

## *Unidad 04*

# Transac en Sql Server

## Parte II

### Objetivo:

Conocer y aplicar desencadenadores, transacciones y funciones.

### Contenido:

- Desencadenadores: Concepto, aplicaciones.
- Transacciones: Concepto, aplicaciones.
- Funciones: Funciones de sql server, creación de funciones

## DESENCADENADORES O TRIGGERS

Un desencadenador es una clase especial de procedimiento almacenado que se ejecuta automáticamente cuando se produce un evento en una base de datos que ha ejecutado una instrucción DML, DDL o LOGON.

Los desencadenadores DML se ejecutan con un INSERT, UPDATE o DELETE, mientras que los desencadenadores DDL se ejecutan con una instrucción CREATE, ALTER y DROP de Transact-SQL, por último los desencadenadores LOGON se activan cuando se establece la sesión de un usuario.

Los desencadenadores se pueden crear directamente a partir de instrucciones Transact-SQL o de métodos de ensamblados que se crean en Microsoft.NET Framework Common Language Runtime (CLR) y se cargan en una instancia de SQL Server. SQL Server permite crear varios desencadenadores para una instrucción específica.

Los disparadores también permiten realizar cambios “en cascada” en tablas relacionadas, imponer restricciones de columna más complejas que las permitidas por las reglas, compara los resultados de las modificaciones de datos y llevar a cabo una acción resultante.

### Funcionamiento de los Desencadenadores

Cuando creamos un desencadenador del tipo DML se crea una copia de la tabla que se afectara, si la acción es para ejecutar algo al insertar un registro la tabla creada será un **INSERTED**, si es para eliminar, será un **DELETED**, estas tienen la misma estructura del cual se originó el trigger, de tal manera que se pueda verificar los datos y ante un error podrían revertirse los cambios.

### Estructura de un trigger:

```
Create trigger Nombre_Trigger
On Nombre_Tabla
For Insert, Update o Delete (Puede usarse para un)
As
```

Instrucciones Sql

Después de la palabra On se indica la tabla para la cual se está creando el desencadenador.

Después de FOR se debe escribir la instrucción de la acción que puede ser Insert, UpDate, Delete.

Después de AS se escriben las instrucciones SQL del desencadenador.

### Ejemplo:

1. Crear un trigger que desencadene la muestra de los registros de una tabla cada vez que se realice un delete en ella.

```
create trigger tr_ejemplo1
on Cliente
for delete
as
    select*from Cliente
```

go

Para comprobar el accionar del Trigger ejecutemos la siguiente instruccion:

```
delete Cliente
where idcliente=4
go
```

Observaras que se mostró la tabla cliente, después de realizarse una eliminación de un registro.

2. Crear un desencadenador para que al insertar un registro en la tabla Alumno en la base de datos EduTec, salga un mensaje: "Se insertó con éxito"

SOLUCION:

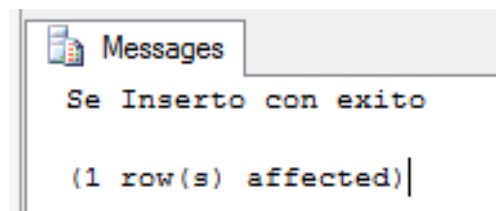
Primero creamos el desencadenador.

```
CREATE TRIGGER Ejemplo2_tr
ON Alumno
FOR INSERT
AS
PRINT 'Se Inserto con exito'
GO
```

Segundo, insertamos una fila.

```
INSERT INTO Alumno
VALUES('A0031','Lopez Guerrero','Luis','','')
```

El resultado que obtendremos será:



3. Crear un desencadenador con las siguientes características:

- ✓ El trigger se afecta a la tabla ORDEN\_DETALLE de la base de datos MerKetPERU.
- ✓ El nombre del Trigger será tr\_ActualizarStock.
- ✓ El objetivo del Trigger es actualizar el Stock de la tabla Productos, cuando se inserta una fila a la tabla ORDEN\_DETALLE.

SOLUCION:

Creación del Trigger:

```
CREATE TRIGGER tr_ActualizarStock
ON ORDEN_DETALLE
FOR INSERT
AS
DECLARE @IdProducto INT
DECLARE @CantidadRecibida SMALLINT
```

```
SELECT @IDProducto = I.IdProducto,  
        @CantidadRecibida = I.CantidadRecibida  
FROM INSERTED AS I INNER JOIN Orden_Detalle AS O  
ON O.IdOrden = I.IdOrden  
UPDATE Producto  
SET StockActual = StockActual + @CantidadRecibida  
WHERE IdProducto = @IdProducto  
GO
```

Verificacion de funcionamiento del Trigger:

Realizamos una Orden para el Producto cuyo IdProducto=20

```
SELECT*FROM Producto  
WHERE IdProducto = 20  
GO
```

Observaras que el Stock de este producto es 120.

Insertamos una Fila a la tabla ORDEN:

```
SELECT*FROM Orden  
GO  
INSERT INTO ORDEN(IdOrden,FechaOrden,FechaEntrada)  
VALUES(19,2011/06/01,2011/06/06)  
GO
```

Insertamos una Fila a la tabla ORDEN\_DETALLE:

```
SELECT*FROM Orden_Detalle  
GO  
INSERT INTO  
ORDEN_DETALLE(IdOrden,IdProducto,PrecioCompra,CantidadSolicitada,Cantida  
dRecibida,Estado)  
VALUES(19,20,1.20,30,30,'Entregado')  
GO
```

Ahora verificamos si se actualizo la tabla productos:

```
SELECT*FROM Producto  
WHERE IdProducto = 20  
GO
```

Observaras que se acualizo el producto cuyo IdProducto es 20 en 150 unidades.

Limitaciones de los triggers.

- Solo se pueden aplicar a una tabla específica, es decir, un trigger no sirve para dos o más tablas
- El trigger se crea en la base de datos de trabajo pero desde un trigger puedes hacer referencia a otras bases de datos.
- Un Trigger devuelve resultados al programa que lo desencadena de la misma forma que un Stored Procedure aunque no es lo más idóneo, para impedir que una instrucción de asignación devuelva un resultado se puede utilizar la sentencia SET NOCOUNT al principio del Trigger.

### Modificar un trigger

Se utiliza la siguiente sintaxis:

ALTER TRIGGER Nombre\_Trigger

ON Nombre\_Tabla de aplicacion

FOR INSERT, UPDATE, DELETE (En que acción se ejecutara)

AS

PRINT "UD.ACABA DE MODIFICAR VALORES EN LA TABLA USUARIO"

(Acción que realizara el Trigger)

### Borrar un triggers

Se utiliza la siguiente sintaxis:

DROP TRIGGER Nombre\_Trigger

## FUNCIONES

SQL Server pone a disposición una serie de funciones que pueden ayudar a resolver problemas complejos con la menor cantidad de código. Veamos algunas de ellas.

<b>FUNCION</b>	<b>DESCRIPCION</b>
ABS(n)	Calcula el valor absoluto de n
CEILING(N)	Devuelve el entero más pequeño mayor o igual que la expresión numérica especificada.
PI()	Devuelve el valor de PI
RAND()	Devuelve el valor aleatorio entre 0 y 1
ROUND(x,y)	Devuelve x redondeado a la longitud o precisión especificados por y.
SQRT(n)	Calcula la raíz cuadrada de n.
LEFT(Exp,n)	Devuelve los n caracteres de Exp, empezando desde la izquierda.
RIGHT(Exp,n)	Devuelve los n caracteres de Exp, empezando desde la derecha.
LEN(Exp)	Devuelve el número de caracteres de Exp.
REVERSE(Exp)	Invierte el valor de una cadena.
GETDATE()	Devuelve la fecha y la hora del equipo
DATEPART(datepart, date)	Devuelve un entero que representa el parámetro datepart especificado del parámetro date especificado.
DAY(date)	Devuelve un entero que es el número del día de date.
MONTH(date)	Devuelve el mes de date.
YEAR(date)	Devuelve el año de date.
DATEDIFF(datepart, startdate, enddate)	Devuelve la diferencia de fechas u horas entre startdate y enddate.
DATEADD(datepart, number, date)	Permite agregar un interval de tiempo dado por numero a date, el tipo de dato devuelto depende de datepart.
CONVERT(data_tipo, Expresión)	Convierte de un tipo de dato en otro.
ERROR_MESSAGE()	Devuelve el texto del mensaje del error que ha provocado la ejecución del bloque CATCH al usar un TRY...CATCH
ISDATE(Expresión)	Determina si una expresión dada es una fecha u hora valida
ISNULL(Expresión,valor)	Reemplaza por valor cada vez que la expresión es NULL.
ISNUMERIC(Expresión)	Verifica si la expresión es de tipo numérico.

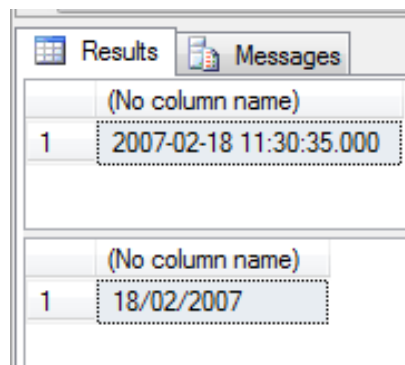
### Ejemplos:

1. Convierta una fecha con el formato usado por el sistema en las bases de dato al formato convencional que usamos normalmente.

### Solución:

El código 103 que usaremos en la conversión indica el formato de fecha que nos solicita el ejercicio.

```
DECLARE @fecha DATETIME
SET @fecha='2007-02-18T11:30:35'
SELECT @fecha
SELECT CONVERT(CHAR,@fecha,103)
GO
```



	(No column name)
1	2007-02-18 11:30:35.000
	(No column name)
1	18/02/2007

- Realiza lo mismo que el caso anterior.

```
Declare @Fecha datetime
Set @Fecha = Getdate()
Select Convert(Char(10), @Fecha,103) As SoloFecha, Convert(Char(8),
@Fecha, 108) As SoloHora
GO
```

- Otra Forma

```
Select CONVERT(Varchar, GETDATE(), 6)
GO
```

- Hallar el numero de clientes registrados en la base de datos Neptuno.

```
USE Neptuno
GO
```

```
DECLARE @num INT
SELECT @num=count(*) FROM Clientes
SELECT 'Número de clientes'=@num
```

**Funciones definidas por el usuario**

Es posible la creación de funciones personalizadas con un fin específico. Para crear una función usamos la siguiente sintaxis:

```
CREATE FUNCTION función_name
```

```
Declaración de variables.
```

```
RETURNS data_type
```

```
BEGIN
```

```
Cuerpo de la function
```

```
END
```

Para llamar a una función podemos hacerlo mediante la palabra reservada EXECUTE o SELECT

**EJC1.6.-**

```
CREATE FUNCTION Sumax
```

```
(@var1 INT,
```

```
@var2 INT)
```

```
RETURNS INT
```

```
AS
```

```
BEGIN
```

```
    RETURN @var1+@var2
```

```
END
```

```
GO
```

Para llamar a la funcion creada usamos:

```
EXEC Sumax 10,50
```

```
SELECT dbo.Sumax(10,50)
```

```
GO
```



## TRANSACCIONES

Durante la ejecución de las operaciones con los datos, el programador debe definir el conjunto de operaciones a ejecutarse como una unidad lógica de proceso. Adicionalmente el motor de base de datos utiliza los bloques como un mecanismo para garantizar las transacciones y mantener la consistencia de la base de datos cuando muchos usuarios acceden simultáneamente a ella.

### Definición de Transacción

Una **transacción** es un conjunto de modificaciones de datos (operaciones) que debe ser procesado como una unidad. Estas deben cumplir cuatro propiedades fundamentales comúnmente conocidas como ACID (Atomicidad, Coherencia, Aislamiento, y Durabilidad).

### Propiedades

**Atomicidad.** - Se entiende así porque una transacción es vista como una unidad de trabajo, es decir siempre se realiza todo el trabajo incluido dentro de la transacción o bien no se hace nada, nunca se queda el trabajo a medias.

**Coherencia.** - Una vez terminada la transacción la base de datos debe quedar en un estado coherente, esto quiere decir que una transacción no debe ir contra las leyes y reglas de la base de datos.

**Aislamiento.** - Este concepto se refiere a que si se están ejecutando dos transacciones al mismo tiempo, estas no se reconocen entre sí, nunca una transacción encontrará una tabla en el medio de las operaciones de otra transacción.

**Durabilidad.** - Este concepto se refiere a que cuando la transacción ha sido confirmada, sus efectos sobre la tabla o tablas son permanentes.

Las Variables con las cuales trabaja de muy cerca una transacción son:

@@TRANCOUNT.- Almacena el número de transacciones activas.

@@ERROR.- Devuelve el número de error de la última transacción ejecutada, si este valor es cero (0) es porque no se efectuó ningún error.

### Ejemplo de Transacción:

Supongamos que tenemos las tablas **Factura**, **Detalle\_Factura** y **Producto** de una base de datos de ventas. La tabla **producto** registra el nivel de inventario (stock) de cada producto. Suponemos también que se tiene que registrar la venta de 10 unidades del producto **X**. Para ello, la aplicación procede de la siguiente manera:

**Operación 1** Registra en la tabla **Factura**, los datos de la cabecera de la factura.

**Operación 2** Registra en la tabla **Detalle\_Factura**, los detalles de la venta.

**Operación 3** Actualiza el nivel de inventario del producto **X** en la tabla **Producto**.

### Pregunta:

¿Qué ocurriría si durante la ejecución de la aplicación se completan las operaciones 1 y 2, y por una falla del sistema, la operación 3 no se lleva a cabo?

**Respuesta:**

Si no se diseña un mecanismo para corregir el error, la base de datos perdería consistencia.

Según las tablas Factura y Detalle\_Factura se han vendido 10 unidades del producto **X**, por lo que en la tabla producto deberíamos tener 10 unidades menos. Esto último no es cierto al no haberse completado la operación 3.

En otras palabras, las operaciones 1, 2 y 3 forman una transacción, y si ésta no se completa, el sistema debe deshacer todas las operaciones.

***Una transacción asegura que las operaciones se llevarán a cabo completas, o en caso contrario, la transacción será anulada para garantizar la consistencia de los datos.***

Las transacciones pueden ser implícitas, de auto confirmación, o explícitas.

**INSTRUCCIONES EN UNA TRANSACCION**

**BEGIN TRANSACTION.-** Es una instrucción que se usa al inicio de toda transacción, para que SQL Server tenga en cuenta que existe una transacción que se está ejecutando.

**COMMIT TRANSACTION.-** Se usa en la finalización de la transacción y confirmar de manera permanente todos los cambios realizados.

**ROLLBACK TRANSACTION.-** Se usa cuando se ha producido un error en la transacción, descartando las modificaciones realizadas a lo largo de la transacción y dejando los datos en el punto de inicio o en un punto de retorno especificado dentro de la transacción.

**SAVE TRANSACTION.-** Esta instrucción nos permite establecer un punto de restauración de la transacción.

**TIPOS DE TRANSACCION****Transacción implícita**

Es aquella en la que cada una de las sentencias INSERT, UPDATE o DELETE se ejecuta como una transacción. Es decir, que si ejecutamos una de estas sentencias, al finalizar la misma, la siguiente sentencia inicia otra transacción.

El modo transacciones implícitas se configura ejecutando la instrucción:

```
SET  
IMPLICIT_TRANSACTIONS ON
```

**Transacción De Autoconfirmación (autocommit)**

Se presenta cuando cada sentencia es confirmada automáticamente cuando se completa. No es necesario especificar ninguna instrucción de control de transacciones.

En este caso, las modificaciones hechas por cada una de las sentencias INSERT, UPDATE o DELETE no podrán ser deshechas o anuladas. Este es el modo predeterminado del motor de base de datos.

**Transacción Explícita**

Es aquella definida explícitamente por el programador utilizando la sentencia BEGIN TRANSACTION.

Consiste en un conjunto de sentencias agrupadas entre las sentencias BEGIN TRANSACTION y COMMIT TRANSACTION.

### **ESTRUCTURA DE UNA TRANSACCION**

#### **Sintaxis:**

```
BEGIN TRANSACTION [ nombre_transacción ]
sentencias_SQL_transaccionales
COMMIT TRANSACTION [ nombre_transacción ]
```

- La sentencia BEGIN TRANSACTION define que cada una de las sentencias SQL que la siguen, forman parte de una transacción.
- Todas las sentencias SQL después de BEGIN TRANSACTION deben ejecutarse para que se considere que la transacción se ha completado, salvo que se decida anular la transacción de manera explícita ejecutando ROLLBACK TRANSACTION.
- La sentencia COMMIT TRANSACTION confirma todas las modificaciones llevadas a cabo por las sentencias SQL anteriores y finaliza la transacción. Cada transacción es registrada en el log de transacciones de la base de datos para mantener la consistencia de la base de datos y ayudar en la recuperación ante la eventualidad de una falla del sistema.

### **CANCELACIÓN DE UNA TRANSACCIÓN**

#### **La sentencia ROLLBACK TRANSACTION**

Como una parte del manejo de errores, puede incluir dentro de la transacción, sentencias de control, de manera tal que al producirse un error la transacción pueda ser deshecha para ello se utiliza la sentencia ROLLBACK TRANSACCIÓN.

#### **Sintaxis:**

```
ROLLBACK TRANSACTION [nombre_transacción]
```

#### **Ejemplos.**

1.- Como primer ejemplo veamos cómo funciona el ROLLBACK, para ello crearemos tablas y con ellas iremos creando transacciones a su vez iremos contando con la función @@TRANCOUN, y luego anularemos las acciones de la transacción. Por ultimo contaremos una vez más y esta sera cero porque se anularon con ROLLBACK.

#### **USE Prueba1**

GO

CREATE TABLE Cabecera

(Id INT, Fecha DateTime)

GO

CREATE TABLE Detalle

(CabeceraId INT,

```
Linea INT NOT NULL,  
Cantidad DECIMAL(8,2) NOT NULL)
```

```
GO
```

```
CREATE TABLE Auditoria
```

```
(Id INT Identity,
```

```
Fecha DateTime NOT NULL)
```

```
GO
```

```
--Indagamos si hay Transacciones
```

```
SELECT @@TRANCOUNT
```

```
GO
```

```
EL Resultado que nos muestra sera 0
```

```
--Abrimos una transacción
```

```
BEGIN TRAN
```

```
--hacemos los insert
```

```
INSERT INTO Cabecera
```

```
VALUES (1,GETDATE())
```

```
INSERT INTO Detalle
```

```
Values (1,1,100)
```

```
--Observe que ya hay una transaccion abierta (1)
```

```
SELECT @@TRANCOUNT
```

```
EL Resultado que nos muestra sera 1
```

```
--Abrimos otra transacción
```

```
BEGIN TRAN
```

```
-- Verificamos que Trancount se incremento en 1
```

```
SELECT @@TRANCOUNT
```

```
EL Resultado que nos muestra sera 2
```

```
-- Hacemos un Rollback
```

```
ROLLBACK TRAN
```

```
-- Verificamos que Trancount quedo en 0
```

```
-- El rollback anulo todas las transacciones
```

```
SELECT @@TRANCOUNT
```

```
GO
```

```
EL Resultado que nos muestra sera 0
```

2.- El siguiente ejemplo muestra cómo funciona el Commit con el manejo de transacciones.

```
-- Abrimos una transacción
```

```
BEGIN TRAN
```

```
-- Realizamos los insert
```

```
INSERT INTO Cabecera
```

```
VALUES (1,GETDATE())
```

```
INSERT INTO Detalle
```

```
Values (1,1,100)
```

```
-- vemos las transacciones abiertas (1)
SELECT @@TRANCOUNT
-- Abrimos otra transacción
BEGIN TRAN
-- Verificamos que Trancount se incremento en 1
SELECT @@TRANCOUNT
-- Insetamos datos en auditoria y confirmamos la accion con commit
INSERT INTO Auditoria (Fecha)
VALUES (GETDATE())
COMMIT TRAN
-- Verificamos que Trancount decrecio en 1
SELECT @@TRANCOUNT
-- Realizamos el utimo Commit
COMMIT TRAN
-- Verificamos que Trancount queda en 0
SELECT @@TRANCOUNT
GO
```

Con los dos ejemplos realizados debe estar claro como funciona **Rollback y el Commit**.

3.- Ejecutar Una transacción, para insertar una fila de la tabla productos de la BD VENTA.

```
USE VENTA
GO
BEGIN TRANSACTION
INSERT INTO Producto(IdProducto,Nombre,Precio)
VALUES('P009','Swich 5 puertos',22.50)
COMMIT TRANSACTION
GO
```

La transacción ejecutada es sencilla, solo inserta una fila y luego la confirma.

4.- Ejecute una transacción para eliminar la fila insertada en el ejemplo anterior.

```
BEGIN TRANSACTION
DELETE Producto
WHERE IdProducto='P009'
IF @@ERROR=0
COMMIT TRANSACTION
ELSE
ROLLBACK TRANSACTION
GO
```

En la transaccion ejecutada primero se borra la fila insertada, luego se comprueba si se produjo un error, si no se produjo error, @@ERROR=0 y se ejecuta COMMIT

TRANSACTION; si se hubiera producido un error, entonces se anula el borrado de la fila con ROLLBACK TRANSACTION.

### 5.- Transacción explícita

1. Digite y ejecute las siguientes instrucciones en el Code Editor:

USE Prueba

GO

```
CREATE TABLE Padre(  
codPadre INT PRIMARY KEY,  
nomPadre varchar(15) NOT NULL)  
CREATE TABLE Hijo(  
codHijo INT PRIMARY KEY,  
nomHijo varchar(15) NOT NULL,  
codPadre INT NOT NULL FOREIGN KEY  
REFERENCES Padre)  
GO
```

2. Defina y ejecute la siguiente transacción explícita. En ella se ha establecido Intencionalmente un error de integridad referencial.

```
BEGIN TRANSACTION  
INSERT INTO Padre VALUES(100, 'Juan')  
IF (@@error <> 0) GOTO hayError  
INSERT INTO Hijo VALUES(101, 'Pedro', 99)  
-- padre 99 no existe  
IF (@@error <> 0) GOTO hayError  
COMMIT TRANSACTION  
RETURN  
hayError:  
ROLLBACK TRANSACTION  
GO  
--Ahora consulte las tablas. Note que ninguno de los  
--INSERT se ha confirmado.  
SELECT * FROM Padre  
SELECT * FROM Hijo  
GO
```

### Transacciones Anidadas

Son transacciones que pueden involucrar a otros dentro de ellas, y estas a su vez a otras. Las transacciones anidadas pueden ser ejecutadas por partes por tal razón el rollback no siempre se va a cero "0" si no a un punto que podemos determinar.

- 1.- El siguiente ejemplo nos ilustra el uso SAVE TRAN que como sabemos nos permite establecer un punto de restauración en la transacción, además usamos ROLLBACK en transacciones anidadas.

--Vemos que no hay Transacciones

```
SELECT @@TRANCOUNT
```

```
GO
```

--Abrimos una transacción

```
BEGIN TRAN A
```

--hacemos los insert

```
INSERT INTO Cabecera
```

```
VALUES (1,GETDATE())
```

```
INSERT INTO Detalle
```

```
Values (1,1,100)
```

--Hacemos un punto de salvacion

```
SAVE TRAN B
```

--vemos las transacciones abiertas (1)

```
SELECT @@TRANCOUNT
```

--Abrimos otra transacción

```
BEGIN TRAN C
```

--Verificamos que Trancount se incremento en 1

```
SELECT @@TRANCOUNT
```

--Hacemos un Rollback

```
INSERT INTO Auditoria (Fecha)
```

```
VALUES (GETDATE())
```

```
ROLLBACK TRAN B
```

--Verificamos que Trancount es 2 y no 0

--Esto se da porque el rollback afecta

--el punto de almacenamiento y no la transacción

--en si

```
SELECT @@TRANCOUNT
```

--Hacemos los Commit

```
COMMIT TRAN
```

```
COMMIT TRAN
```

--Verificamos que Trancount queda en 0

```
SELECT @@TRANCOUNT
```

```
GO
```

--Verificamos los datos y vemos que en Auditoria

--no se guardaron registros porque hubo un rollback

--pero solo de ese punto

```
SELECT * FROM Cabecera
```

```
SELECT * FROM Detalle
```

```
SELECT * FROM Auditoria
```

```
GO
```

**EJC5.7.-** En el siguiente ejemplo realizamos otra aplicación de SAVE TRAN

**USE Prueba**

```
GO
CREATE TABLE Tabla1 (Columna1 varchar(50))
GO
BEGIN TRAN
INSERT INTO Tabla1 VALUES ('Primer valor')
    SAVE TRAN Punto1
    INSERT INTO Tabla1 VALUES ('Segundo valor')
    ROLLBACK TRAN Punto1
    INSERT INTO Tabla1 VALUES ('Tercer valor')
COMMIT TRAN
SELECT * FROM Tabla1
GO
```

Un ROLLBACK a un SAVE TRAN no deshace la transacción en curso ni modifica @@TRANCOUNT, simplemente cancela lo ocurrido desde el 'SAVE TRAN nombre' hasta su 'ROLLBACK TRAN nombre'

### EL CHECKPOINT Y LA RECUPERACIÓN DE TRANSACCIONES

SQL Server garantiza que todas las transacciones confirmadas se reflejarán en la base de datos en la eventualidad de una falla del sistema. Para ello, utiliza el log de transacciones para escribir en la base de datos las transacciones confirmadas (committed transactions), y deshace las transacciones no confirmadas (uncommitted transactions).

SQL Server ejecuta periódicamente el checkpoint, que consiste en la verificación del log de transacciones para determinar las transacciones confirmadas.

El siguiente diagrama resume algunas situaciones que pueden presentarse cuando se produce una falla del sistema.

- Para la transacción 1 se verificó COMMIT antes del checkpoint, por lo tanto ya está confirmada plenamente en la base de datos.
- Para las transacciones 2 y 3 el COMMIT se verificó después del checkpoint, por lo tanto deben ser reconstruidas desde el log de transacciones.
- Para las transacciones 4 y 5 no se verificó el COMMIT, por lo tanto deben ser deshechas.