---|
| **DECLARE** | **a) Variables**<br>Pemite definir variables. Las definiciones deben incluirse al comienzo, después de la sentencia BEGIN. Son variables locales dentro del procedimiento almacenado.<br><br>`DECLARE nombre_variable [ ,…] tipo [ DEFAULT valor ]`<br><br>**b) Declaración de condiciones**<br>Especifica condiciones de error sobre las que se precisa un tratamiento especial. El nombre de la condición se puede utilizar en una sentencia DECLARE HANDLER.<br><br>`DECLARE nombre_condición CONDITION FOR valor_condición`<br><br>Valor condición = SQLSTATE[ VALUE ] *valor_sqlstate* \| *código_error_mysql*<br><br>**c) Declaración de manipuladores**<br>Indica las acciones que se han de ejecutar en el caso que se produzca una condición de error<br><br>`DECLARE tipo_manipulador HANDLE FOR valor_condición [ ,… ] sentencia`<br><br>Tipo_manipulador = CONTINUE \| EXIT<br>Valor_condición = SQLSTATE[ VALUE ] *valor_sqlstate* \| *nombre_condición* \| *código_error_mysql* |
| **SET** | Permite asignar valores a las variables locales del procedimiento<br><br>SET nombre_variable = expresión [ , nombre_variable = expresión] … |
| **SELECT ... INTO** | Permite almacenar el resultado de una fila en una variable<br><br>SELECT nombre_columna [ ,...] INTO nombre_variable [ ,... ] |
| **Cursores**<br><br>**DECLARE**<br><br>**OPEN**<br><br>**FETCH**<br><br>**CLOSE** | Cuando una sentencia SELECT devuelve mas de una fila, podemos utilizar los cursores para movernos por las filas resultantes.<br><br>Permite declarar un cursor<br><br>DECLARE nombre_cursor CURSOR FOR sentencia_SELECT<br><br>Cuando se quiere utilizar un cursor es necesario abrirlo, con esta instrucción<br><br>OPEN nombre_cursor<br><br>Obtiene una fila del conjunto de resultados y avanza el puntero a la fila siguiente. Cuando ya no quedan más filas se producen la condición SQLSTATE “02000”.<br><br>FETCH nombre_cursor INTO nombre_variable [, nombre_variable ] …<br><br>Permite cerrar un cursor<br><br>CLOSE nombre_cursor |
| **Estructuras de control de flujo**<br><br>**IF** | Dentro de un procedimiento almacenado se pueden utilizar sentencias de control de flujo para poder escribir procedimientos que tengan cierta complejidad<br><br>Instrucción condicional<br><br>IF condición THEN lista_de_sentencias<br>[ ELSEIF condición THEN lista_de_sentencias ] …<br>[ ELSE lista_de _sentencias ]<br>END IF |

| | |
|---|---|
| **CASE** | Permite ejecutar una o mas sentencias dependiendo del valor que toma una expresión<br><br>CASE expresión<br>[ WHEN valor THEN sentencia ] …<br>[ ELSE sentencia ]<br>END CASE |
| **WHILE** | Repite un bloque de sentencias, mientras se cumpla una condición<br><br>[ etiqueta_inicio: ] WHILE condición DO<br>lista_de_sentencias<br>END WHILE [ etiqueta_fin ] |
| **REPEAT** | Repite un bloque de sentencias, hasta que se cumple una condición<br><br>[ etiqueta_inicio: ] REPEAT<br>lista_de_sentencias<br>UNTIL condición<br>END REPEAT[ etiqueta_fin ] |

Ejemplo de creación y uso de un procedimiento almacenado para llevar un registro de los intentos fallidos de inicio de sesión:

```
DROP PROCEDURE sp_error_login1;
delimiter //

CREATE PROCEDURE sp_error_login (pUsuario CHAR(16), pPalabraPaso CHAR(16), OUT mensaje
VARCHAR(50))
BEGIN
  DECLARE errorAnterior DATETIME;
  DECLARE errorActual DATETIME;
  DECLARE intentos INT;
  DECLARE difSegundos INT;


  SET errorActual = NOW();


  SELECT count(*) INTO intentos FROM LogError WHERE UsuarioID = pUsuario;

  IF intentos < 3 then
       SET mensaje = "Revise los datos de usuario y contraseña";
  else
       SET mensaje = "Se bloqueará la cuenta de usuario";
       UPDATE Clientes SET PalabraPaso = "asdfg1(23-78SD" WHERE ClienteID = pUsuario;
  END IF;

  IF intentos > 0 then

    SELECT Fecha INTO errorAnterior FROM LogError WHERE UsuarioID = pUsuario ORDER BY
Fecha DESC Limit 1;

    SET difSegundos= TIME_TO_SEC(TIMEDIFF(errorActual,errorAnterior));

    IF difSegundos  > 3 THEN

          INSERT INTO LogError (UsuarioID, PalabraPaso, Fecha) VALUES (pUsuario,
pPalabraPaso, errorActual);

    END IF;
```

```
  ELSE

    INSERT INTO LogError (UsuarioID, PalabraPaso, Fecha) VALUES (pUsuario, pPalabraPaso ,
errorActual);

 END IF;


END
//
delimiter ;
call sp_error_login(1,"111",@met);
SELECT @met;
SELECT * FROM LogError WHERE UsuarioID = 1 ORDER BY Fecha DESC;
```

Ejemplo de creación de un procedimiento almacenado para contar las compras realizadas por nuestros clientes, y en función de los resultados modificar un campo de la tabla de clientes que debería almacenar el número de compras que ha realizado (clt_compras):

```
DROP PROCEDURE ventas_todos;
DELIMITER //

CREATE PROCEDURE ventas_todos()

 BEGIN

   DECLARE cliente, contar INTEGER;
   DECLARE final INTEGER DEFAULT 0;

   /* declaracion de un cursor para recorrer los resultados de la consulta que devuelve los números
de todos los clientes que hicieron compras, y el número de compras que hizo cada uno */
   DECLARE c1 CURSOR FOR
      SELECT ven_clt, COUNT(*) FROM ventas GROUP BY ven_clt;

   /* declaración de un manipulador para la condición de error de final de lectura de resultados
*/
   DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET final=1;

   OPEN c1;

   REPEAT
     /* avanzamos una línea de resultados y pasamos las columnas de la selección a las variables */
     FETCH c1 INTO cliente, contar;
     IF NOT final THEN
       /* si no se llegó al final de los resultados modificamos los datos del cliente */
       UPDATE clientes SET clt_compras = contar WHERE clt_num = cliente;
     END IF;
   UNTIL final END repeat;
   CLOSE c1;
 END

 //

 DELIMITER ;
```

## 3.CONTROL DE ACCESO CONCURRENTE

Los mecanismos de control de concurrencia en bases de datos están basados en las técnicas de bloqueo. MySQL ofrece dos niveles de bloqueo: a nivel de fila o a nivel de tabla, dependiendo que se trabaje con tablas tipo InnoDB o tablas MyISAM.

En las tablas tipo InnoDB el bloqueo de una fila individual lo hace automáticamente el sistema, al ejecutar una orden que modifique el contenido de la tabla, por ejemplo: UPDATE. Si queremos bloquear un grupo de filas en ese caso hay que usar una transacción.

Cuando se está ejecutando una transacción se produce un bloqueo de las líneas afectadas por las operaciones UPDATE, INSERT y DELETE contenidas en la transacción, hasta que se llega al final de la misma. El resto de usuarios sólo pueden acceder a esas líneas en modo de lectura mientras se está ejecutando la transacción. Teniendo en cuenta que el número de bloqueos sobre líneas tiene una limitación, es recomendable limitar el número de estos comandos en una transacción.

Algunas veces, por ejemplo, cuando se está haciendo un trabajo de recuperación de una tabla, necesitamos bloquear la tabla para que el resto de usuarios no pueda acceder a ella. El bloqueo a nivel de tabla se hace ejecutando una orden SQL introducida por el usuario. La sintaxis de la orden es:

**LOCK TABLE** nombre-tabla { READ | WRITE }

- La palabra READ permite al resto de usuarios acceder en modo lectura, pero no permite escribir
- La opción WRITE impide cualquier acceso (leer y escribir) al resto de usuarios

Para liberar las tabla previamente bloqueadas se utiliza el comando:

**UNLOCK TABLES**

## 4.SALVAGUARDIA (COPIA DE SEGURIDAD) DE UNA BD

Por razones de seguridad, está recomendado efectuar una salvaguardia periódica de la base de datos. Esta copia de seguridad (backup) consiste en efectuar una copia física de los ficheros de la BD sobre soportes auxiliares (CD, unidades ZIP, cintas magnéticas, etc.).

Mysql dispone de una utilidad dentro del directorio bin, llamada *mysqldump*, que permite crear un script con el esquema y los datos de la base de datos. Este script se puede utilizar como copia de seguridad, para mover bases de datos de un servidor a otro, o para crear una base de datos de pruebas basada en una ya existente, etc. . La sintaxis es la siguiente.

***mysqldump*** *[opciones] nombre_base_de datos [nombre_tabla]*

Algunas de las opciones que se pueden utilizar:

| | |
|---|---|
| `--help` | Muestra un mensaje de ayuda, con todas las opciones (-?) |
| `--add-locks` | Añade LOCK TABLES, y UNLOCK TABLE para copiar cada tabla. |
| `--databases` | Permite indicar el nombre de varias bases de datos (-B) |
| `--all-databases` | Vuelca el contenido de todas las bases de datos del servidor (-A) |
| `--flush-logs` | Vacía los ficheros de registro de log antes de comenzar el volcado (-F) |
| `--lock-tables` | Bloquea todas las tablas antes de comenzar el volcado (-l) |
| `--no-create-db` | No incluye la sentencia CREATE DATABASE (-n) |
| `--no-create-info` | Copia solamente los datos. No incluye las sentencia de creación (base de datos ni tablas) |
| `--no-data` | No incluirá ninguna información sobre los registros de la tabla. Esta opción sirve para crear una copia de sólo la estructura de la base de datos |
| `--opt` | Hace una operación de volcado rápida. Es lo mismo que especificar: `--add-drop-table --add-locks --–create-options --disable-keys --extended-insert --lock-tables --quick --set-charset.` |
| `--quick` | Acelera el proceso en tablas grandes, ya que no carga en memoria los resultados, sino que recupera y graba fila a fila (-q) |
| `--result-file=` | Guarda la salida en el fichero indicado (-r *fichero*) |

Para tablas transaccionales (InnoDB, BDB):

`--single-transaction` Vuelca las tablas transaccionales en un estado consistente, sin bloquear ninguna aplicación. Es incompatible con `–lock-tables.` Para tablas grandes es recomendable combinar con `--quick`

**Nota**: Estas opciones pueden grabarse en el fichero de configuración, dentro de la sección [mysqldump]

Ejemplo de comandos mysqldump:

shell> **mysqldump --add-drop-table --add-locks -u usuario -p --databases bd1 bd2 > copia-01-2006.sql**

La frecuencia de esta operación depende estrechamente de la frecuencia y la importancia de las modificaciones efectuadas sobre la BD. Para una BD utilizada exclusivamente en modo consulta una sola salvaguardia es suficiente, mientras que para otra donde, mas del 30% cambia en una jornada, necesita una salvaguardia diaria. Además de la frecuencia de salvaguardia, hace falta elegir el momento para efectuar esta operación ya que requiere bloquear el acceso a los datos. Se debe elegir el momento de mínima actividad de la BD, por ejemplo, la noche para las salvaguardias diarias, o el fin de semana para las salvaguardias semanales.

La sentencia *mysqldump* es mas lenta que la copia directa, pero es mucho mas segura, porque opera en cooperación con el servidor MySQL, evitando problemas en el caso de no tener bloqueado el acceso de los usuarios a la base de datos, y se produzcan operaciones en ella mientras se está haciendo la copia de seguridad. Además, produce ficheros de texto que pueden ser utilizados en otras máquinas, permitiendo la recuperación de la base de datos en otros sistemas diferentes al nuestro.

Podemos resumir los pasos que hay que hacer cada vez que se haga la copia de seguridad:

- Bloquear las tablas para que no se pueda modificar la base de datos mientras se está haciendo la copia
- Hacer copia de seguridad de los ficheros log
- Reiniciar los ficheros log (FLUSH LOGS)
- Asegurarse que todas las páginas de índices activas se escriben en el disco (FLUSH TABLES)
- Hacer la copia de la base de datos
- Desbloquear el acceso a los usuarios de la base de datos

Otra técnica para realizar copias de seguridad de la base de datos es utilizar el script ***mysqlhotcopy***. Este script está escrito en Perl, y usa *LOCK TABLES*, *FLUSH TABLES*, y *cp* o *scp* para realizar una copia de seguridad rápida de la base de datos. Es la forma más rápida de hacer una copia de seguridad de la base de datos o de tablas, pero sólo puede ejecutarse en la misma máquina donde está el directorio de base de datos, y sólo realiza copias de seguridad de tablas MyISAM. Funciona en Unix y NetWare. Sintaxis:

shell> **mysqlhotcopy nombre_de_base_de_datos [/ruta/al/nuevo_directorio]**

Para hacer copias de seguridad de una tabla a un nivel SQL, se pueden utilizar las sentencias *SELECT INTO OUTFILE*, o *BACKUP TABLE*. En estos casos, el archivo de salida no debe existir previamente. La sintasis de la orden SELECT ... INTO OUTFILE permite especificar el formato de los campos y las líneas:

```
SELECT columnas INTO OUTFILE 'path/nombre_fichero.txt'
[FIELDS
        [TERMINATED BY '\t']
        [[OPTIONALLY] ENCLOSED BY '']
        [ESCAPED BY '\\' ]
]
[LINES
        [STARTING BY '']
        [TERMINATED BY '\n']
]
FROM nombre_tabla;
```

Esta forma de copia crea un fichero ASCII de texto, que puede ser utilizado para exportar datos a otras aplicaciones que puedan leer ficheros tipo txt o csv (*comma-separated values* ), como por ejemplo una hoja de cálculo.

**Copias de seguridad incrementales**: Teniendo en cuenta que el proceso de volcado de la base de datos completa puede llevar mucho tiempo, puede ser interesante establecer una política de copias de seguridad basada en la utilización de copias incrementales, con la ayuda de los ficheros log binarios.

## 5.RECUPERACION DE UNA BASE DE DATOS

Recordemos que mysqldump crea un fichero de órdenes SQL, por tanto, si queremos restaurar una copia de seguridad hecha con mysqldump, lo haremos igual que se ejecuta cualquier fichero de órdenes SQL (desde la línea de comandos o usando un cliente que que permita editar y ejecutar ficheros de órdenes SQL). Por ejemplo, si queremos recuperar la copia de seguridad copia-01-2006.sql, desde la línea de comandos del DOS, ejecutaríamos la orden:

shell> **mysql -u usuario -p < copia-01-2006.sql**

Si lo que queremos es ejecutar las órdenes que contien un fichero de registro binario, podemos utilizar la utilidad mysqlbinlog, y enviar la salida al cliente mysql. Por ejemplo:

shell> **mysqlbinlog nombre_ordenador-bin.00002 | mysql –u root -p**

Estas dos operaciones, utilizadas conjuntamente, nos permiten recuperar una base de datos que esté dañada, siempre que dispongamos de una copia de seguridad y u fichero de registro binario que guarde los cambios que se han hecho desde el momento en que se hizo la copia de seguridad. A continuación se muestran los pasos a seguir en el caso de querer recuperar una o mas bases de datos del servidor que se han estropeado por un fallo lógico o físico:

- ***Bloquear el acceso al servidor:*** Por ejemplo, arrancando el servidor sin permitir conexiones a través de la red, con la opción: *--skip-networking*
- ***Recuperar la última copia de seguridad***
  Si la copia de seguridad de la base de datos se ha realizado con órdenes de sistema operativo como copy, cp o tar, se realizará la recuperación copiando los ficheros a los directorios adecuados. En este caso es necesario conocer muy bien la estructura de directorios de las bases de datos MySQL, para volver a situar cada fichero en su sitio. Además, el servidor ha de desconectarse antes de iniciar la copia y reiniciarlo al finalizar la tarea de copia.
  Si la copia se hizo com la utilidad mysqldump, ejecutaremos es fichero de ordenes creado:

  shell> **mysql -u usuario -p < copia-01-2006.sql**

- ***Recuperar las operaciones de modificación contenidas en los ficheros de registro binarios****:*

  shell> **mysqlbinlog nombre_ordenador-bin.00002 | mysql –u root -p**

- ***Desbloquear el acceso al servidor:*** Volviendo a reiniciar el servidor, permitiendo conexiones a través de la red.

***Nota:*** Si se ha dañado la base de datos mysql, o no podemos acceder como root porque no conocemos la contraseña, es necesario comenzar arrancando el servidor con la opcion *--skip-grant-tables*, para que no tenga en cuenta las tablas en las que se guardan los privilegios de acceso.
Además, después de restaurar las tablas será necesario indicar al servidor que cargue las tablas de privilegios y que empiece a usarlas mediante la sentencia: shell> **mysqladmin flush-privileges**

## 6.IMPORTAR TABLAS DESDE FICHEROS DE TEXTO DELIMITADOS

Prácticamente todos los sistemas de gestión de bases de datos tienen la capacidad de importar y exportar tablas a ficheros de texto (ASCII) de campos delimitados. Por lo tanto esta utilidad es muy aconsejable para pasar datos de un SGBD a otro además de para hacer copias de seguridad de tablas individuales.

Un ejemplo de fichero de texto de campos delimitados podría ser el siguiente (Fichero personal.txt):

```
1112345;"Martínez Salas,;Fernando";"PROFESOR;2200.00;10
4123005;"Bueno Zarco,;Elisa";"PROFESOR";2200.00;10
4122025;"Montes García, M.Pilar";"PROFESOR";2200.00;10
1112346;"Rivera Silvestre, Ana";"PROFESOR";2050.00;15
9800990;"Ramos Ruiz, Luis";"PROFESOR";2050.00;15
8660990;"De Lucas Fdez, M.Angel";"PROFESOR";2050.00;15
7650000;"Ruiz Lafuente, Manuel";"PROFESOR";2200.00;22
43526789;"Serrano Laguía, María";"PROFESOR";2050.00;45
4480099;"Ruano Cerezo, Manuel";"ADMINISTRATIVO";1800.00;10
1002345;"Albarrán Serrano, Alicia";"ADMINISTRATIVO";1800.00;15
7002660;"Muñoz Rey, Felicia";"ADMINISTRATIVO";1800.00;15
5502678;"Marín Marín, Pedro";"ADMINISTRATIVO";1800.00;22
6600980;"Peinado Gil, Elena";"CONSERJE";1750.00;22
4163222;"Sarro Molina, Carmen";"CONSERJE";1750.00;45
```

Estos ficheros se caracterizan por estar almacenados en formato ASCII habitualmente con las extensiones .txt o .csv. Tienen los registros de longitud variable y los campos están delimitados por un carácter especial (en este caso “;”). También habitualmente los campos de caracteres vienen entrecomillados.

MySQL proporciona dos métodos para importar directamente este tipo de ficheros a tablas. Una de ellas es la utilidad *mysqlimport* , el otro método es lanzar la sentencia LOAD DATA INFILE. En este apartado se comenta únicamente este último método. La utilidad *mysqlimport* se puede ver en detalle en la sección *8.10 del Manual de MySQL.*

Sintaxis genérica de la sentencia LOAD DATA INFILE :

**LOAD DATA** [LOCAL] INFILE *'path/nombre_fichero.txt'* [REPLACE | IGNORE]
INTO TABLE *nombre_tabla*
[FIELDS
[TERMINATED BY '\t']
[[OPTIONALLY] ENCLOSED BY "]
[ESCAPED BY '\\' ]
]
[LINES
[STARTING BY "]
[TERMINATED BY '\n']
]
[IGNORE *número* LINES]
[(*nombre_columna*,...)]

Se observa en esta sintaxis que la mayoría de las cláusulas de esta sentencia son optativas, vamos a comentar las más interesantes.

Si se especifica la cláusula LOCAL significa que el fichero es leído por el por el programa cliente en el equipo donde se esta ejecutando el cliente de MySQL desde el que se esta lanzando la sentencia LOAD DATA. Si la cláusula LOCAL se omite significa que el fichero debe encontrarse en el host del servidor MySQL .

Para poder ejecutar esta sentencia es necesario crear previamente la tabla sobre la que van a ir los datos. La tabla puede o no estar vacía. Habrá que tener cuidado al hacer una carga masiva de datos con esta sentencia el evitar entradas duplicadas en campos marcados como UNIQUE o PRIMARY KEY.

En el siguiente ejemplo se va a crear una tabla y lanzar la sentencia LOAD DATA que permita cargar los datos del fichero de personal (personal.txt) anterior.

```
USE TEST;
CREATE TABLE IF NOT EXISTS personal(
        dni int(10),
        apellidos varchar(30),
        funcion enum('PROFESOR','ADMINISTRATIVO','CONSERJE'),
        salario float(6,2),
        cod_centros int(5),
        CONSTRAINT pk_codigo PRIMARY KEY (dni));
LOAD DATA INFILE 'C:\\temp\\personal.txt'
        INTO TABLE personal
        FIELDS TERMINATED BY ';' ENCLOSED BY ""
        LINES TERMINATED BY '\r\n';
```

Muy importante es construir correctamente las cláusulas *FIELDS* y *LINES* de la sentencia *LOAD DATA*. En la cláusula *FIELDS* indicamos que el carácter delimitador es el ";" y que las cadenas de texto van entrecomilladas. En la cláusula *LINES* indicamos que el carácter de final de línea es *'\r\n'*. (Los ficheros de texto de Windows suelen llevar *\r\n* como carácter de final de línea mientras que los ficheros de texto en Linux es únicamente *\n*).

Para poder lanzar una sentencia *LOAD DATA* es necesario ser administrador de la base de datos o al menos tener concedido el privilegio *FILE*.