Implementación de Triggers en Firebird/Interbase SQL Server

Objetivos:

- Que el alumno comprenda las opciónes de implementación que tiene toda base de datos y sus implicancias.
- Que el alumno sea capaz de realizar un análisis de los tipos de triggers a implementar en situaciones típicas de la programación de base de datos
- Que el alumno sea capaz de implementar triggers utilizando el lenguaje PL/SQL

Requisitos:

- Servidor Firebird 2.5 previamente instalado
- Cliente gráfico FlameRobin previamente instalado (no excluyente, puede utilizarse interfase linea de comandos isql-fb)
- Que el alumno comprenda/conozca los distintos tipos de objetos de una base de datos Firebird: generators, triggers, stored procedures, exceptions, constraints, etc.

Observaciones:

Los lineamientos generales de este documento, no sólo se aplican a Firebird/Interbase sino también a la programación de triggers y stored procedures en general.

1. Implementación de una base de datos

En líneas generales, hay dos formas extremas de implementar una base de datos:

1.1 El usuario o la aplicación interactúa directamente a las tablas del sistema

Esta forma de implementación implica un acceso directo a las tablas, el usuario tiene permisos suficientes para hacer insert, update, delete, select sobre las mismas. En este caso, no existe ningún tipo de capa de software que actúe a modo de aislar al usuario o la aplicación de la implementación "real" de las tablas del sistema. Solo existen algunos stored procedures para realizar algunos procesos repetitivos, cierres mensuales, etc.. Cualquier cambio que deba hacerse a futuro sobre las tablas del sistema, sus privilegios, etc., impactará directamente sobre los usuarios y las aplicaciones que interactúen con esta base de datos. **Llamaremos a esta opción de implementación, la opción A.**

1.2 El usuario o la aplicación no interactúa directamente con las tablas del sistema

Esta forma de implementación implica un acceso indirecto a las tablas, el usuario o la aplicación no tiene acceso directo sobre las tablas, sino que éstos interactúan a

través de una interfase definida por un conjunto de stored procedures de ejecución, sobre los cuales el usuario tiene permiso de ejecución. Para hacer inserts, updates, deletes sobre las tablas, el usuario utiliza stored procedures de ejecución (execute procedures); para hacer selects el usuario utiliza stored procedures de selección (select procedures). El usuario solo tiene permisos de ejecución sobre determinados stored procedures del sistema, nada más. No tiene conocimiento de la existencia de tablas o vistas (o a lo sumo, si las accede para consultarlas, podría tener permiso select); de esta forma, el usuario o la aplicación tiene una capa de software intermedia que lo aísla de la implementación. Es posible hacer cambios sobre las tablas, vistas y relaciones, mientras que no se afecte la interfase del sistema, no habrá impacto en la aplicación. **Llamaremos a esta opción de implementación, la opción B.**

2. Restricciones

Existen varias operaciones que no pueden realizarse dentro de un trigger o bien no están recomendadas, entre ellas:

-No hacer operaciones de DDL dentro de triggers/stored procedures

-Dentro de un trigger estamos dentro de una transacción, la cual tiene lockeos y recursos asignados; debido a ello, no se puede acceder a la tabla sobre la cual se disparó el evento, es decir, sobre un trigger de before update sobre la tabla factura, no puedo hacer un query (en forma directa o indirecta) sobre la tabla factura, solo puedo manejarme con los alias NEW y OLD, nada mas. Intentar una lectura de la tabla puede implicar que la transacción quede bloqueada.

-Las acciones dentro de los triggers siempre tratan de detectar "lo malo"; el formato general de la programación es:

si <condición> entonces error!

por falso, la transacción continua su curso sin problemas.

-Poner en las relaciones padre los datos que deban ser accedidos desde relaciones hijas (siempre y cuando el tipo de implementación (A o B) de la base de datos lo permita). Por ejemplo, si deseamos controlar la cantidad máxima de items que puede tener una factura, se puede pensar en poner un contador de items en la tabla factura; para que ésta pueda ser actualizado por triggers que corren sobre la tabla de items que es una tabla hija de factura.

-Siempre tratar de detectar el problema en eventos de tipo BEFORE; luego de AFTER, el costo de hacer ROLLBACK de la transacción no es aceptable. Volviendo al ejemplo de la factura y los items de factura, si una factura no puede tener más de 5 items, si utilizamos un trigger de AFTER INSERT sobre la tabla de items y allí controlamos si el contador de items ubicado en la tabla de factura, tiene un valor >= 5, deberemos hacer un ROLLBACK de la transacción[1], pero este ROLLBACK tendrá un costo mucho mayor que si hubiésemos puesto este control en un trigger de BEFORE INSERT, porque en un BEFORE INSERT aun no se considera insertada la tupla. Al hacerlo en un trigger de AFTER INSERT, la tupla ya esta insertada y dicho INSERT deberá "volverse para atrás", provocando un costo mayor en términos de performance.

1 Esto puede hacerse invocando explícitamente a la instrucción ROLLBACK o bien lanzando una EXCEPTION con un mensaje de error.

-En todo trigger de BEFORE, el alias NEW es modificable, puedo cambiar los valores provistos por el usuario. Por ejemplo, si escribo la instrucción:

NEW.ID = 4;

dentro de un trigger de BEFORE (INSERT, UPDATE) el ID tomará el valor de 4 y éste será utilizado dentro de la transacción, independientemente del valor que haya indicado el usuario para este atributo. En cambio, si hacemos esta misma instrucción en un trigger de AFTER (INSERT, UPDATE), ya es tarde, el valor de ID ya fue utilizado y esta instrucción no tiene efecto alguno. La instrucción compila y no emite ningún tipo de error.

-En todo trigger de AFTER, el alias NEW no es modificable (o bien, ya no tiene sentido hacerlo, porque no provocará ningún efecto), no puedo cambiar los valores provistos por el usuario. Ver ejemplo anterior.

-El orden de ejecución dentro de la transacción es el siguiente:

1. Eventos de BEFORE
2. Aplicación de CONSTRAINT's (foreign key, primary key, unique, check, domains, unique index, etc.).
3. Eventos de AFTER

esto es relevante en la programación de triggers, por ejemplo, no tiene sentido controlar en un trigger de AFTER si el valor de una foreign key es correcto, pues ello, ya fue controlado por Firebird anteriormente. Tampoco tiene sentido controlar el valor de una foreign key en un trigger de BEFORE si sabemos que luego lo hará Firebird, a menos que, necesitemos saberlo con anterioridad o bien pretendamos lanzar una EXCEPTION emitiendo un mensaje de error mas entendible para el usuario[2]. La recomendación es "no programar de más", no hacer cosas que ya se hacen en forma automática.

-Los trigger y stored procedures requieren de permisos para hacer sus operaciones. Se deben otorgar permisos SELECT, UPDATE, DELETE, INSERT sobre las tablas y vistas que utiliza el trigger o stored procedure.

-Un objeto trigger o stored procedure puede tener permiso para hacer algo sobre otro objeto, sin que el usuario tenga permisos. Por ejemplo, un usuario puede tener permiso de ejecución de un stored procedure y dicho stored procedure, puede hacer INSERT en un tabla sobre la cual el usuario no tiene permiso. Esto es una característica fundamental en la programación de bases de datos, para implementar correctamente controles basados en triggers en implementaciones de tipo A.

-La instrucción COMMIT solo se usa desde el cliente, no puede hacerse un COMMIT dentro de un stored procedure o trigger.

-Se puede ejecutar ROLLBACK en trigger (cancela la transacción, igual que EXCEPTION, etc.) o stored procedure.

-No se puede hacer un BEGIN TRANSACTION / COMMIT TRANSACTION en trigger o stored procedure.

3. Implementaciones posibles

2 En cuanto a este asunto, la recomendación es utilizar una nomenclatura en el nombrado de objetos, en especial, los constraints, de esta forma, es posible implementar en la aplicación alguna forma de parsing del mensaje de error para emitir un mensaje de error mas entendible por el usuario o permitir que la aplicación tome alguna acción al respecto.

Teniendo en cuenta lo indicado en los puntos anteriores, la programación de triggers requiere de atributos adicionales (generalmente ubicados en tablas padre[3]) para hacer conteos, sumarizaciones, etc. La ubicación física de estos atributos varía acorde con el tipo de implementación:

3.1 **Implementaciones de tipo A**

En este caso, se deben crear nuevas tablas, que las llamaremos "tablas auxiliares"[4] y debemos almacenar dentro de ellas los atributos adicionales que necesitemos para implementar los triggers. Estas tablas auxiliares tendrán la misma clave primaria que la tabla padre con la cual están "conceptualmente" relacionadas y tendrán el mismo comportamiento y contendrán los mismos valores de clave primaria que la tabla padre, en todo momento. Ningún usuario tendrá permisos sobre estas tablas auxiliares[5], solo los triggers y stored procedures que la utilicen tendrán permisos sobre las mismas. Veamos un ejemplo: volviendo al ejemplo de la factura y los items, supongamos que las tablas tienen la siguiente estructura:

factura(**nro**,fecha,monto)
item(**nro,idp**,cantidad,precio)
nro foreign key factura
idp foreign key producto
producto(**idp**,descripcion,stock)

debemos agregar un atributo nuevo, supongamos cantidad de items (ci) y debemos hacerlo sobre una nueva tabla, oculta para el usuario, pues sino, éste podría cambiar a su antojo el valor de ci y el control no tendría sentido. Por lo tanto, debemos tener la siguiente nueva estructura:

factura_aux(**nro**,ci)
factura(**nro**,fecha,monto)
item(**nro,idp**,cantidad,precio)
nro foreign key factura
idp foreign key producto
producto(**idp**,descripcion,stock)

en donde factura_aux es una tabla "paralela" a factura, cuando se inserta en factura, se debe insertar en factura_aux, cuando se borra en factura, se debe borrar en factura_aux, etc. El usuario no debería tener permisos sobre factura_aux, solo deberían tener permisos sobre la misma, los triggers y stored procedures que la utilizan.

---

3 Tablas que están referenciadas por otras tablas en CONSTRAINT's de FOREIGN KEY.
4 Son tablas iguales que cualquier otra tabla de la base de datos.
5 A lo sumo, el usuario podrá tener permiso SELECT sobre estas tablas, se debe impedir que el usuario pueda cambiar los valores de los atributos que son utilizados dentro del control implementado por los triggers.

3.2 **Implementaciones de tipo B**

Tomando el ejemplo anterior, el atributo ci podría incluirse dentro de factura sin problemas, sin necesidad de tablas auxiliares, pues, el usuario no accede directamente a ninguna tabla del sistema. Aquí puede observarse una de las ventajas de la "aislación" a la que me refería anteriormente. Por lo tanto, la estructura podría ser:

factura(**nro**,fecha,monto,ci)
item(**nro,idp**,cantidad,precio)
nro foreign key factura
idp foreign key producto
producto(**idp**,descripcion,stock)

4. Restricciones de implementación

Considere las restricciones que tendría el usuario o la aplicación, si éstas son aceptables; ello permitirá bajar el nivel de complejidad del control a implementar. Por ejemplo, suele facilitar la programación el imponer la restricción de no permitir cambios en claves primarias y esto posiblemente sea aceptable para el usuario y la aplicación. Lo mismo sobre atributos que, por ejemplo, implican un determinado "estado", ¿cuáles son los cambios posibles que pueden hacerse sobre el mismo?.

5. Programación de Triggers

La lógica propuesta es la siguiente:

1. Identificar, según el problema a resolver, las tablas involucradas y si es una implementación de tipo A o B.
2. Armar una grilla en donde ubicar todos los posibles tipos de eventos (INSERT, UPDATE, DELETE considerando además, que pueden ser de BEFORE o AFTER) en las filas y poner en cada columna, cada una de las tablas involucradas. Agregar las tablas auxiliares que se requieran si estamos en una implementación de tipo A.
3. Completar la grilla, escribiendo en cada celda, la acción a realizar para ese tipo de evento y esa tabla en particular en pseudo-código.
4. Comenzar la programación
   4.1 Crear todos los objetos requeridos por la solución propuesta (tablas auxiliares, exceptions, generators, etc.) y que serán usados desde distintos triggers.
   4.2 Por cada celda de la grilla que contiene pseudo-código, crear un trigger, con la siguiente nomenclatura: TRG_{BI | AI | BD | AD | BU | AU}<tabla> donde B significa BEFORE, A significa AFTER, I significa INSERT, D significa DELETE, U significa UPDATE; por ejemplo, si requerimos implementar un trigger de BEFORE INSERT sobre la tabla factura, el nombre del mismo será

TRG_BIFACTURA. De esta forma, será más fácil ubicarlos en la base de datos y ante el nombre del objeto podremos saber que tipo de trigger es.

6. Casos típicos

En la programación de bases de datos, se pueden identificar situaciones típicas, que se detallan a continuación, junto con su análisis correspondiente y su implementación. El análisis e implementación de estos casos típicos nos servirán para luego implementar el resto de reglas semánticas que cada base de datos requiera.

6.1 Implementación Generator sobre tabla para generar ID en forma automática.

Ejemplo, supongamos la tabla:

CLIENTE(**ID**,NOMBRE)

donde id es pk y debe generarse ese numero de forma automática, controlado por el sistema, para ello, se puede usar objeto generator GEN_CLIENTE_ID (que asegura generar id's distintos, pero no necesariamente consecutivos[6]) y lo "seteamos" a valor 0 (así la próxima vez que se llame a gen_id(gen_cliente_id,1)[7] sumará 1 y generará el valor 1):

```
CREATE GENERATOR GEN_CLIENTE_ID;
SET GENERATOR GEN_CLIENTE_ID TO 0;
```

Análisis, que eventos debemos controlar en CLIENTE para controlar el valor de su PK?

EVENTO		CLIENTE
INSERT	BEFORE	ID = GEN_ID(GEN_CLIENTE_ID,1);
	AFTER	
UPDATE	BEFORE	No permitir cambio de ID
	AFTER	
DELETE	BEFORE	
	AFTER	

Implementación:

```
CREATE TABLE CLIENTE
(
  ID integer NOT NULL primary key,
  NOMBRE varchar(50)
```

6 Ello se debe a que, una vez invocada la función gen_id(), se genera un nuevo valor del generator y si la transacción es "vuelta para atrás" (rollback), el valor del generator no es vuelto para atrás; no tiene una lógica transaccional el valor del objeto generator.

7 Puede usar gen_id() para sumar, restar y obtener el valor actual de un generator.

```
);


SET TERM ^;

CREATE TRIGGER TRG_BICLIENTE FOR CLIENTE
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
    /* enter trigger code here */
    NEW.ID = GEN_ID(GEN_CLIENTE_ID,1);
END^

SET TERM ;^
```

se requiere emitir un error/excepción cuando se pretenda cambiar el valor de la columna ID:

```
CREATE EXCEPTION EX_CLIENTE1 'No puede cambiar ID de cliente!';

SET TERM ^;

CREATE TRIGGER TRG_BUCLIENTE FOR CLIENTE
ACTIVE BEFORE UPDATE POSITION 0
AS
BEGIN
    /* Si cambia Id entonces error! */
    IF ( NEW.ID <> OLD.ID ) THEN
        EXCEPTION EX_CLIENTE1;
END^

SET TERM ;^
```

6.2 Implementación Tipo-Subtipo mutuamente exclusivos.

Ejemplo, supongamos que una persona puede ser contratista o empleado, una cosa o la otra, pero no ambas:

PERSONA(**DNI**,APENOM)
CONTRATISTA(**DNI**,VALORHORA)
DNI FK PERSONA
EMPLEADO(**DNI**,SALARIO)
DNI FK PERSONA

Análisis, que eventos debemos controlar en estas tablas para evitar que una persona sea contratista y empleado al mismo tiempo?

EVENTO		PERSONA	CONTRATISTA	EMPLEADO
INSERT	BEFORE		Si ya existe tupla en empleado entonces error!	Si ya existe tupla en contratista entonces error!

	AFTER			
UPDATE	BEFORE		No permitir cambio de DNI (o bien controlar que el nuevo dni no exista en empleado)	No permitir cambio de DNI (o bien controlar que el nuevo dni no exista en contratista)
	AFTER			
DELETE	BEFORE			
	AFTER			

Implementación:

```
CREATE TABLE PERSONA
(
  DNI integer NOT NULL primary key,
  APENOM varchar(100)
);

CREATE TABLE CONTRATISTA
(
    DNI INTEGER NOT NULL PRIMARY KEY,
    VALORHORA DOUBLE PRECISION,
    CONSTRAINT FK_CONTRATISTA_PER FOREIGN KEY (DNI) REFERENCES PERSONA
);

CREATE TABLE EMPLEADO
(
    DNI INTEGER NOT NULL PRIMARY KEY,
    SALARIO DOUBLE PRECISION,
    CONSTRAINT FK_EMPLEADO_PER FOREIGN KEY (DNI) REFERENCES PERSONA
);


CREATE EXCEPTION EX_DNI 'No puede cambiar DNI !';
CREATE EXCEPTION EX_PERSONA 'Debe ser Empleado o Contratista!,no ambos!';

SET TERM ^;

CREATE TRIGGER TRG_BICONTRATISTA FOR CONTRATISTA
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
    /* Si cambia Id entonces error! */
    IF ( EXISTS ( SELECT * FROM EMPLEADO
        WHERE DNI = NEW.DNI ) ) THEN
        EXCEPTION EX_PERSONA;
END^

SET TERM ;^



SET TERM ^;

CREATE TRIGGER TRG_BIEMPLEADO FOR EMPLEADO
```

```
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
    /* Si cambia Id entonces error! */
    IF ( EXISTS ( SELECT * FROM CONTRATISTA
        WHERE DNI = NEW.DNI ) ) THEN
        EXCEPTION EX_PERSONA;
END^

SET TERM ;^

SET TERM ^;

CREATE TRIGGER TRG_BUEMPLEADO FOR EMPLEADO
ACTIVE BEFORE UPDATE POSITION 0
AS
BEGIN
    /* Si cambia Id entonces error! */
    IF ( NEW.DNI <> OLD.DNI ) THEN
        EXCEPTION EX_DNI;
END^

SET TERM ;^



SET TERM ^;

CREATE TRIGGER TRG_BUCONTRATISTA FOR CONTRATISTA
ACTIVE BEFORE UPDATE POSITION 0
AS
BEGIN
    /* Si cambia Id entonces error! */
    IF ( NEW.DNI <> OLD.DNI ) THEN
        EXCEPTION EX_DNI;
END^

SET TERM ;^
```

6.3 Implementación de cardinalidad máxima.

Ejemplo, dada las tablas:

FACTURA(**NRO**,IMPORTE)
DETALLE(**NRO,ID**,CANTIDAD,PRECIO)
NRO FK FACTURA
ID FK PRODUCTO
PRODUCTO(**ID**,DESCR,STOCK)

supongamos que una factura no puede tener mas de 3 productos/items.
La idea es crear una serie de triggers que realicen el conteo de los items que se van asociando a cada factura.

La cuestión pasa por determinar ¿en dónde guardo el contador de items? La cantidad de items es un dato asociado a la factura, pero … aquí la solución varía acorde con el tipo de implementación de la base de datos (A o B):

a) los usuarios acceden directamente a las tablas, es decir, pueden leer y grabar directamente en ellas. Por lo tanto, los usuarios podrían modificar el valor de cantidad de items (CI) a su antojo y nuestro control fallaría[8]. En estos casos, debemos implementar CI en una tabla auxiliar, paralela[9] a FACTURA, FACTURA_AUX y nadie tendrá permisos de UPDATE, INSERT, DELETE sobre FACTURA_AUX, sólo podrán hacer todo tipo de operaciones en FACTURA_AUX los triggers que diseñemos para mantener el valor de CI[10]. Debemos explicitar los permisos que requiere cada objeto[11].

Las tablas a utilizar aquí serían:
AFACTURA(**NRO**,IMPORTE)
ADETALLE(**NRO,ID**,CANTIDAD,PRECIO)
NRO FK FACTURA
ID FK PRODUCTO
APRODUCTO(**ID**,DESCR,STOCK)
AFACTURA_AUX(**NRO**,CI)

Análisis, que eventos debemos controlar en estas tablas para evitar que una factura tenga mas de 3 artículos asociados?

Si hay updates/deletes en cascada, se asume que los mismos también se propagan a factura_aux; sino, no habría que permitir cambios en pk de detalle, factura.

EVENTO		AFACTURA	AFACTURA_AUX	ADETALLE	APRODUCTO
INSERT	BEFORE			Si afactura_aux.ci >= 3 entonces error!	
	AFTER	Insertar tupla en afactura_aux		afactura_aux.ci = afactura_aux.ci + 1	
UPDATE	BEFORE			Si cambia NRO y afactura_aux.ci >= 3 para NRO nuevo entonces error!	
	AFTER			Si cambia NRO, restar 1 en CI para NRO viejo y sumar 1 en CI para NRO nuevo	

8 No por una cuestión de mala programación, sino por una cuestión de falta de seguridad.
9 Me refiero a que tendrá la misma PK, los mismos valores en NRO, las mismas tuplas, etc. será idéntica a FACTURA, pero solo almacenando su PK y CI (en este caso).
10 Es posible dar permiso a un objeto sobre otro objeto. Por Ejemplo, un objeto trigger puede tener permisos de INSERT, UPDATE, DELETE, SELECT sobre el objeto tabla FACTURA_AUX.
11 Esta opción de implementación requiere de una administración de seguridad muy distinta a la otra opción.

DELETE	BEFORE				
	AFTER	Borrar tupla en afactura_aux		afactura_aux.ci = afactura_aux.ci - 1	

Implementación:

```
CREATE EXCEPTION EX_FACTURA 'Una factura no puede tener mas de 3 items!';

CREATE TABLE APRODUCTO
(
    ID INTEGER NOT NULL PRIMARY KEY,
    DESCR VARCHAR(100) NOT NULL,
    STOCK INTEGER
);

CREATE TABLE AFACTURA
(
    NRO INTEGER NOT NULL PRIMARY KEY,
    IMPORTE DOUBLE PRECISION
);

CREATE TABLE AFACTURA_AUX
(
    NRO INTEGER NOT NULL PRIMARY KEY,
    CI INTEGER DEFAULT 0 NOT NULL
);
```

Nota: no es necesario implementar fk con afactura porque se supone que cumplirá con dicha restricción por ser una tabla administrada por triggers.

```
CREATE TABLE ADETALLE
(
    NRO INTEGER NOT NULL,
    ID INTEGER NOT NULL,
    CANTIDAD INTEGER NOT NULL,
    PRECIO DOUBLE PRECISION NOT NULL,
    CONSTRAINT PK_ADETALLE PRIMARY KEY (NRO,ID),
    CONSTRAINT FK_ADETALLE_AF FOREIGN KEY (NRO) REFERENCES AFACTURA,
    CONSTRAINT FK_ADETALLE_AP FOREIGN KEY (ID) REFERENCES APRODUCTO
);


SET TERM ^;

CREATE TRIGGER TRG_AIAFACTURA FOR AFACTURA
ACTIVE AFTER INSERT POSITION 0
AS
BEGIN
    /* Si cambia Id entonces error! */
    INSERT INTO AFACTURA_AUX(NRO,CI) VALUES(NEW.NRO,0);
END^
```

```
SET TERM ;^



SET TERM ^;

CREATE TRIGGER TRG_ADAFACTURA FOR AFACTURA
ACTIVE AFTER DELETE POSITION 0
AS
BEGIN
    /* Si cambia Id entonces error! */
    DELETE FROM AFACTURA_AUX WHERE NRO = OLD.NRO;
END^

SET TERM ;^


SET TERM ^;

CREATE TRIGGER TRG_BIADETALLE FOR ADETALLE
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
    /* Si cambia Id entonces error! */
    IF ( (SELECT CI FROM AFACTURA_AUX
            WHERE NRO = NEW.NRO) >= 3 ) THEN
        EXCEPTION EX_FACTURA;
END^

SET TERM ;^



SET TERM ^;

CREATE TRIGGER TRG_AIADETALLE FOR ADETALLE
ACTIVE AFTER INSERT POSITION 0
AS
BEGIN
    /* Si cambia Id entonces error! */
    UPDATE AFACTURA_AUX SET CI = CI + 1
    WHERE NRO = NEW.NRO;
END^

SET TERM ;^



SET TERM ^;

CREATE TRIGGER TRG_BUADETALLE FOR ADETALLE
ACTIVE BEFORE UPDATE POSITION 0
AS
BEGIN
    /* Si cambia nro y el nuevo nro supera 3 */
    IF ( NEW.NRO <> OLD.NRO AND
```

```
            (SELECT CI FROM AFACTURA_AUX
            WHERE NRO = NEW.NRO) >= 3 ) THEN
        EXCEPTION EX_FACTURA;
END^

SET TERM ;^


SET TERM ^;

CREATE TRIGGER TRG_AUADETALLE FOR ADETALLE
ACTIVE AFTER UPDATE POSITION 0
AS
BEGIN
    /* Si cambio nro, resto el viejo y sumo el nuevo! */
    IF ( NEW.NRO <> OLD.NRO ) THEN BEGIN
        UPDATE AFACTURA_AUX SET CI = CI - 1 WHERE NRO = OLD.NRO;
        UPDATE AFACTURA_AUX SET CI = CI + 1 WHERE NRO = NEW.NRO;
    END

END^

SET TERM ;^



SET TERM ^;

CREATE TRIGGER TRG_ADADETALLE FOR ADETALLE
ACTIVE AFTER DELETE POSITION 0
AS
BEGIN
    /* Si borro detalle, resto uno! */
    UPDATE AFACTURA_AUX SET CI = CI - 1 WHERE NRO = OLD.NRO;

END^

SET TERM ;^
```

Luego solo resta otorgar los permisos que correspondan a los triggers.

b) los usuarios no tienen acceso directo a las tablas, las actualizan a través de stored procedures. Por lo tanto, puedo agregar la columna CI a FACTURA para contar allí la cantidad de items y ello no afectará ni a los stored procedures ya implementados ni a nuestro control[12]. No hace falta implementar ninguna restricción de seguridad.

Las tablas a utilizar aquí serían:
BFACTURA(**NRO**,IMPORTE,CI)
BDETALLE(**NRO,ID**,CANTIDAD,PRECIO)

12 Acá se aprecia una de las ventajas de tener "otra capa de aislación" entre las tablas y los usuarios; podemos hacer cambios, mientras que no cambiemos la interfase externa de los procedures que utilizan los usuarios y las aplicaciones, el sistema continuará funcionando sin problemas.

NRO FK FACTURA
ID FK PRODUCTO
BPRODUCTO(**ID**,DESCR,STOCK)

Análisis, que eventos debemos controlar en estas tablas para evitar que una factura tenga mas de 3 artículos asociados?

EVENTO		BFACTURA	BDETALLE	BPRODUCTO
INSERT	BEFORE	CI=0	Si bfactura.ci >= 3 entonces error!	
	AFTER		bfactura.ci = bfactura.ci + 1	
UPDATE	BEFORE		Si cambia NRO y bfactura.ci >= 3 para NRO nuevo entonces error!	
	AFTER		Si cambia NRO, restar 1 en CI para NRO viejo y sumar 1 en CI para NRO nuevo	
DELETE	BEFORE			
	AFTER		bfactura.ci = bfactura.ci - 1	

Implementación:

```
CREATE EXCEPTION EX_FACTURA 'Una factura no puede tener mas de 3 items!';


CREATE TABLE BPRODUCTO
(
    ID INTEGER NOT NULL PRIMARY KEY,
    DESCR VARCHAR(100) NOT NULL,
    STOCK INTEGER
);

CREATE TABLE BFACTURA
(
    NRO INTEGER NOT NULL PRIMARY KEY,
    IMPORTE DOUBLE PRECISION,
    CI INTEGER DEFAULT 0 NOT NULL
);

CREATE TABLE BDETALLE
(
    NRO INTEGER NOT NULL,
    ID INTEGER NOT NULL,
    CANTIDAD INTEGER NOT NULL,
    PRECIO DOUBLE PRECISION NOT NULL,
    CONSTRAINT PK_BDETALLE PRIMARY KEY (NRO,ID),
    CONSTRAINT FK_BDETALLE_AF FOREIGN KEY (NRO) REFERENCES BFACTURA,
    CONSTRAINT FK_BDETALLE_AP FOREIGN KEY (ID) REFERENCES BPRODUCTO
);
```

```
SET TERM ^ ;
CREATE TRIGGER TRG_BIBFACTURA FOR BFACTURA
BEFORE INSERT POSITION 0
AS
BEGIN
    NEW.CI = 0;
END^
SET TERM ; ^


SET TERM ^ ;
CREATE TRIGGER TRG_BIBDETALLE FOR BDETALLE
BEFORE INSERT POSITION 0
AS
BEGIN
    /* Si cambia Id entonces error! */
    IF ( (SELECT CI FROM BFACTURA
            WHERE NRO = NEW.NRO) >= 3 ) THEN
        EXCEPTION EX_FACTURA;
END^
SET TERM ; ^



SET TERM ^ ;
CREATE TRIGGER TRG_AIBDETALLE FOR BDETALLE
AFTER INSERT POSITION 0
AS
BEGIN
    UPDATE BFACTURA SET CI = CI + 1 WHERE NRO = NEW.NRO;
END^
SET TERM ; ^


SET TERM ^ ;
CREATE TRIGGER TRG_BUBDETALLE FOR BDETALLE
BEFORE UPDATE POSITION 0
AS
BEGIN
    /* Si cambia nro y el nuevo nro supera 3 */
    IF ( NEW.NRO <> OLD.NRO AND
            (SELECT CI FROM BFACTURA
            WHERE NRO = NEW.NRO) >= 3 ) THEN
        EXCEPTION EX_FACTURA;
END^
SET TERM ; ^



SET TERM ^ ;
CREATE TRIGGER TRG_AUBDETALLE FOR BDETALLE
AFTER UPDATE POSITION 0
AS
BEGIN
    /* Si cambio nro, resto el viejo y sumo el nuevo! */
    IF ( NEW.NRO <> OLD.NRO ) THEN BEGIN
```

```
        UPDATE BFACTURA SET CI = CI - 1 WHERE NRO = OLD.NRO;
        UPDATE BFACTURA SET CI = CI + 1 WHERE NRO = NEW.NRO;
    END

END^
SET TERM ; ^


SET TERM ^ ;
CREATE TRIGGER TRG_ADBDETALLE FOR BDETALLE
AFTER DELETE POSITION 0
AS
BEGIN
    /* Si borro detalle, resto uno! */
    UPDATE BFACTURA SET CI = CI - 1 WHERE NRO = OLD.NRO;

END^
SET TERM ; ^
```

6.4 Implementación Entidad Débil-Entidad Fuerte con discriminador ascendente consecutivo.

Ejemplo, sistema de biblioteca, con las siguientes tablas:

LIBRO(**ISBN**,TITULO)
EJEMPLAR(**ISBN,NRO**,UBICACION)
ISBN FK ISBN

donde LIBRO es la entidad fuerte, EJEMPLAR es la entidad débil, NRO es el discriminante; los valores de NRO van de 1..N para cada ISBN de la biblioteca y se pretende que sean correlativos ascendentes. UBICACION indica la ubicación del ejemplar dentro de los estantes de la biblioteca.

La idea es crear una serie de triggers que mantengan el último numero de ejemplar (UE) por cada isbn, para luego, obligar a que el nuevo ejemplar tenga ese número + 1 y así mantener la correlatividad y la ascendencia de los números de ejemplar.
La cuestión pasa por determinar ¿en dónde guardo el ultimo número de ejemplar? El último número de ejemplar es un dato asociado al libro, pero ...
Aquí la solución varía acorde con el tipo de implementación de la base de datos: (ídem anterior)

a) los usuarios acceden directamente a las tablas, por lo tanto, las tablas serían:

ALIBRO(**ISBN**,TITULO)
AEJEMPLAR(**ISBN,NRO**,UBICACION)
ISBN FK ISBN
ALIBRO_AUX(**ISBN**,UE)

Análisis:

EVENTO		ALIBRO	ALIBRO_AUX	AEJEMPLAR
INSERT	BEFORE			Nro = alibro_aux.ue+1
	AFTER	Insertar tupla en alibro_aux, ue=0		alibro_aux.ue = alibro_aux.ue + 1
UPDATE	BEFORE			No permitir cambio en Pk
	AFTER			
DELETE	BEFORE			Si nro <> alibro_aux.ue entonces error! Se borra "de atrás para adelante"
	AFTER	Borrar tupla en alibro_aux		alibro_aux.ue = alibro_aux.ue - 1

Implementación:

```
CREATE EXCEPTION EX_EJEMPLAR 'Se borra a partir del ultimo ejemplar';

CREATE TABLE ALIBRO
(
  ISBN char(20) NOT NULL PRIMARY KEY,
  TITULO varchar(100) NOT NULL
);

CREATE TABLE ALIBRO_AUX
(
    ISBN CHAR(20) NOT NULL PRIMARY KEY,
    UE INTEGER DEFAULT 0 NOT NULL
);


CREATE TABLE AEJEMPLAR
(
    ISBN CHAR(20) NOT NULL,
    NRO INTEGER NOT NULL,
    UBICACION VARCHAR(20),
    CONSTRAINT PK_AEJEMPLAR PRIMARY KEY(ISBN,NRO),
    CONSTRAINT FK_AEJEMPLAR_L FOREIGN KEY(ISBN) REFERENCES ALIBRO
);

SET TERM ^;

CREATE TRIGGER TRG_AIALIBRO FOR ALIBRO
ACTIVE AFTER INSERT POSITION 0
AS
BEGIN
    /* enter trigger code here */
    INSERT INTO ALIBRO_AUX(ISBN,UE) VALUES(NEW.ISBN,0);
END^
```

```
SET TERM ;^


SET TERM ^;

CREATE TRIGGER TRG_ADALIBRO FOR ALIBRO
ACTIVE AFTER DELETE POSITION 0
AS
BEGIN
    /* enter trigger code here */
    DELETE FROM ALIBRO_AUX WHERE ISBN = OLD.ISBN;
END^

SET TERM ;^


SET TERM ^;

CREATE TRIGGER TRG_BIAJEMPLAR FOR AEJEMPLAR
ACTIVE BEFORE INSERT POSITION 0
AS
    DECLARE VARIABLE VUE TYPE OF COLUMN ALIBRO_AUX.UE;
BEGIN
    /* enter trigger code here */
    SELECT UE FROM ALIBRO_AUX WHERE ISBN = NEW.ISBN
    INTO :VUE;
    NEW.NRO = VUE+1;
END^

SET TERM ;^


SET TERM ^;

CREATE TRIGGER TRG_AIAJEMPLAR FOR AEJEMPLAR
ACTIVE AFTER INSERT POSITION 0
AS
BEGIN
    /* enter trigger code here */
    UPDATE ALIBRO_AUX SET UE = UE + 1 WHERE ISBN = NEW.ISBN;
END^

SET TERM ;^



SET TERM ^;

CREATE TRIGGER TRG_BUAJEMPLAR FOR AEJEMPLAR
ACTIVE BEFORE UPDATE POSITION 0
AS
BEGIN
    /* enter trigger code here */
    IF ( NEW.ISBN <> OLD.ISBN OR
         NEW.NRO  <> OLD.NRO ) THEN
         EXCEPTION EX_PK;
```

```
END^

SET TERM ;^


SET TERM ^ ;
CREATE TRIGGER TRG_BDAEJEMPLAR FOR AEJEMPLAR
BEFORE DELETE POSITION 0
AS
    DECLARE VARIABLE VUE TYPE OF COLUMN ALIBRO_AUX.UE;
BEGIN
    /* enter trigger code here */
    SELECT UE FROM ALIBRO_AUX WHERE ISBN = OLD.ISBN
    INTO :VUE;
    IF ( VUE <> OLD.NRO ) THEN EXCEPTION EX_EJEMPLAR;
END^
SET TERM ; ^


SET TERM ^ ;
CREATE TRIGGER TRG_ADAEJEMPLAR FOR AEJEMPLAR
AFTER DELETE POSITION 0
AS
BEGIN
    /* enter trigger code here */
    UPDATE ALIBRO_AUX SET UE = UE - 1 WHERE ISBN = OLD.ISBN;
END^
SET TERM ; ^
```

Luego solo resta otorgar los permisos que correspondan a los triggers.

b) los usuarios no tienen acceso directo a las tablas, las actualizan a través de stored procedures.

BLIBRO(**ISBN**,TITULO,UE)
BEJEMPLAR(**ISBN,NRO**,UBICACION)
ISBN FK ISBN

Análisis:

EVENTO		**BLIBRO**	**BEJEMPLAR**
INSERT	BEFORE	UE=0	Nro = blibro.ue+1
	AFTER		blibro.ue = blibro.ue + 1
UPDATE	BEFORE		No permitir cambio en Pk
	AFTER		
DELETE	BEFORE		Si nro <> blibro.ue entonces error! Se borra “de atrás para adelante”
	AFTER		blibro.ue = blibro.ue - 1

Implementación:

```
CREATE TABLE BLIBRO (
  ISBN char(20) NOT NULL PRIMARY KEY,
  TITULO varchar(100) NOT NULL,
  UE INTEGER DEFAULT 0
);

CREATE TABLE BEJEMPLAR
(
  ISBN char(20) NOT NULL,
  NRO integer NOT NULL,
  UBICACION varchar(20),
  CONSTRAINT PK_BEJEMPLAR PRIMARY KEY (ISBN,NRO),
  CONSTRAINT FK_BEJEMPLAR_L FOREIGN KEY (ISBN) REFERENCES BLIBRO (ISBN)
);


SET TERM ^;

CREATE TRIGGER TRG_BIBLIBRO FOR BLIBRO
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
    NEW.UE=0;
END^

SET TERM ;^


SET TERM ^ ;
CREATE TRIGGER TRG_BIBEJEMPLAR FOR BEJEMPLAR
BEFORE INSERT POSITION 0
AS
    DECLARE VARIABLE VUE TYPE OF COLUMN BLIBRO.UE;
BEGIN
    /* enter trigger code here */
    SELECT UE FROM BLIBRO WHERE ISBN = NEW.ISBN
    INTO :VUE;
    NEW.NRO = VUE+1;
END^
SET TERM ; ^


SET TERM ^ ;
CREATE TRIGGER TRG_AIBEJEMPLAR FOR BEJEMPLAR
AFTER INSERT POSITION 0
AS
BEGIN
    /* enter trigger code here */
    UPDATE BLIBRO SET UE = UE + 1 WHERE ISBN = NEW.ISBN;
END^
SET TERM ; ^
```

```
SET TERM ^ ;
CREATE TRIGGER TRG_BUBEJEMPLAR FOR BEJEMPLAR
BEFORE UPDATE POSITION 0
AS
BEGIN
    /* enter trigger code here */
    IF ( NEW.ISBN <> OLD.ISBN OR
         NEW.NRO  <> OLD.NRO ) THEN
         EXCEPTION EX_PK;
END^
SET TERM ; ^



SET TERM ^ ;
CREATE TRIGGER TRG_BDBEJEMPLAR FOR BEJEMPLAR
BEFORE DELETE POSITION 0
AS
    DECLARE VARIABLE VUE TYPE OF COLUMN ALIBRO_AUX.UE;
BEGIN
    /* enter trigger code here */
    SELECT UE FROM BLIBRO WHERE ISBN = OLD.ISBN
    INTO :VUE;
    IF ( VUE <> OLD.NRO ) THEN EXCEPTION EX_EJEMPLAR;
END^
SET TERM ; ^


SET TERM ^ ;
CREATE TRIGGER TRG_ADBEJEMPLAR FOR BEJEMPLAR
AFTER DELETE POSITION 0
AS
BEGIN
    /* enter trigger code here */
    UPDATE BLIBRO SET UE = UE - 1 WHERE ISBN = OLD.ISBN;
END^
SET TERM ; ^
```

6.5 Implementación de log de actividad sobre una tabla

Ejemplo: dadas las siguientes tablas:

CLIBRO(**ISBN**,TITULO)

cada vez que se se hacen altas, bajas, cambios sobre CLIBRO se guarda en:

LOG(**ID**,FECHA,USUARIO,TIPOCAMBIO,REGISTRO)

el cambio realizado, teniendo en cuenta:
ID es un numero unívoco generado por el generator GEN_LOGID
FECHA es la fecha hora actual (usar 'TODAY')

USUARIO es el usuario actual (usar USER)
TIPOCAMBIO puede tener los valores:
"A" alta y REGISTRO contiene el nuevo registro
"B" baja y REGISTRO contiene el viejo registro borrado
"UV" update y REGISTRO contiene el registro anterior, viejo
"UN" update y REGISTRO contiene el registro actual, nuevo
REGISTRO contiene el nuevo o viejo registro con formato "ISBN=valor&TITULO=valor"

Análisis:

Tanto en la opción a) como en la opción b) de implementación de la b.d., la tabla LOG no estaría afectada y el usuario no podría impedir la ejecución de los triggers. No obstante, la tabla LOG requiere ser protegida con los permisos adecuados, para que el registro de eventos no se pueda alterar.

EVENTO		**CLIBRO**	**LOG**
INSERT	BEFORE		ID=GEN_ID(GEN_LOGID,1) FECHA='TODAY' USUARIO=USER
	AFTER	Insertar tupla en LOG, tipocambio='A'	
UPDATE	BEFORE	Insertar tupla en LOG, tipocambio='UV'	
	AFTER	Insertar tupla en LOG, tipocambio='UN'	
DELETE	BEFORE		
	AFTER	Insertar tupla en LOG, tipocambio='B'	

Implementación:

```
CREATE GENERATOR GEN_LOGID;
SET GENERATOR GEN_LOGID TO 0;


CREATE TABLE CLIBRO
(
  ISBN char(20) NOT NULL primary key,
  TITULO varchar(100) NOT NULL
);


CREATE TABLE LOG
(
    ID INTEGER NOT NULL PRIMARY KEY,
    FECHA DATE DEFAULT 'TODAY' NOT NULL ,
    USUARIO CHAR(32) DEFAULT USER NOT NULL ,
    TIPOCAMBIO CHAR(2) NOT NULL,
    REGISTRO VARCHAR(200) NOT NULL,
    CONSTRAINT CK_LOG_TC CHECK (TIPOCAMBIO IN('A','B','UV','UN'))
);


SET TERM ^;
```

```
CREATE TRIGGER TRG_BILOG FOR LOG
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
    /* enter trigger code here */
    NEW.ID = GEN_ID(GEN_LOGID,1);
    NEW.FECHA = 'TODAY';
    NEW.USUARIO = USER;
END^

SET TERM ;^


SET TERM ^ ;
CREATE TRIGGER TRG_AICLIBRO FOR CLIBRO
AFTER INSERT POSITION 0
AS
BEGIN
    /* enter trigger code here */
    INSERT INTO LOG(TIPOCAMBIO,REGISTRO)
    VALUES('A','ISBN='||NEW.ISBN||'&TITULO='||NEW.TITULO);
END^
SET TERM ; ^



SET TERM ^ ;
CREATE TRIGGER TRG_BUCLIBRO FOR CLIBRO
BEFORE UPDATE POSITION 0
AS
BEGIN
    /* enter trigger code here */
    INSERT INTO LOG(TIPOCAMBIO,REGISTRO)
    VALUES('UV','ISBN='||OLD.ISBN||'&TITULO='||OLD.TITULO);
END^
SET TERM ; ^


SET TERM ^ ;
CREATE TRIGGER TRG_AUCLIBRO FOR CLIBRO
AFTER UPDATE POSITION 0
AS
BEGIN
    /* enter trigger code here */
    INSERT INTO LOG(TIPOCAMBIO,REGISTRO)
    VALUES('UN','ISBN='||NEW.ISBN||'&TITULO='||NEW.TITULO);
END^
SET TERM ; ^


SET TERM ^ ;
CREATE TRIGGER TRG_ADCLIBRO FOR CLIBRO
AFTER DELETE POSITION 0
AS
BEGIN
```

```
    /* enter trigger code here */
    INSERT INTO LOG(TIPOCAMBIO,REGISTRO)
    VALUES('B','ISBN='||OLD.ISBN||'&TITULO='||OLD.TITULO);
END^
SET TERM ; ^
```

Atte. Guillermo Cherencio
UNLu BD II 11078