

TEMA 5. PROGRAMACIÓN DE BASES DE DATOS

(Parte II - Desencadenadores o Triggers)

1. PROCEDIMIENTOS ALMACENADOS EN Microsoft SQL Server
2. DESENCADENADORES O TRIGGERS EN Microsoft SQL Server

2. DESENCADENADORES (LMD) O TRIGGERS (DML) EN Microsoft SQL Server

INTRODUCCIÓN

Los desencadenadores son una **clase especial de procedimiento almacenado** que se define para que **se ejecute automáticamente** cuando se **emita una instrucción UPDATE, INSERT o DELETE** contra una **tabla**. Los desencadenadores son una herramienta eficaz que permite que cada sitio exija automáticamente las reglas de la compañía cuando se modifican los datos. **Los desencadenadores amplían la lógica de comprobación de integridad de las restricciones, valores predeterminados y reglas**, aunque **se deben utilizar las restricciones y los valores predeterminados siempre que éstos aporten toda la funcionalidad necesaria**.

Las tablas pueden tener varios desencadenadores. La instrucción CREATE TRIGGER **se puede definir** con las cláusulas **FOR UPDATE, FOR INSERT o FOR DELETE**, para asignar un desencadenador a una acción específica de modificación de datos.

Los desencadenadores pueden automatizar los procesos de las organizaciones. En un sistema de inventario, los desencadenadores de actualización pueden detectar cuándo se alcanza el nivel mínimo y generar automáticamente un pedido al proveedor. En una base de datos que registre los procesos de una fábrica, los desencadenadores pueden enviar mensajes a los operadores, por correo electrónico o mediante servicios de localización, cuando un proceso sobrepase los límites de seguridad definidos.

Los desencadenadores **contienen instrucciones Transact-SQL**, como los procedimientos almacenados. Tanto los desencadenadores como los procedimientos almacenados, devuelven el conjunto de resultados generado por las instrucciones SELECT. Por tanto, no se recomienda que se incluya en los desencadenadores instrucciones **SELECT**, excepto las necesarias para asignar valores a los parámetros, ya que los usuarios no esperan ningún conjunto de resultados en las instrucciones UPDATE, INSERT o DELETE.

Por defecto, los desencadenadores se ejecutan después de la terminación de la instrucción que los desencadena. Si la instrucción termina con un error, como la violación de una restricción o un error de sintaxis, el desencadenador no se ejecuta.

CONCEPTO

Es un tipo especial de procedimiento almacenado que en lugar de ejecutarse como respuesta a una llamada explícita al mismo, **se pone en funcionamiento automáticamente como respuesta a ciertas modificaciones de los datos en una tabla.**

UTILIDAD

Descarga a la aplicación de revisar las acciones de actualización de tablas, puesto que permite examinar el estado de una tabla después de una modificación y obrar en consecuencia.

Se puede utilizar por tanto, para controlar:

- **Reglas de Integridad Semántica (R.I Específicas para una base de datos en concreto).** Por ejemplo, que **no se pueda insertar un pedido si el cliente tiene cuentas por pagar**. Cuando se inserte el pedido se disparará un trigger que comprobará si este cliente tiene cuentas pendientes, en cuyo caso se deshace lo insertado.
- **Reglas de Integridad Referencial** (en el caso de no estar implementadas).

TIPOS

Se dividen en dos tipos principales:

1. **Desencadenadores o Triggers DML** (Data Manipulation Language)
2. **Desencadenadores o Triggers DDL** (Data Definition Language)

Principales diferencias:

Característica	Desencadenador DML	Desencadenador DDL
Se activa con...	INSERT , UPDATE , DELETE	CREATE , ALTER , DROP
Se asocia a...	Tablas o vistas	Base de datos
Puede acceder a...	inserted , deleted	EVENTDATA()
Se usa para...	Registrar cambios en datos, aplicar reglas de negocio	Prevenir modificaciones en la estructura de la BD

CREACIÓN (TRIGGER DML)

La sentencia Transact-SQL que permite crear desencadenadores es CREATE TRIGGER. Una sintaxis reducida es la siguiente:

```
CREATE TRIGGER Nombre_Trigger ON { Nombre_Tabla | Nombre_Vista }  
{ FOR | AFTER | INSTEAD OF }  
{ [ INSERT ] [, UPDATE ] [, DELETE ] }  
AS  
[BEGIN]  
    instrucciónSQL [...n]  
[END]
```

En **Microsoft SQL Server**, cuando se crean **triggers (desencadenadores)**, se pueden usar diferentes cláusulas para definir **cuándo** y **cómo** se ejecuta el trigger. Las más importantes son:

1. **AFTER** (Equivalente a FOR)

- Se ejecuta **después** de que la operación INSERT, UPDATE o DELETE se haya realizado con éxito.
- No se puede usar en vistas, solo en **tablas**.
- Si la operación genera un error, el trigger **no** se ejecuta.

2. **INSTEAD OF**: En lugar del evento

- Se ejecuta **en lugar** de la operación INSERT, UPDATE o DELETE.
- Se puede usar tanto en **tablas** como en **vistas**.
- Útil para aplicar restricciones personalizadas o redirigir la operación.
- No se permite en DDL. Es decir, no podemos crear un trigger DDL con la cláusula INSTEAD OF.

Por lo tanto, los triggers en Microsoft SQL Server pueden ser:

1. DML (Data Manipulation Language)

- **AFTER**
 - Se ejecutan después de una operación DML (INSERT, UPDATE, DELETE).
 - Solo actúan sobre tablas.
- **INSTEAD OF**
 - Se ejecutan en lugar de la operación DML.
 - Pueden actuar sobre tablas y vistas.

2. DDL (Data Definition Language)

- **AFTER**
 - Se ejecutan después de eventos DDL (CREATE, ALTER, DROP, etc.).
 - Actúan sobre bases de datos.

ANIDAMIENTO

Existe anidamiento cuando un trigger actúa sobre una tabla que a su vez tiene definido un trigger. Los triggers en Microsoft SQL Server pueden anidarse hasta 32 niveles.

EJEMPLO:

```
CREATE TRIGGER t1 ON Tabla1
FOR INSERT
AS
BEGIN
    DELETE FROM Tabla2 ...
END;
```

--Tabla2 tiene definido a su vez otro trigger para cuando se hace un borrado.

```
CREATE TRIGGER t2 ON Tabla2
FOR DELETE
AS
BEGIN
    UPDATE Tabla3 ...
END;
```

SENTENCIAS NO ADMISIBLES

Las instrucciones Transact-SQL de **creación, modificación y borrado de objetos NO** están permitidas en un desencadenador o trigger.

BORRADO

```
DROP TRIGGER Nombre_Trigger
```

TABLAS INSERTED Y DELETED

- ✓ En las **instrucciones de desencadenadores** se utilizan dos tablas especiales: la tabla **deleted** y la tabla **inserted**. Se pueden utilizar estas tablas temporales para probar los efectos de determinadas modificaciones de datos y establecer condiciones para las acciones de los desencadenadores.
- ✓ La tabla **deleted** **almacena copias de las filas afectadas por las instrucciones DELETE y UPDATE**. Durante la ejecución de una instrucción DELETE o UPDATE, **las filas se eliminan de la tabla del desencadenador y se transfieren a la tabla deleted**. La tabla **deleted** y la **tabla del desencadenador** **no suelen tener filas en común**.
- ✓ La tabla **inserted** **almacena copias de las filas afectadas por las instrucciones INSERT y UPDATE**. Durante una transacción INSERT o UPDATE, se agregan nuevas filas a la tabla **inserted** y a la tabla del desencadenador. Las filas de la tabla **inserted** son copias de las nuevas filas de la tabla del desencadenador.
- ✓ Una **transacción UPDATE es como una eliminación seguida de una inserción**: primero, se copian las filas antiguas en la tabla **deleted**, y a continuación, se copian las filas nuevas en la tabla del desencadenador y en la tabla **inserted**.
- ✓ Cuando se establecen condiciones para el desencadenador, se utilizan las tablas **inserted** y **deleted** según la acción que activará el desencadenador. Aunque no se produce ningún error si se hace referencia a **deleted** cuando se comprueba INSERT, o a **inserted** cuando se comprueba DELETE, estas tablas de prueba del desencadenador no contendrán filas en estos casos.

<p>CREATE TRIGGER trig1 ON TABLA1 FOR DELETE AS</p> <p>SENTENCIAS SQL</p>	<p>Se insertan las filas que van a ser borradas en tabla DELETED</p> <p>Se borran las filas en TABLA1</p> <p>Se Ejecuta el TRIGGER</p>
<p>CREATE TRIGGER trig2 ON TABLA1 FOR INSERT AS</p> <p>SENTENCIAS SQL</p>	<p>Se insertan las filas en tabla INSERTED</p> <p>Se insertan las filas en TABLA1</p> <p>Se Ejecuta el TRIGGER</p>
<p>CREATE TRIGGER trig3 ON TABLA1 FOR UPDATE AS</p> <p>SENTENCIAS SQL</p>	<p>Se insertan las filas afectadas antes de la modificación de la tabla TABLA1 en DELETED.</p> <p>Se insertan las filas afectadas después de la modificación de la tabla TABLA1 en INSERTED.</p> <p>Se modifican las filas en tabla TABLA1</p> <p>Se Ejecuta el TRIGGER</p>

TRIGGERS Y TRANSACCIONES

Los triggers pueden ser ejecutados en respuesta a una acción que está dentro de una transacción. Entonces si uno de esos triggers contiene una sentencia ROLLBACK TRANSACTION, la transacción completa en la que dicho trigger se ejecuta será descartada.

EJEMPLO:

```
CREATE PROCEDURE PROC AS
  BEGIN TRANSACTION
    DELETE TEMPLE WHERE COMIS >500
    .....
    UPDATE TEMPLE SET COMIS = COMIS + 100
  COMMIT

CREATE TRIGGER TRIG ON TEMPLE
FOR UPDATE AS
  IF (SELECT SUM(COMIS) FROM TEMPE) >2000
    ROLLBACK TRANSACTION
```


Para los siguientes ejemplos ejecuta el script “FORMACION.sql” que contiene la base de datos formacion. Observa que las tablas no tienen relaciones. El archivo pertenece a EJEMPLOS_TRIGGERS.

```
/*-Supongamos que alguien con pocos conocimientos de BBDD Relacionales
  ha hecho esta base de datos. Ha creado todas las claves primarias,
  pero no ha creado las claves foráneas y por tanto, no hay relaciones
  entre las tablas.
-Supongamos también que no tenemos permiso para modificar la
  estructura de las tablas. Es decir, no tenemos permiso para ejecutar
  ALTER TABLE.
*/
CREATE DATABASE formacion;
GO
USE formacion;

CREATE TABLE departamento
(CodDepto INT NOT NULL PRIMARY KEY,
 NomDepto VARCHAR(50) NOT NULL
);

CREATE TABLE trabajador
(CodTrab INT NOT NULL PRIMARY KEY,
 NomTrab VARCHAR(50) NOT NULL,
 ApeTrab VARCHAR(50) NOT NULL,
 FechaNac DATETIME NULL,
 Salario MONEY NULL,
 Comis MONEY NULL,
 CodDepto INT NULL
);

CREATE TABLE curso
(CodCurso INT NOT NULL PRIMARY KEY,
 NomCurso VARCHAR(50) NOT NULL,
 Horas INT NULL,
 Fecha DATETIME NULL
);
```

Tema 5. Programación de Bases de Datos

```
CREATE TABLE cursado
(CodCursado INT NOT NULL PRIMARY KEY,
CodCurso INT NOT NULL,
CodTrab INT NOT NULL,
Apto CHAR(1) NULL
);
```

Veamos un ejemplo de **trigger que se desencadena como resultado de la ejecución de una sentencia INSERT sobre una tabla**. El ejemplo se encuentra en “Ej1 trigger sobre tabla para INSERT.sql”. El archivo pertenece a: EJEMPLOS_TRIGGERS.

```
/*Queremos controlar que el nombre de un curso no esté duplicado en la
tabla curso. La tabla curso no tiene la restricción UNIQUE en el
campo NomCurso. Además, no tenemos permiso para modificar la
estructura de las tablas para poder añadirle esta restricción.*/

/*Trigger que controla que no se den de alta dos cursos con el mismo
nombre.*/

USE formacion;

SELECT * FROM curso;

CREATE OR ALTER TRIGGER trig1
ON curso
FOR INSERT
AS
BEGIN

DECLARE @CODCURSO INT

SELECT @CODCURSO=CodCurso FROM inserted

/*PRINT @CODCURSO
SELECT * FROM inserted*/
```

```
IF (SELECT COUNT(*)
    FROM curso c, inserted i
    WHERE c.NomCurso=i.NomCurso)>1
BEGIN
    DELETE curso WHERE CodCurso=@CODCURSO
    PRINT 'No se puede insertar el curso porque el nombre ya
        existe'
    END
END;

--Comprobaciones:
INSERT INTO curso
VALUES (100, 'Curso1', 30, '20/04/2024');

INSERT INTO curso
VALUES (101, 'Curso2', 30, '20/04/2024');

INSERT INTO curso
VALUES (102, 'Curso1', 30, '20/04/2024');

SELECT * FROM curso;

--Vamos a hacer el mismo trigger, pero de otra manera.

--Borrar un Trigger:
DROP TRIGGER trig1;

CREATE OR ALTER TRIGGER trig1
ON curso
FOR INSERT
AS
BEGIN

DECLARE @CODCURSO INT, @NOMCURSO VARCHAR(50)

SELECT @CODCURSO=CodCurso, @NOMCURSO=NomCurso FROM inserted
```

```
IF (SELECT COUNT(*)
    FROM curso
    WHERE NomCurso=@NOMCURSO)>1
BEGIN
    DELETE curso WHERE CodCurso=@CODCURSO
    PRINT 'No se puede insertar el curso porque el nombre ya
        existe'
END
END;

--Vaciamos la tabla:
SELECT * FROM curso;

DELETE FROM curso;

--Comprobaciones:
INSERT INTO curso
VALUES (100, 'Curso1', 30, '20/04/2024');

INSERT INTO curso
VALUES (101, 'Curso2', 30, '20/04/2024');

INSERT INTO curso
VALUES (102, 'Curso1', 30, '20/04/2024');

SELECT * FROM curso;
```

Veamos un ejemplo de **trigger que se desencadena como resultado de la ejecución de una sentencia DELETE sobre una tabla**. El ejemplo se encuentra en “Ej2 trigger sobre tabla para DELETE.sql”. El archivo pertenece a: EJEMPLOS_TRIGGERS.

```
/*Queremos controlar la Regla de Integridad Referencial para cuando borramos un departamento, de manera que se ponga a NULL el campo CodDepto en la tabla trabajador para todos los trabajadores que pertenecían al departamento que acabamos de borrar. Además, no tenemos permiso para modificar la estructura de las tablas para poder añadirle esta restricción.*/
```

```
/*Trigger que controla que cuando borremos un departamento, para todos los empleados pertenecientes al mismo, tengan NULL en su campo CodDepto hasta que se les asigne a otro departamento*/
```

```
USE formacion;
```

```
CREATE TRIGGER trig2  
ON departamento  
FOR DELETE  
AS  
BEGIN
```

```
DECLARE @CODDEPTO INT
```

```
SELECT @CODDEPTO=CodDepto FROM deleted
```

```
IF (SELECT COUNT(*)  
FROM trabajador  
WHERE CodDepto=@CODDEPTO) > 0  
BEGIN  
    UPDATE trabajador  
    SET CodDepto=NULL  
    WHERE CodDepto = @CODDEPTO  
END  
END;
```

```
--Comprobaciones:
```

```
INSERT INTO departamento  
VALUES(1, 'DEPARTAMENTO1'), (2, 'DEPARTAMENTO2'), (3, 'DEPARTAMENTO3');
```

```
SELECT * FROM departamento;
```

Tema 5. Programación de Bases de Datos

```
INSERT INTO trabajador (CodTrab,NomTrab,ApeTrab,Salario,CodDepto)
VALUES (1,'JOSÉ','PÉREZ PÉREZ',1500,1),
       (2,'JUAN','PÉREZ LÓPEZ',1500,1),
       (3,'ANA','GÓMEZ LÓPEZ',1500,2);

SELECT * FROM trabajador;

DELETE FROM departamento
WHERE CodDepto=1;
```

Veamos un ejemplo de **trigger** que se desencadena como resultado de la ejecución de una sentencia **UPDATE** sobre una tabla. El ejemplo se encuentra en “Ej3 trigger sobre tabla para UPDATE.sql”. El archivo pertenece a: EJEMPLOS_TRIGGERS.

```
/*Queremos controlar la Regla de Integridad Referencial para cuando
modificamos un código de departamento en la tabla departamento, de manera
que se modifique en CASCADA en el campo CodDepto en la tabla trabajador.
Además, no tenemos permiso para modificar la estructura de las tablas para
poder añadirle esta restricción.*/

/*Trigger que controla que cuando modifiquemos el código de departamento en
la tabla departamento, se modifique en CASCADA en la tabla de trabajador.*/
USE formacion;

CREATE TRIGGER trig3
ON departamento
FOR UPDATE
AS
BEGIN

DECLARE @VANTIGUO INT
DECLARE @VNUEVO INT

SELECT @VANTIGUO=CodDepto FROM deleted
--PRINT @VANTIGUO

SELECT @VNUEVO=CodDepto FROM inserted
--PRINT @VNUEVO

IF (SELECT COUNT(*) FROM trabajador WHERE CodDepto = @VANTIGUO) > 0
    BEGIN
        UPDATE trabajador SET CodDepto = @VNUEVO WHERE CodDepto = @VANTIGUO
    END
END;
```

```
--Comprobaciones:
SELECT * FROM departamento;
SELECT * FROM trabajador;

UPDATE departamento
SET CodDepto=200
WHERE CodDepto=2;
```

Veamos un ejemplo de **trigger** que se desencadena como resultado de la ejecución de una sentencia **INSERT** y **UPDATE** sobre una tabla. El ejemplo se encuentra en “Ej4 trigger sobre tabla para INSERT y UPDATE.sql”. El archivo pertenece a: EJEMPLOS_TRIGGERS.

```
/*Queremos controlar la Regla de Integridad Referencial, de manera que no
pueda existir un trabajador en un departamento que no exista. El trabajador
puede tener el campo CodDep a NULL, pero si tiene un valor en dicho campo,
este valor debe existir en la tabla departamento. De esta manera, no
existirá un valor de clave foránea sin concordancia. Además, no tenemos
permiso para modificar la estructura de las tablas para poder añadirle
esta restricción.*/

/*Trigger que controla que cuando se introduzca un código de departamento
en la tabla trabajador (bien por añadir una fila, o bien por modificar el
valor de CodDepto), este código exista en la tabla departamento o bien
este sea nulo.*/

USE formacion;

--Deshaciendo a transacción
CREATE OR ALTER TRIGGER trig4
ON trabajador
FOR INSERT, UPDATE
AS
BEGIN

DECLARE @CODDEPTO INT

SELECT @CODDEPTO=CodDepto FROM inserted
```

Tema 5. Programación de Bases de Datos

```
IF @CODDEPTO IS NOT NULL
    BEGIN
        IF (SELECT COUNT(*)
            FROM departamento
            WHERE CodDepto = @CODDEPTO) = 0
            BEGIN
                ROLLBACK TRANSACTION
                PRINT 'El código que intentó introducir no se
                    corresponde con ningún departamento'
            END
        END
    END;

--Comprobaciones con UPDATE:
SELECT * FROM departamento;
SELECT * FROM trabajador;

--Departamento que existe.
UPDATE trabajador
SET CodDepto=3
WHERE CodTrab=2;

--Quitamos el departamento al trabajador 2.
UPDATE trabajador
SET CodDepto=NULL
WHERE CodTrab=2;

--Departamento que no existe.
UPDATE trabajador
SET CodDepto=1000
WHERE CodTrab=3;

--Comprobaciones con INSERT:
SELECT * FROM departamento;
SELECT * FROM trabajador;

--Departamento que existe.
INSERT INTO trabajador
VALUES (4, 'ROCIO', 'GÁLVEZ ROMERO', '12/09/1990', 2000, 100, 3);

--Trabajador sin departamento.
INSERT INTO trabajador
VALUES (5, 'ROBERTO', 'GONZÁLEZ MORALES', '19/10/1999', 2000, 100, NULL);
```



```
--Departamento que no existe.
INSERT INTO trabajador
VALUES (6, 'DAVID', 'LÓPEZ ROMERO', '18/03/1998', 2000, 100, 1000);

/* Vamos a hacer el mismo trigger, pero sin deshacer la transacción.
-Si el valor incorrecto en la tabla trabajador lo provocó un UPDATE,
hacemos de nuevo un UPDATE para dejar la tabla como estaba antes de
ejecutar la instrucción.
-Si el valor incorrecto en la tabla trabajador lo provocó un INSERT,
hacemos un DELETE para dejar la tabla como estaba antes de
ejecutar la instrucción.
*/

--Borramos el trigger:
DROP TRIGGER trig4;

CREATE TRIGGER trig4
ON trabajador
FOR INSERT, UPDATE
AS
BEGIN

DECLARE @CODDEPTO_NUEVO INT , @CODDEPTO_ANTIGUO INT, @CODTRAB INT, @FILAS INT

SELECT @CODDEPTO_NUEVO=CodDepto, @CODTRAB=CodTrab FROM inserted

SELECT @CODDEPTO_ANTIGUO=CodDepto FROM deleted

IF @CODDEPTO_NUEVO IS NOT NULL
BEGIN
    IF (SELECT COUNT(*) FROM departamento WHERE CodDepto = @CODDEPTO_NUEVO) = 0
    BEGIN
        PRINT 'El código que intentó introducir no se corresponde
        ningún departamento'
        SELECT @FILAS = COUNT(*) FROM deleted
        --PRINT @FILAS
        IF (@FILAS)=0 --Ha sido un INSERT
        BEGIN
            DELETE FROM trabajador WHERE CodTrab=@CODTRAB
        END
    END
END
```

Tema 5. Programación de Bases de Datos

```
ELSE --Ha sido un UPDATE
BEGIN
    --PRINT @CODDEPTO_ANTIGUO
    UPDATE trabajador
    SET CodDepto=@CODDEPTO_ANTIGUO
    WHERE CodTrab=@CODTRAB
END
END
END;

--Comprobaciones con UPDATE:
SELECT * FROM departamento;
SELECT * FROM trabajador;

--Departamento que existe.
UPDATE trabajador
SET CodDepto=3
WHERE CodTrab=2;

--Quitamos el departamento al trabajador 2.
UPDATE trabajador
SET CodDepto=NULL
WHERE CodTrab=2;

--Departamento que no existe.
UPDATE trabajador
SET CodDepto=1000
WHERE CodTrab=3;

--Comprobaciones con INSERT:
SELECT * FROM departamento;
SELECT * FROM trabajador;

--Departamento que existe.
INSERT INTO trabajador
VALUES (6, 'ALBA', 'GÁLVEZ ROMERO', '12/09/1990', 2000, 100, 3);

--Trabajador sin departamento.
INSERT INTO trabajador
VALUES (7, 'DARIO', 'GONZÁLEZ MORALES', '19/10/1999', 2000, 100, NULL);

--Departamento que no existe.
INSERT INTO trabajador
VALUES (8, 'ROSA', 'LÓPEZ ROMERO', '18/03/1998', 2000, 100, 1000);
```

Veamos un ejemplo de **trigger** que se desencadena como resultado de la ejecución de la sentencia (en este caso **UPDATE**). Cuando se cumple una condición en el cuerpo del trigger, se ejecuta un **ROLLBACK TRANSACTION** y puesto que, las sentencias **UPDATE** están dentro de una transacción, esto provocará que se deshaga todas las sentencias que estaban dentro de ella. El ejemplo se encuentra en “Ej5 procedimiento almacenado-transaccion-trigger.sql”. El archivo pertenece a: **EJEMPLOS_TRIGGERS**.

```
/*Vamos a comprobar que Los triggers pueden ser ejecutados en respuesta a una acción que está dentro de una transacción. Entonces, si uno de esos triggers contiene una sentencia ROLLBACK TRANSACTION, la transacción completa en la que dicho trigger se ejecuta será descartada.*/
```

```
/*Queremos crear un procedimiento almacenado para ir bajando los salarios de 50 en 50 euros e ir subiendo las comisiones de 100 en 100 euros. Podemos hacer esta operación mientras la suma de las comisiones no supere 2000 euros.*/
```

```
/*Vamos a definir un trigger sobre la tabla trabajador para cuando se realice UPDATE de modo, que permita la bajada de los salarios y la subida de las comisiones mientras no se superen los 2000 euros. Cuando esto ocurra, se realiza un ROLLBACK que deshará la transacción donde estaban los UPDATE para bajar salarios y subir comisiones*/
```

```
USE formacion;
```

```
CREATE OR ALTER PROCEDURE proc1 AS  
BEGIN
```

```
BEGIN TRANSACTION
```

```
--Bajamos los salarios
```

```
UPDATE trabajador
```

```
SET salario = salario - 50
```

```
--Subimos las comisiones
```

```
UPDATE trabajador
```

```
SET comis = ISNULL(comis,0) + 100
```

```
COMMIT
```

```
END;
```

```
CREATE TRIGGER trig5 ON trabajador
```

```
FOR UPDATE AS
BEGIN
/*Si nos salimos del presupuesto asignado para comisiones lo dejamos
todo como estaba*/

IF (SELECT SUM(comis)
      FROM trabajador) > 2000
    BEGIN
      ROLLBACK TRANSACTION
    END
END;

SELECT * FROM trabajador;
SELECT SUM(comis) from trabajador;

EXEC proc1;
```

Veamos un ejemplo de **trigger INSTEAD OF** El ejemplo se encuentra en “Ejemplo Trigger INSTEAD OF.sql”. El archivo pertenece a: EJEMPLOS_TRIGGERS.

```
/*En Microsoft SQL Server, los triggers se pueden definir para que se
ejecuten después de la instrucciones INSERT, DELETE o UPDATE (estos
son los triggers AFTER, que es como funcionan por defecto). Pero,
también se pueden definir para que actúen en lugar de estas
instrucciones (se conocen como triggers INSTEAD OF). Sin embargo, SQL
Server no soporta directamente los triggers que se ejecutan antes de
estos eventos, como sí lo hacen otros sistemas de gestión de bases de
datos como PostgreSQL con sus triggers BEFORE.*/

/*Un trigger AFTER se ejecuta justo después de que se ha completado la
acción que lo activa, permitiendo revisar o cambiar los datos
afectados por la acción original. Por otro lado, un trigger INSTEAD OF
se ejecuta en lugar de la acción que lo activa, permitiendo
personalizar completamente el comportamiento de la acción, lo cual
puede ser útil para validar o modificar los datos antes de que se
realice la operación original.*/
```

```
/*Si se necesita una funcionalidad similar a un trigger BEFORE,
en SQL Server podrías utilizar un trigger INSTEAD OF para realizar
validaciones o modificaciones antes de proceder con la inserción,
eliminación o actualización manualmente dentro del cuerpo del trigger.
Esto implica que, dentro del trigger INSTEAD OF, deberás escribir
explícitamente la sentencia para insertar, actualizar o eliminar los
datos, ya que el trigger reemplaza la operación original.*/
```

```
/*Queremos evitar la inserción de un email si ya existe, es decir,
evitar que se repitan como hicimos con los nombres de los cursos en el
primer ejemplo.*/
```

```
CREATE DATABASE usuarios;
GO
USE usuarios;
```

```
CREATE TABLE usuario
( UsuarioID INT PRIMARY KEY,
  Nombre VARCHAR(50),
  Apellido VARCHAR(50),
  Email VARCHAR(100)
);
```

```
CREATE OR ALTER TRIGGER trgEmpleadoInsert
ON usuario
INSTEAD OF INSERT
AS
BEGIN
  -- Evitar la inserción si el email ya existe
  IF
    NOT EXISTS (SELECT 1 FROM usuario u
                JOIN inserted i ON u.Email = i.Email)
  BEGIN
    -- Insertar el registro si el email no existe
    INSERT INTO usuario (UsuarioID, Nombre, Apellido, Email)
    SELECT UsuarioID, Nombre, Apellido, Email
    FROM inserted;
  END
```

Tema 5. Programación de Bases de Datos

```
ELSE
  BEGIN
    /*Podemos lanzar un error o registrar el intento de inserción
    fallido*/
    RAISERROR ('El email ya existe en la base de datos.', 16, 1);
  END
END;

--Comprobaciones:
INSERT INTO usuario (UsuarioID, Nombre, Apellido, Email)
VALUES (1, 'Juan', 'Pérez', 'juan.perez@example.com'),
       (2, 'Ana', 'García', 'ana.garcia@example.com');

SELECT * FROM usuario;

--Introducimos de nuevo el mismo email
INSERT INTO usuario (UsuarioID, Nombre, Apellido, Email)
VALUES (3, 'Antonio', 'Jiménez', 'juan.perez@example.com');

INSERT INTO usuario (UsuarioID, Nombre, Apellido, Email)
VALUES (3, 'Antonio', 'Jiménez', 'antonio.jimenez@example.com');

SELECT * FROM usuario;
```

Veamos dos ejemplos de **trigger DDL**. En Microsoft SQL Server Management Studio, podemos ver los triggers DDL en el apartado Programmability / Database Triggers. Los ejemplos se encuentran en “Ejemplos Triggers DDL.sql”. El archivo pertenece a: EJEMPLOS_TRIGGERS.

```
/*Supongamos que queremos evitar que se eliminen tablas en la
base de datos:*/
```

```
USE usuarios;
```

```
CREATE TRIGGER trg_PreventDropTable
ON DATABASE
FOR DROP_TABLE
AS
BEGIN
```

```
    PRINT '¡No está permitido eliminar tablas en esta base de
datos!';
```

```
    ROLLBACK TRANSACTION;
```

```
END;
```

```
--Comprobación:
```

```
DROP TABLE usuario;
```

```
SELECT * FROM usuario;
```

```
/*Supongamos que queremos registrar en una tabla todas las
creaciones, modificaciones de estructura o borrados de tablas.*/
```

```
CREATE TABLE AuditoriaDDL
( ID INT IDENTITY(1,1) PRIMARY KEY,
  Evento XML,
  Fecha DATETIME DEFAULT GETDATE()
);
```

```
CREATE TRIGGER trg_AuditDDL
ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
    INSERT INTO AuditoriaDDL (Evento)
    VALUES (EVENTDATA());
END;

--Comprobación:
ALTER TABLE usuario
ADD Edad DATE NULL;

SELECT * FROM AuditoriaDDL;
```