CTSQL

Procedimientos almacenados y *triggers*

BASE 100, S.A.
www.base100.com

Índice

1. NOVEDADES Y CONSIDERACIONES PREVIAS ... 3
1.1 ACTUALIZACIÓN DE BASES DE DATOS GENERADAS CON VERSIONES ANTERIORES ... 3
1.2 ASPECTOS A TENER EN CUENTA ... 3
2. INTRODUCCIÓN A LOS PROCEDIMIENTOS ALMACENADOS Y *TRIGGERS* ... 5
2.1 PROCEDIMIENTOS ALMACENADOS ... 5
2.1.1 Ventajas de los procedimientos almacenados ... 5
2.2 TRIGGERS ... 6
2.2.1 Ventajas de los triggers ... 6
2.2.2 Consideraciones sobre triggers ... 6
2.2.3 Tipos de triggers ... 6
3. INSTALACIÓN DEL SERVIDOR DE PROCEDIMIENTOS ALMACENADOS ... 7
3.1 REQUISITOS PREVIOS ... 7
3.2 SERVIDOR DE PROCEDIMIENTOS ALMACENADOS ... 7
3.3 PUESTA EN EJECUCIÓN DEL SERVIDOR DE PROCEDIMIENTOS ALMACENADOS (SP) ... 7
3.3.1 Fichero storedserverlauncher.ini ... 9
3.3.2 Fichero StoredProcServer.properties ... 9
3.3.3 Fichero ctsql.ini ... 10
4. PROCEDIMIENTOS ALMACENADOS EN CTSQL ... 11
4.1 CREACIÓN DE PROCEDIMIENTOS ALMACENADOS ... 11
4.2 EJECUCIÓN DIRECTA DE PROCEDIMIENTOS ALMACENADOS ... 11
4.3 ELIMINACIÓN DE UN PROCEDIMIENTO ALMACENADO ... 11
4.4 ACTUALIZACIÓN DE UN PROCEDIMIENTO ALMACENADO ... 12
4.5 TABLA SYSPROCEDUR ... 12
5. *TRIGGERS* EN CTSQL ... 13
5.1 CREACIÓN DE *TRIGGERS* ... 13
5.2 ELIMINACIÓN DE *TRIGGERS* ... 13
5.3 ACTIVACIÓN / DESACTIVACIÓN DE *TRIGGERS* ... 13
5.4 TABLA SYSTRIGGERS ... 14
6. CREANDO PROCEDIMIENTOS ALMACENADOS ... 15
6.1 CLASE ABSTRACTSQLPROCEDURE ... 15
6.1.1 Métodos de la clase AbstractSqlProcedure ... 15
7. EJEMPLOS ... 17

© Copyright BASE 100, S.A. Todos los derechos reservados. Ninguna parte de este documento puede ser reproducida ni transmitida por medio alguno sin permiso previo por escrito del titular del copyright. Todos los productos citados en este documento son marcas registradas o marcas comerciales registradas de sus respectivos propietarios.

NTUTprocalmatriggers - V.2.1

1. Novedades y consideraciones previas

A partir de la versión 3.4 de MultiBase y de la versión 4.2 de Cosmos, el gestor CTSQL incluye las siguientes novedades:

1. Soporte para tablas de más de 2 Gbytes para sistemas operativos basados en tecnología NT.
2. Uso de procedimientos almacenados.
3. Uso de *triggers.*

1.1 Actualización de bases de datos generadas con versiones anteriores

Como consecuencia de los puntos 2 y 3 anteriormente indicados, ha sido necesario crear dos nuevas tablas en el catálogo de la base de datos: SYSPROCEDUR y SYSTRIGGERS. Esto implica que para poder utilizar estos mecanismos en bases de datos antiguas es necesario proceder de acuerdo a los siguientes pasos:

1. Creación de las tablas a partir del siguiente *script* de SQL:

```
create table systriggers (name char (20), tabname char (20),
statement char (10), event char (10), callproc char(20),
status char(10)) set 16;

create unique index trigname on systriggers (name);

create index trigtabname on systriggers (tabname);

create table sysprocedur (name char (20), classname char (256),
parameters char (128), returning char (128)) set 15;

create unique index procname on sysprocedur (name);
```

2. Renombrar como se indica a continuación los ficheros físicos ".dat" e ".idx" desde el sistema operativo en el directorio de la base de datos:

```
systri16.dat  →  systrigger.dat
systri16.idx  →  systrigger.idx
syspro15.dat  →  sysprocedu.dat
syspro15.idx  →  sysprocedu.idx
```

3. Modificar la columna "dirpath" de la tabla SYSTABLES como sigue:

```
update systables set dirpath = "sysprocedu" where tabname = "sysprocedur";

update systables set dirpath = "systrigger" where tabname = "systriggers";
```

1.2 Aspectos a tener en cuenta

- Para la creación y el reemplazo de los procedimientos almacenados es necesario utilizar el editor de sentencias SQL (SQL Interactivo) proporcionado con la nueva versión. Por ejemplo:
  - Para la versión de Windows, ejecutando el archivo **csql.exe** que se encuentra en el directorio "c:\multiway\bin".
  - Para la versión de Linux/Unix, utilizando la función **insertjavaproc.**
- Esta nueva versión de CTSQL lleva incorporado el "monitor CTSQL". Para que los clientes puedan conectarse al servidor es necesario que este monitor esté arrancado. Si la versión de CTSQL es superior a

la 3.4 0.0 y no se desea arrancar el monitor deberá definirse la variable de entorno ALLOWNOMONITOR con valor YES en el fichero de configuración del motor (ctsql.ini).

- En Cosmos, la llamada a procedimientos almacenados se realizará mediante el método SqlExec de la clase SqlServer o bien a través de la clase SqlStatement.

  En MultiBase, la ejecución de los procedimientos almacenados se realizará a través de la instrucción “tsql”.

2. Introducción a los procedimientos almacenados y *triggers*

Este apartado tiene por objeto explicar brevemente qué son, cómo se crean y para qué se utilizan los procedimientos almacenados y los *triggers* contra el motor de base de datos CTSQL.

El motor CTSQL podrá interactuar con un *servidor de procedimientos almacenados* (SP), el cual puede ser instalado en otro servidor distinto al CTSQL y así "distribuir" la carga de operaciones entre ellos.

El servidor SP nos ayudará también a disminuir la carga de las estaciones de trabajo del cliente, haciendo que las tareas con mayor sobrecarga se lleven a cabo en nuestro servidor SP, como pueden ser los trabajos en *batch.* Por lo tanto, si ejecutamos procesos de la aplicación mediante procesos almacenados aprovecharemos todos los recursos de hardware disponibles en nuestro servidor.

Los procedimientos almacenados facilitan el desarrollo de nuestras aplicaciones y minimizan el número de modificaciones ante futuros cambios. Así mismo, pueden ser ejecutados como consecuencia de una activación de parte de un *triggers*. Esto nos permitirá administrar la información de la base de datos, manteniéndola consistente, íntegra y segura.

2.1 Procedimientos almacenados

Los procedimientos almacenados son programas que se referencian en la base de datos. En el caso del CTSQL, estos programas están escritos en Java. Al ser referenciados en la base de datos, primeramente se deben crear en el catálogo de la base de datos mediante una instrucción del CTSQL, y posteriormente podrán ser ejecutados desde el programa cliente o como consecuencia de los *triggers*.

Los procedimientos almacenados podrán recibir parámetros de entrada. No podrán interactuar con el usuario a través de una interfaz o pantalla. En cambio, no presentan inconvenientes en escribir en ficheros de textos, XML, etc., o en tablas de la base de datos o en la generación y envío de correos electrónicos, por ejemplo.

2.1.1 Ventajas de los procedimientos almacenados

- Diseño modular y posibilidad de acceder a bases de datos de otros motores mediante la tecnología JDBC.
- Las aplicaciones que acceden a la misma base de datos pueden compartir los procedimientos almacenados, eliminando el código doble y reduciendo el tamaño de las aplicaciones.
- Fácil mantenimiento.
- Cuando un procedimiento se actualiza, los cambios se reflejan automáticamente en todas las aplicaciones, sin necesidad de recompilar los programas. Las aplicaciones son compiladas sólo una vez para cada cliente.
- Los procedimientos almacenados son ejecutados por el servidor, no por el cliente, lo que reduce el tráfico en la red y mejora el rendimiento, especialmente para el acceso del cliente remoto.
- Los procedimientos están almacenados en los servidores y asegurados por las medidas tomadas en la instalación, lo que impide que los usuarios normales puedan modificarlos, ya que, incluso, desconocen su existencia. Éste es un elemento de gran valor en lo que a seguridad respecta.

2.2 Triggers

Los *triggers* permiten "disparar" (ejecutar) procedimientos almacenados cada vez que se realice una acción sobre los datos de una tabla. Esta acción puede consistir en la inserción, modificación o eliminación de un registro.

De esta manera, podemos indicar que se ejecuten acciones sobre los datos de la tabla, o de otras tablas, cada vez que se modifican, agregan o eliminan datos de una tabla.

2.2.1 Ventajas de los *triggers*

Algunos usos de los *triggers* son:

- Generación automática de valores derivados de una columna.
- Prevención de transacciones inválidas.
- Proporciona auditorias sofisticadas.
- Mantener la sincronía en tablas replicadas.
- Generación de estadísticas de acceso.
- Publicar información de los eventos generados por la base de datos, las actividades de los usuarios o de las estructuras SQL que se han ejecutado.
- Actualizar totales de la suma de campos de una tabla en otra.
- El mantenimiento de la aplicación se reduce, los cambios a *triggers* se reflejan automáticamente en todas las aplicaciones que tienen que ver con la tabla sin necesidad de recompilar.

2.2.2 Consideraciones sobre *triggers*

- Los *triggers* no tienen parámetros de entrada. Los únicos valores de entrada con los que pueden trabajar son los del registro que han insertado, modificado o eliminado.
- Los *triggers* no devuelven valores como los procedimientos almacenados. Sólo pueden modificar otras tablas o los mismos valores del registro agregado o modificado (obviamente, el eliminado no).
- Hay que tener especial cuidado con los *triggers* recursivos, es decir, aquellos que puedan realizar operaciones que lancen nuevos *triggers*.

2.2.3 Tipos de *triggers*

Dependiendo de la acción sobre la cual queremos que actúen, se pueden crear tres tipos de *triggers*:

- Al insertar un registro.
- Al modificar un registro.
- Al eliminar un registro.

Cada uno tipo de estos tipos se puede dividir a su vez en dos subtipos: antes y después de la acción. En consecuencia, podemos disponer de hasta seis tipos distintos de *triggers:*

- BEFORE INSERT. Antes de insertar un registro.
- AFTER INSERT. Después de insertar un registro.
- BEFORE UPDATE. Antes de modificar un registro.
- AFTER UPDATE. Después de modificar un registro.
- BEFORE DELETE. Antes de eliminar un registro.
- AFTER DELETE. Después de eliminar un registro.

3. Instalación del servidor de procedimientos almacenados

3.1 Requisitos previos

- Motor CTSQL iniciado.

  [Para configurar el CTSQL en Windows (a partir de la versión NT) consulte el Anexo IV del documento: Particularidades sobre la puesta en marcha del motor de base de datos en arquitectura cliente-servidor.]

- Máquina virtual Java 1.3.x.
- Arrancar MONITOR CTSQL iniciado, si la versión del CTSQL lo requiere.

3.2 Servidor de procedimientos almacenados

El servidor de procedimientos almacenados es un paquete Java que se encarga de comunicarse con el servidor CTSQL y de gestionar el alta, la baja, la modificación y la ejecución de procedimientos almacenados escritos en Java.

Funciona como un servicio que escucha en un puerto determinado y espera a que un servidor CTSQL se conecte con él para realizar operaciones con procedimientos almacenados. Por cada sesión CTSQL que se conecte al servidor, lanzará un *thread*, que será el que se ocupe de la comunicación con el CTSQL, quedando libre el servidor de procedimientos almacenados para aceptar nuevas conexiones.

3.3 Puesta en ejecución del servidor de procedimientos almacenados (SP)

Existen tres modos de lanzar el servidor de procedimientos almacenados:

- La primera opción, y la más recomendable, en Windows, es mediante la utilidad **storedserverconf.**

  Este programa está situado en el directorio "bin" del directorio donde está instalado el servidor de procedimientos almacenados, que en Windows por defecto es "c:\Multiway".

  En la versión para Linux/Unix este programa no existe.

- Empleando el programa **storedserver.exe,** situado en el mismo directorio que el anterior para la versión de Windows .

  En la versión de Linux/Unix se encuentra en el directorio $TRANSDIR/bin. Ésta es la opción que debe utilizarse en estos sistemas operativos.

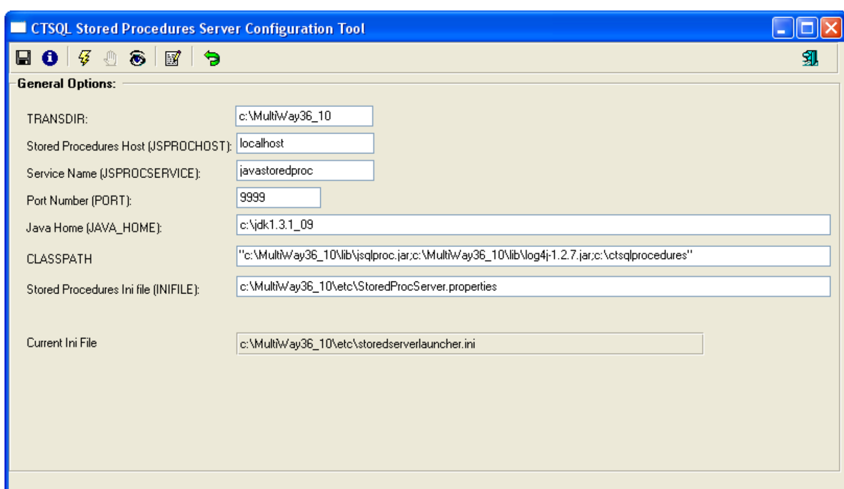
- Directamente desde la línea de comando.

A continuación vamos a ver en detalle cada una de estas posibilidades.

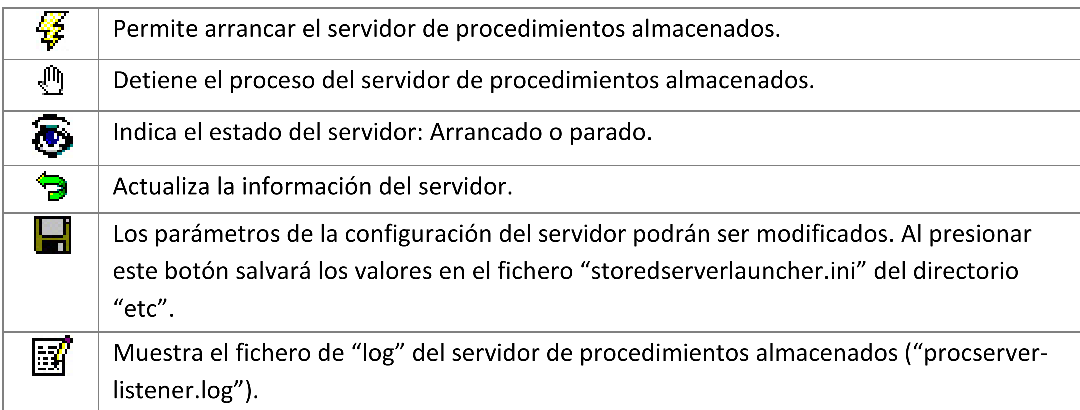
a) Utilidad storedserverconf

El programa "storedserverconf.exe" permite parar y arrancar el servidor de procedimientos almacenados, comprobar si está activo o inactivo y cambiar parámetros de configuración del servidor, tales como la TRANSDIR, el *host* donde está escuchando el servidor de procedimientos almacenados, el servicio, el puerto, la JAVA_HOME, el CLASSPATH y el fichero de propiedades del servidor de procedimientos almacenados. Esta interfaz almacenará dichos valores en el archivo INI indicado en "Current Ini File".

Ésta es la manera más recomendable para manejar y configurar el servidor de procedimientos almacenados, ya que internamente utiliza las dos posibilidades que se van a explicar a continuación, pero empleando una interfaz gráfica.



Las distintas acciones que se pueden ejecutar desde esta interfaz gráfica son las siguientes:



	Permite arrancar el servidor de procedimientos almacenados.
	Detiene el proceso del servidor de procedimientos almacenados.
	Indica el estado del servidor: Arrancado o parado.
	Actualiza la información del servidor.
	Los parámetros de la configuración del servidor podrán ser modificados. Al presionar este botón salvará los valores en el fichero "storedserverlauncher.ini" del directorio "etc".
	Muestra el fichero de "log" del servidor de procedimientos almacenados ("procserver-listener.log").

b) Programa storedserver.exe

Este programa tiene los parámetros: *start, stop, status* y *versión* para, respectivamente, arrancar, parar, mostrar el estado y mostrar la versión del servidor de procedimientos almacenados. Utiliza el fichero de conexión "storedserverlauncher.ini" (al igual que en la opción anterior), y se encuentra en el directorio "etc".

o

```
Usage:
storedserver:
    <-start> | <-stop> | <-status> | <-version>
```

c) Desde la línea de comando

```
java.exe com.transtools.sql.ProcServerListener –inifile
C:\MultiWaySP\etc\StoredProcServer.properties -port 9999
```

Esta forma de ejecutar el servidor de procedimientos almacenados es la menos recomendada.

3.3.1 Fichero storedserverlauncher.ini

El contenido del fichero de conexión al servidor de procedimientos almacenados, que se encuentra en el directorio “etc”, es el siguiente:

```
[Public Environment]

TRANSDIR=c:\MultiWaySP

JSPROCHOST=localhost

JSPROCSERVICE=javastoredproc

PORT=9999

JAVA_HOME=C:\jdk1.3.1_09

INIFILE=c:\MultiWaySP\etc\StoredProcServer.properties

CLASSPATH="c:\MultiWaySP\lib\jsqlproc.jar;c:\MultiWay\
lib\log4j-1.2.7.jar; c:\\ctsqlprocedures"
```

La variable JSPROCHOST contiene el nombre de la máquina, o la IP, donde se encuentra instalado el servidor de procedimientos almacenados.

La variable JSPROCSERVICE indica el nombre del servicio asignado al servidor de procedimientos almacenados. Este servicio debe ser dado de alta en el fichero “services” del servidor.

La variable PORT indica el puerto utilizado por el servidor de procedimientos almacenados en el fichero “services” del servidor. El nombre del servicio es el valor que tiene la variable JSPROCHOST y el puerto el valor que tenga la variable PORT.

La variable JAVA_HOME indica dónde está instalado Java.

La variable INIFILE indica cuál es el fichero de propiedades que utiliza el servidor de procedimientos almacenados.

La variable CLASSPATH indica el CLASSPATH que utilizará el servidor de procedimientos almacenados. Se incluyen: “jsqlproc.jar” y “log4j1.2.7.jar”. El CLASSPATH debe ir siempre entre comillas.

3.3.2 Fichero StoredProcServer.properties

El servidor de procedimientos almacenados ejecuta la clase ProcServerListener, pasándole como parámetros el fichero de propiedades y el puerto por el que escuchará.

El contenido del fichero de propiedades StoredProcServer.properties, que se encuentra en el directorio “etc”, será el siguiente:

```
JSTOREDPROCEDURESPATH=c:\\ctsqlprocedures

LOGGERACTIVE=FALSE
```

La variable JSTOREDPROCEDURESPATH indica el directorio del servidor donde se almacenarán los procedimientos almacenados.

El instalador creará por defecto el *path* indicado en dicha variable. Si se modifica su valor será necesario reiniciar el servidor de procedimientos almacenados para que los cambios tengan efecto.

Es importante tener en cuenta que el *path* debe indicarse con doble carácter “\”, ya que para Java éste es un carácter de escape (solo si el servidor de procedimientos almacenado está en un servidor Windows).

La variable LOGGERACTIVE indica si se va a generar o no un fichero de *log.* Los valores posibles son TRUE o FALSE. Si el valor es TRUE se generará el fichero en el directorio “*c:\tmp\procserverlistener.log* o “*\tmp\procserverlistener.log”,* dependiendo del sistema operativo en el que esté instalado el servidor *,* mientras que si su valor es FALSE o no se define la variable, no se generará ningún fichero de *log*. Su valor por defecto es FALSE.

3.3.3 Fichero ctsql.ini

El contenido del fichero de configuración del CTSQL se encuentra en el directorio “etc”.

```
[Private Environment]
TRANSDIR=c:\MultiWaySP
DOSNAME=OFF
JSPROCHOST=localhost
JSPROCSERVICE=javastoredproc
JSSUPPORT=ON
```

La variable TRANSDIR indica el directorio donde está instalado el CTSQL

La variable DOSNAME indica si las bases de datos que se creen con el CTSQL junto con las tablas del catálogo NO tendrán la limitación de 8 caracteres en su nombre. (DOSNAME = OFF).

Mediante las variables JSPROCHOST y JSPROCSERVICE estamos indicando donde encontrará el servidor de procedimientos almacenados.

La variable JSSUPPORT habilita y deshabilita la posibilidad de interactuar con el servidor de procedimientos almacenados.

Para que los triggers y los procedimientos almacenados funcionen, la variable de entorno JSSUPPORT deberá tener el valor “ON”.

4. Procedimientos almacenados en CTSQL

Como hemos mencionado anteriormente, es necesario registrar los procedimientos almacenados en la base de datos. Para ello emplearemos las siguientes cuatro instrucciones SQL incorporadas al motor, para crear, eliminar, modificar y ejecutar los procedimientos almacenados que utilizaremos.

4.1 Creación de procedimientos almacenados

Cosmos

```
INSERT JAVA PROCEDURE <procname> FROM <filename> IN <package>
```

Ejemplo:

```
Insert java procedure myproc1 from “c:\com\tt\test\myproc1.class” in
“com.tt.test.myproc1”
```

MultiBase Linux/Unix

```
insertjavaprocedure("<procname>", "<filename>", "<package>", 1).
```

La instrucción enviará un fichero previamente compilado (clase Java) al servidor de procedimientos almacenados.

Si consigue enviarlo correctamente, lo da de alta en el catálogo en la tabla SYSPROCEDUR.

Si no consigue enviarlo correctamente, devuelve un mensaje de error indicando la causa por la que no se dio de alta.

4.2 Ejecución directa de procedimientos almacenados

Cosmos

```
CALL JAVA PROCEDURE <procname>([ parameters] );
```

Ejemplo:

```
call java procedure myproc1();
```

MultiBase Linux

```
TSQL "call java procedure <procname>([parameters])";
```

Esta instrucción ejecuta directamente el procedimiento almacenado que se indique. La instrucción comprobará su existencia en el catálogo y comunicará al servidor de procedimientos almacenados la clase Java con la que se encuentra asociado, así como los parámetros con los que se ejecutará.

Si no existe el procedimiento almacenado, devuelve error indicándolo.

4.3 Eliminación de un procedimiento almacenado

Cosmos

```
DROP JAVA PROCEDURE <procname>;
```

Ejemplo:

```
drop java procedure myproc1;
```

MultiBase Linux

```
TSQL "drop java procedure<procname>"
```

La instrucción comprueba la existencia del procedimiento almacenado en el catálogo (tabla SYSPROCEDUR). Si existe, comunicará la decisión de eliminarlo al servidor de procedimientos almacenados. Éste comprobará que existe la clase en el *path* donde se almacenan los procedimientos almacenados y lo eliminará. Si el servidor consigue eliminarlo, será eliminado también del catálogo. Si falla el borrado físico (permisos, inexistencia) o la eliminación del catálogo (inexistencia), devolverá el pertinente error y cancelará la operación.

4.4 Actualización de un procedimiento almacenado

Cosmos

```
REPLACE JAVA PROCEDURE <procname> FROM <filename> IN <package>;
```

Ejemplo:

```
replace java procedure myproc1 from “c:\com\tt\test\myproc2.class” in
“com.tt.test.myproc2”;
```

MultiBase Linux

```
insertjavaproc("<procname>", "<filename>", "<package>", 2)
```

La instrucción enviará un fichero previamente compilado (clase Java) al servidor de procedimientos almacenados. Si el procedimiento almacenado no existe en el catálogo, o si no consigue enviarlo correctamente, devolverá un error indicando la causa por la que no se reemplazó.

4.5 Tabla sysprocedur

Esta tabla gestionará los procedimientos almacenados, y se creará automáticamente al generar la base de datos junto con las demás tablas del catalogo del sistema.

SYSPROCEDUR:

Name	class	Parameters	Returning
Char(20)	Char(256)	Char(128)	Char(128)

PRIMARY INDEX: name

Este catálogo nos sirve para poder encontrar las clases Java asociadas a los procedimientos almacenados.

5. *Triggers* en CTSQL

Como ya hemos mencionado, en nuestro caso el *trigger* será un mecanismo de activación de procedimientos definidos en clases Java que residan en la parte servidora. Cuando un *trigger* invoque a un procedimiento almacenado, pasará a éste como parámetro el tipo y el valor de las columnas de la fila en curso.

5.1 Creación de *triggers*

Cosmos

```
CREATE TRIGGER <trigname>  BEFORE|AFTER INSERT|DELETE|UPDATE ON <tabname>
<procname>;
```

Ejemplo:

```
create trigger mytrigger1 before insert on clientes myproc1;
```

MultiBase Linux

```
TSQL "create trigger <trigname> BEFORE|AFTER INSERT|DELETE|UPDATEon <tabname>
```

La instrucción creará un *trigger,* que ejecutará la clase Java asociada al procedimiento almacenado <procname>, antes o después de que se ejecute una operación de modificación de datos sobre la tabla <tabname>.

Si el procedimiento almacenado no existe en el catálogo, o si no consigue ejecutarlo correctamente, devolverá un error indicando la causa por la que no se ejecutó.

Solamente se podrá insertar un *trigger* por cada acción y tiempo sobre una tabla, es decir, solamente existirá un *trigger* para un AFTER UPDATE TABLA1, aunque se declaren con nombres y procedimientos distintos, ya que tendrán características similares.

5.2 Eliminación de *triggers*

Cosmos

```
DROP TRIGGER <trigname>
```

Ejemplo:

```
drop trigger mytrigger1;
```

MultiBase Linux

```
TSQL "drop trigger <trigname>"
```

La instrucción eliminará el *trigger* que se indica.

5.3 Activación / Desactivación de *triggers*

Cosmos

```
ALTER TRIGGER <trigname> ENABLE|DISABLE;
```

Ejemplo:

```
alter trigger mytrigger1 disable;
```

MultiBase Linux

```
TSQL "alter trigger <trigname> disable|enable"
```

La instrucción activará o desactivará el *trigger* que se indica. Si un *trigger* se encuentra desactivado, no ejecutará el procedimiento almacenado asociado a él.

5.4 Tabla systriggers

Para gestionar los *triggers* nos valdremos de la tabla “systriggers” incorporada en nuestro catálogo de la base de datos, cuya estructura es la siguiente:

Name	tabname	Statement	when	callproc	status
Char(20)	Char(20)	Char(10)	Char(10)	Char(30)	Char(10)

6. Creando procedimientos almacenados

6.1 Clase AbstractSqlProcedure

La clase **AbstractSqlProcedure** es de la que derivan los procedimientos almacenados y la que se tiene que utilizar como base para la creación de procedimientos almacenados. Se encuentra declarada en el paquete **com.transtools.sql,** por lo tanto, deberá incluirse este paquete en la realización de nuestros procedimientos almacenados.

El paquete "com.transtools.sql" se encuentra incluido en el fichero "jsqlproc.jar", que está presente en el directorio "lib" de la distribución.

Un procedimiento almacenado debe declarar el método "run()", ya que éste es exportado por la clase "AbstractSqlProcedure". Dicho método será el método de entrada del procedimiento almacenado.

6.1.1 Métodos de la clase AbstractSqlProcedure

Exporta los siguientes métodos públicos:

- public SqlArgument getArgument(int index)
  Devuelve el enésimo parámetro pasado al procedimiento almacenado.

- public SqlArgument getArgument(string colName)
  Devuelve el enésimo parámetro pasado al procedimiento almacenado.

- public String getArgumentName(int index)
  Devuelve el nombre del enésimo parámetro pasado al procedimiento almacenado.

- public int getArgumentCount()
  Devuelve el número de parámetros pasados al procedimiento almacenado.

- Connection getConnection()
  Devuelve un objeto de la clase "Connection", que se corresponde con la conexión que tiene establecida con el servidor CTSQL.

- public int isTrigger()
  Devuelve TRUE si el procedimiento almacenado ha sido lanzado desde un *trigger.*

- public int isTriggerBefore()
  Devuelve TRUE si el procedimiento almacenado ha sido lanzado desde un *trigger* declarado como BEFORE.

- public int isTriggerAfter()
  Devuelve TRUE si el procedimiento almacenado ha sido lanzado desde un *trigger* declarado como AFTER.

- public int isTriggerInsert()
  Devuelve TRUE si el procedimiento almacenado ha sido lanzado desde un *trigger* declarado en una operación INSERT.

- public int isTriggerDelete()
  Devuelve TRUE si el procedimiento almacenado ha sido lanzado desde un *trigger* declarado en una operación DELETE.

- public int isTriggerUpdate()
  Devuelve TRUE si el procedimiento almacenado ha sido lanzado desde un *trigger* declarado en una operación update.

Como ya hemos mencionado, cuando un *trigger* invoque a un procedimiento almacenado, le pasará como parámetro el tipo (mediante "isTriggerBefore()", "isTriggerInsert()", etc. podremos saber qué tipo de *trigger* se ha "disparado") y el valor de las columnas de la fila en curso, para lo que haremos uso del método "getArgument(i)", teniendo en cuenta las siguientes consideraciones respecto del tipo de *trigger:*

Operación	BEFORE isTriggerBefore()	AFTER isTriggerAfter()
INSERT isTriggerInsert()	Todos los campos son null.	Valores que serán insertados cuando se complete la instrucción.
UPDATE isTriggerUpdate()	Valores originales de la fila antes de la actualización.	Nuevos valores que serán escritos cuando se complete la orden.
DELETE isTriggerDelete()	Valores antes del borrado de la fila.	Todos los campos son null.

7. Ejemplos

Para realizar nuestros primeros ejemplos debemos comprobar que tenemos arrancado el motor CTSQL y el monitor (si corresponde) y el servidor de procedimientos almacenados. Realizamos una conexión y creamos una nueva base de datos, verificando que en el catálogo de nuestra base de datos se encuentren las tablas “sysprocedur” y “systriggers”.

Trabajaremos, por ejemplo, con el paquete “com.tt.test”, por lo que los procedimientos Java que creemos de aquí en adelante lo haremos en el directorio “c:\com\tt\test”.

Escribiremos nuestro código en un editor de texto o en cualquier otro IDE. Seguidamente deberemos compilar los fuentes Java, verificando que se generen los ficheros “.class” correspondientes, que serán los que utilizará nuestro servidor de procedimientos almacenados.

Recordemos que debemos de incluir el fichero “jsqlproc.jar” en el CLASSPATH para que al compilar encuentre la clase “AbstractSqlProcedure” y no se produzcan errores.

Resumiendo:

- Arrancar CTSQL y monitor (si es necesario).
- Arrancar SP Server (si es necesario).
- Escribir y realizar la compilación del procedimiento correspondiente.

Para procedimientos almacenados:

- Registrar el procedimiento en las tablas del catálogo “sysprocedur”.
- Realizar la llamada al procedimiento desde la aplicación cliente o desde el CTSQL.

Para *triggers:*

- Registrar el procedimiento en las tablas del catálogo “sysprocedur”.
- Activar un *trigger* para que ejecute un procedimiento almacenado determinado.
- Realizar la operación en la tabla para la cual se ha activado el *trigger*.

Ejemplo 1

En este primer ejemplo realizaremos nuestro primer procedimiento almacenado. El código será el siguiente:

```
package com.tt.test;


import com.transtools.sql.*; //Paquete necesario para poder
utilizar la clase AbstractSqlProcedure

import java.io.*;

public class myproc1 extends AbstractSqlProcedure {


  public myproc1() {

  }

//Este método es obligatorio para programar los procedimiento almacenados.

//Implementa el método abstracto de la clase AbstractSqlProcedure
```

```
 public int run() {
 try {
        FileWriter fichero = new FileWriter("c:/tmp/myproc1.log",true);
      BufferedWriter buffer = new BufferedWriter(fichero);
                      PrintWriter fichLog= new PrintWriter(buffer);
                fichLog.println("Mi primer procedimiento almacenado");
        fichLog.println("Hay " + getArgumentCount() + " argumentos");
        fichLog.close();
 }
 catch (Exception e) {
                  e.printStackTrace();
              }
 return (1);
  }
}
```

Posteriormente, insertamos el procedimiento almacenado en la tabla “sysprocedur”:

```
insert java procedure myproc1 from "c:\com\tt\test\myproc1.class" in
"com.tt.test.myproc1";
```

Aquí comprobaremos que el fichero “myproc1.class” se encuentre físicamente donde lo hayamos definido en la variable de entorno JSTOREDPROCEDURESPATH.

Ejecutamos el procedimiento almacenado:

```
call java procedure myproc1();
```

Comprobamos el contenido del fichero de texto “C:\tmp\myproc1.log”.

Ejemplo 2

En este ejemplo realizaremos nuestro primer *trigger*. El código será el siguiente:

```
package com.tt.test;


import com.transtools.sql.*;
import java.io.*;


public class myproc2 extends AbstractSqlProcedure {


 public myproc2() {
 }


 public int run() {
```

```
try {
        FileWriter fichero = new FileWriter("c:/tmp/myproc2.log",true);
     BufferedWriter buffer = new BufferedWriter(fichero);
                     PrintWriter fichLog= new PrintWriter(buffer);
     fichLog.println("Mi primer trigger");
        fichLog.println("Hay " +  getArgumentCount() +  " argumentos" );
        if (isTrigger() ) {
          fichLog.println("Estoy en un trigger");
               }
        if (isTriggerAfter()) {
               if (isTriggerDelete()){
                      fichLog.println("Es AFTER - DELETE");
               } else
               if (isTriggerUpdate()) {
                      fichLog.println("Es AFTER - UPDATE");
               } else {
                      fichLog.println("Es AFTER - INSERT");
               }
               }
        else {
               if (isTriggerDelete()){
                      fichLog.println("Es BEFORE - DELETE");
               } else
               if (isTriggerUpdate()) {
                      fichLog.println("Es BEFORE - UPDATE");
               } else {
                      fichLog.println("Es BEFORE - INSERT");
               }
        }
fichLog.close();
      }
 catch (Exception e) {
                  e.printStackTrace();
               }
 return(1);
 }
        }
```

Posteriormente, insertamos el procedimiento almacenado en la tabla “sysprocedur”:

```
insert java procedure myproc2 from "c:\com\tt\test\myproc2.class" in
"com.tt.test.myproc2";
```

Aquí comprobaremos que el fichero “myproc2.class” se encuentre físicamente donde lo hayamos definido en la variable de entorno JSTOREDPROCEDURESPATH y que se haya insertado el registro correspondiente en la tabla “sysprocedur”.

Posteriormente creamos la tabla provincias:

```
create table  provincias (
    provincia         SMALLINT     NOT NULL LABEL "Cod. Provincia",
    descripcion       CHAR(20)     LABEL "Provincia",
    prefijo           SMALLINT     LABEL "Prefijo"
    )
PRIMARY KEY (provincia);
```

Y agregamos un *trigger* que se ejecutará después de cada inserción de registros en la “provincias”, indicando que el procedimiento almacenado “myproc2” será el que se ejecute en este caso:

```
create trigger mytrigger1 after insert on provincias myproc2;
```

Para comprobar que esto ha surgido efecto, realizamos la verificación correspondiente sobre la tabla “systriggers”.

Por último, realizamos una inserción en la tabla “provincias”, por ejemplo:

```
insert into provincias values (1,”CORDOBA”,32)
```

Comprobamos el contenido del fichero de texto “C:\tmp\myproc2.log” para este caso.

Ejemplo 3

Ahora veremos un ejemplo de cómo recoger los valores de la fila en curso después de un INSERT en la tabla “provincias”, para mostrar cómo se utilizan los métodos “getArgument(i)” y “getArgumentName(i)”.

```
package com.tt.test;
import com.transtools.sql.*;
import java.io.*;
import java.sql.*;

public class myproc3 extends AbstractSqlProcedure {

  public myproc3() {
  }
  public int run() {
try {
        int i = 0;
```

```
        FileWriter fichero = new FileWriter("c:/tmp/myproc3.log",true);
 BufferedWriter buffer = new BufferedWriter(fichero);
        PrintWriter fichLog= new PrintWriter(buffer);
 fichLog.println("Mi segundo trigger");
 fichLog.println("Hay " +  getArgumentCount() +  " argumentos" );
        while (i < getArgumentCount()) {
            fichLog.println("Nombre columna ("+i+"): " + getArgumentName(i) );
            fichLog.println("Valor columna  ("+i+"): " +
                            getArgument(i).getSqlArgument());
            fichLog.println("Tipo Jdbc  ("+i+"): " +
                            getArgument(i).getJdbcType());
            i++;
 }
        fichLog.close();
    }
 catch (Exception e) {
              e.printStackTrace();
        }
        return (1);
        }
}
```

Insertamos nuevamente, el procedimiento almacenado en la tabla sysprocedur:

```
insert java procedure myproc3 from "c:\com\tt\test\myproc3.class" in
"com.tt.test.myproc3";
```

Aquí comprobaremos que el fichero "myproc3.class" se encuentre físicamente donde lo hayamos definido en la variable de entorno JSTOREDPROCEDURESPATH y que se haya insertado el registro correspondiente en la tabla "sysprocedur".

Borramos el *trigger* AFTER INSERT creado anteriormente y creamos uno nuevo asociándole el procedimiento almacenado "myproc3":

```
drop trigger mytrigger1;
create trigger mytrigger1 after insert on provincias myproc3;
```

Para comprobar que esto ha surgido efecto, realizamos la verificación correspondiente sobre la tabla "systriggers".

Volvemos a realizar un INSERT sobre la tabla:

```
insert into provincias values (2,"MURCIA",54);
```

El contenido del fichero “myproc3.log” será el siguiente:

```
Mi segundo trigger
Hay 5 argumentos
Tipo Jdbc  (0): 5
Nombre columna  (0): provincia
Valor columna  (0): 2
Tipo Jdbc  (1): 1
Nombre columna  (1): descripcion
Valor columna  (1): MURCIA
Tipo Jdbc  (2): 5
Nombre columna  (2): prefijo
Valor columna  (2): 54
Tipo Jdbc  (3): 4
Nombre columna  (3): (tabid)
Valor columna  (3): 150
Tipo Jdbc  (4): 4
Nombre columna  (4): (rowid)
Valor columna  (4): 3
```

Como podemos apreciar, la tabla “provincias” tiene 3 campos (que hemos definido mediante el CREATE), pero el trigger le pasa como parámetro al procedimiento almacenado, además de las columnas de la fila que procesa, el “tabid” de la tabla y el “rowid” de fila que está siendo tratada.

Ejemplo 4

En este ejemplo mostraremos cómo realizar un *log* sobre las altas, bajas y modificaciones llevadas a cabo sobre la tabla “provincias”. Para ello, crearemos una nueva tabla con la misma estructura que “provincias”, agregando 3 campos más: el *rowid*, el *tabid* y el tipo de instrucción SQL que se ha ejecutado.

```
create table  log_prov (
    provincia          SMALLINT     NOT NULL LABEL "Cod. Provincia",
    descripcion        CHAR(20)     LABEL "Provincia",
    prefijo            SMALLINT     LABEL "Prefijo",
    mytabid            INTEGER,
    myrowid            INTEGER,
    accion             CHAR(15)
    )
```

El procedimiento contempla todos los posibles tipos de datos del SQL, pudiéndose adaptar muy fácilmente a cualquier otra tabla y no solamente a la de “provincias”.

```
package com.tt.test;
```

```
import com.transtools.sql.*;
import java.sql.*;
import java.util.GregorianCalendar;
import java.util.Calendar;

public class myproc4 extends AbstractSqlProcedure {

 public myproc4() {
 }
 public int run() {
        int i = 0;
        boolean isNull = false;
               String action="";
        PreparedStatement pstmt = null;
               if (isTriggerAfter()) {
               if (isTriggerDelete()){
                      action = new String("AFTERDELETE");
               } else
               if (isTriggerUpdate()) {
                      action = new String("AFTERUPDATE");
               } else { //isTriggerInsert
                      action = new String("AFTERINSERT");
               }
                     }
                else {
               if (isTriggerDelete()){
                      action = new String("BEFOREDELETE");      } else
               if (isTriggerUpdate()) {
                      action = new String("BEFOREUPDATE");
               } else {//isTriggerInsert
                      action = new String("BEFOREINSERT");
               }
                      }

               try {
               pstmt = getConnection().prepareStatement("insert into log_prov
values (?,?,?,?,?,?)");
```

```
                    while (i < getArgumentCount()) {
                if (getArgument(i).getSqlArgument() == null)
                    pstmt.setNull(i + 1, getArgument(i).getJdbcType()
);
                else {
                    switch (getArgument(i).getJdbcType()){
                        case Types.CHAR:
                            pstmt.setString( i + 1,
(String)getArgument(i).getSqlArgument());
                            break;
                        case Types.INTEGER:
                            pstmt.setInt( i + 1, ((Inte-
ger)getArgument(i).getSqlArgument()).intValue() );
                            break;
                        case Types.SMALLINT:
                            pstmt.setShort(i + 1,
((Short)getArgument(i).getSqlArgument()).shortValue());
                            break;
                        case Types.TIME:
                            pstmt.setTime(i + 1, convertTo-
Time(getArgument(i).getSqlArgument()));
                            break;
                        case Types.DECIMAL:
                            pstmt.setDouble(i + 1, ((Dou-
ble)getArgument(i).getSqlArgument()).doubleValue());
                            break;
                        case Types.DATE:
                            pstmt.setDate(i + 1, convertTo-
Date(getArgument(i).getSqlArgument()));
                            break;
                        case Types.TIMESTAMP:
                            pstmt.setTimestamp(i+1, convertTo-
DateTime(getArgument(i).getSqlArgument()));
                            break;
                        default:
                            break;
                    }
                }
                i++;
            }
```

```
                pstmt.setString( i + 1 , action);
                pstmt.execute();
        } catch (SQLException e) {
                                }
        finally{
        if (pstmt != null)
                try {  pstmt.close();
                        } catch (SQLException e1) {
                }
                        }
 return (1);
 }
 private java.sql.Time convertToTime(Object object) throws SQLException {
         Calendar cal = (Calendar) object;
        if (object == null) {
                return null;
        }
        Time timevalue = new Time(1);
                 return new java.sql.Time(cal.getTime().getTime());
 }

 private java.sql.Date convertToDate(Object object) {
         Calendar cal = (Calendar) object;
        Date datevalue = new Date(1);

        if (object == null) {
                return null;
        }
        if (object instanceof java.util.Date) {
                        cal.setTime((java.util.Date) object);
        }
        return new java.sql.Date(cal.getTime().getTime());


 }
 private java.sql.Timestamp convertToDateTime(Object object) throws SQLExcep-
tion {
         Calendar cal = (Calendar) object;
```

```
        if (object == null) {
               return null;
        }
        Timestamp datetimevalue = new Timestamp(1);
        return new java.sql.Timestamp(cal.getTime().getTime());
 }

}
```

Como siempre, insertamos el procedimiento almacenado en la tabla “sysprocedur”:

```
insert java procedure myproc4 from "c:\com\tt\test\myproc4.class" in
"com.tt.test.myproc4";
```

A continuación creamos los *triggers* para realizar la auditoría sobre la tabla, asociándoles a todos el procedimiento “myproc4”:

```
drop trigger mytrigger1;  //elimino ya que no puede haber dos triggers sobre
after insert provincias en este caso
create trigger mytrigger1 before update on provincias myproc4;
create trigger mytrigger2 before delete on provincias myproc4;
create trigger mytrigger3 after insert on provincias myproc4;
create trigger mytrigger4 after update on provincias myproc4;
```

Realizamos una operación de inserción, una de modificación y luego otra de modificar, a fin de probar la ejecución de los cuatro *triggers* agregados anteriormente:

```
insert into provincias values(5,"ALAVA",88);
update provincias set descripcion="MADRID",prefijo=110 where provincia=5;
delete from provincias where provincia=5;
```

Después de realizar las operaciones SQL anteriores, la tabla “log_prov” tendrá el siguiente aspecto:

```
5|ALAVA|88|150|5|AFTERINSERT|
5|ALAVA|88|150|5|BEFOREUPDATE|
5|MADRID|110|150|5|AFTERUPDATE|
5|MADRID|110|150|5|BEFOREDELETE|
```

Ejemplo 5

En este ejemplo veremos cómo lanzar un procedimiento almacenado desde una aplicación Cosmos, con la intención de mostrar cómo se podrían utilizar los procedimientos almacenados como trabajos en *batch*.

Para ello convertiremos el módulo “facturar” del aplicativo de ejemplo “Cosmos Almafac” a un procedimiento almacenado en Java.

```
package com.tt.test;
import com.transtools.sql.*;
import java.sql.*;
```

```
import java.io.*;
public class facturar extends AbstractSqlProcedure {
    public facturar() {
    }

    public int run() {
    try {

         FileWriter fichero = new FileWriter("c:/tmp/facturar.log",true);
         BufferedWriter buffer = new BufferedWriter(fichero);
         PrintWriter fichLog= new PrintWriter(buffer);
         fichLog.println("Hay " + getArgumentCount() + " argumentos");
         String condicionExtra="";
         fichLog.println("Argumento:" + getArgument(0).getSqlArgument());
         String numcli= "" + getArgument(0).getSqlArgument();
         if (numcli.compareTo("0") != 0) {
             condicionExtra= " and clientes.cliente = " + numcli;
             fichLog.println(condicionExtra);
         }

         String cliente="";
         String prevCliente="";
         double totalGeneral=0;
         double totalCliente=0;
         Connection connection = this.getConnection();

         PreparedStatement pStmtUpdateClientes  = null;
         PreparedStatement pStmtUpdateAlbaranes = null;
         PreparedStatement pStmtCursorClientes  = null;
         String sqlUpdateClientes= "update clientes set "+
                                   "total_factura = ? " +
                                   "where cliente = ? ";
         String sqlUpdateAlbaranes= "update albaranes set "+
                                    "estado = 'S' "+
                                    "where albaran = ? ";
```

```
String sqlCursorClientes= "select clientes.cliente, albara-
nes.albaran"+
                          ", sum(precio * (1 - descuento/100) * can-
tidad)"+
                          " from clientes, albaranes, lineas"+
                          " where estado != 'S'" +
                          condicionExtra +
                          " and clientes.cliente =
albaranes.cliente"+
                          " and albaranes.albaran = lineas.albaran"+
                          " group by 1, 2 order by 1, 2";

// Prepares
pStmtUpdateClientes = connection.prepareStatement(sqlUpdateClientes);
pStmtUpdateAlbaranes=connection.prepareStatement(sqlUpdateAlbaranes);
pStmtCursorClientes =connection.prepareStatement(sqlCursorClientes);

ResultSet rs=pStmtCursorClientes.executeQuery();

int pos1=1;
int pos2=2;

 while (rs.next()) {
       int cli= rs.getInt(1);
       cliente= rs.getString(1);
       int albaran= rs.getInt(2);
       double totalAlbaran=rs.getDouble(3);
       pStmtUpdateAlbaranes.setInt(pos1,albaran);
       pStmtUpdateAlbaranes.executeUpdate();
       if (cliente.compareTo(prevCliente) == 0) {
           totalCliente = totalCliente + totalAlbaran;
           fichLog.println("Cliente=:" + prevCliente + " - " + cli-
ente                            + " $ " + new
java.text.DecimalFormat("#,##0.00").format(totalCliente));
       }
       else {
          if ( prevCliente != "") {
              pStmtUpdateClientes.setDouble(pos1,totalCliente);
```

```
                        pStmtUpdateCli-
entes.setInt(pos2,Integer.valueOf(prevCliente).intValue());
                        pStmtUpdateClientes.executeUpdate();
                        fichLog.println("Cliente!=:" + prevCliente + " - " +
cliente + " $ " +
                               new ja-
va.text.DecimalFormat("#,##0.00").format(totalCliente));
                    }
                    totalCliente= totalAlbaran;
                    prevCliente=cliente;
                 }
                 totalGeneral= totalGeneral + totalAlbaran;
         }
         if ( prevCliente != "") {
             pStmtUpdateClientes.setDouble(pos1,totalCliente);
             pStmtUpdateCli-
entes.setInt(pos2,Integer.valueOf(prevCliente).intValue());
             pStmtUpdateClientes.executeUpdate();
          }
         fichLog.println("Total General:  $ " + new
java.text.DecimalFormat("#,##0.00").format(totalGeneral));
         fichLog.close();
  pStmtUpdateClientes.close();
         pStmtUpdateAlbaranes.close();
         pStmtCursorClientes.close();


         }
         catch (Exception e) {
                  e.printStackTrace();
          }
    return(1);
  }
```

Como siempre, insertamos el procedimiento almacenado en la tabla “sysprocedur”:

```
insert java procedure facturar from "c:\com\tt\test\facturar.class" in
"com.tt.test.facturar";
```

Recordemos que en el mecanismo de facturación del proyecto “almafac” se podía realizar una facturación a todos los clientes de la base de datos, así como también facturar a un cliente en particular. Esto nos servirá para mostrar cómo se le pueden pasar parámetros a un procedimiento almacenado; si el parámetro es *null*, se facturará a todos los clientes, si no, en el parámetro se indicará el código de cliente a facturar.

La llamada a este procedimiento almacenado desde Cosmos se hará de la siguiente manera:

```
On command Todos
begin
    facturar(null);
end

On Command UnCliente
objects begin
    cliente as integer
end
begin
  if Sql.SelectWindow
      (
          Self
         ,"select cliente, empresa, total_factura from clientes"
         ,0
         ,"Seleccione Clientes"
         ,1
         ,cliente
      ) then
        if cliente is not null then
            facturar(cliente);
end

Private facturar(clienteOpcional as integer default null)
begin
    Sql.SqlExec("call java procedure facturar(?)", clienteOpcional);
    self.MessageBox("Proceso de facturacion lanzado!!","AVISO");
end
```

También podremos ejecutar este procedimiento como siempre:

```
call java procedure facturar();
```

En este caso hará una facturación a todos los clientes de la base de datos.

```
call java procedure facturar(58);
```

Mediante esta llamada facturará sólo al cliente 58, por ejemplo.

En la aplicación de ejemplo de MultiBase (almacén), el procedimiento almacenado que tenemos que generar será el mismo, y la llamada desde un módulo de MultiBase es la siguiente:

```
database pruebas
main begin
    TSQL "call java procedure facturar(58);"
end main
```