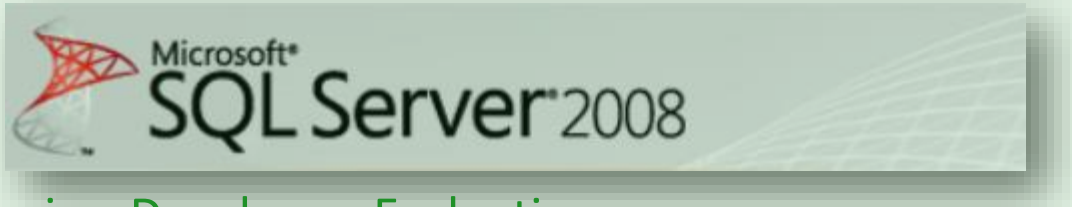
INSTITUTO TECNOLÓGICO SUPERIOR DE
SAN ANDRÉS TUXTLA

Introducción a Transact-SQL

Juan Salvador Rodriguez Aguirre

Firmado digitalmente por Juan Salvador Rodriguez Aguirre
Fecha: 2017.08.16 12:24:13 -05'00'

Versiones de 

- Enterprise, Developer, Evaluation
- Estándar
- Grupo de trabajo
- Express
- Express con Servicios avanzados
- Compacta

Tipos de Datos

Son:

- Numéricos
- Carácter
- Fecha
- Binarios
- XML
- Definidos por el usuario

Numéricos

Tipo de Dato	Rango	Tamaño
tinyint	0 hasta 255	1 byte
smallint	-32.768 Y 32.767	2 bytes
int	-231 A 231	4 bytes
bigint	-263 A 263 -1	8 bytes
decimal (p, s) numeric (p, s)	-1038 1-1038 -1	5 a 17 bytes
smallmoney	-214.748,3648 A 214,748.3647	4 bytes
money	-922.337.203.685.477,5808 A 922,337,203,685,477.5807	8 bytes
real	-3,438 A -1,1838, 0, y 1,1838 a 3,438	4 bytes
float(n)	-1,79308 A -2,23308, 0, y 2,23308 a 1,79308	4 bytes u 8 bytes

Numéricos

```
DECLARE @bit bit,
        @tinyint tinyint,
        @smallint smallint,
        @int int,
        @bigint bigint,
        @decimal decimal(10,3),

        @real real,
        @double float(53),
        @money money
set @bit = 1
print @bit
set @tinyint = 255
print @tinyint
set @smallint = 32767
print @smallint
set @int = 655372229
print @int
set @decimal = 56565.234
print @decimal
set @money = 99.555
print @money
```

Carácter

Tipo de datos	Espacio de almacenamiento
char (n)	1 byte por carácter definido por n hasta un máximo de 8000
varchar (n)	bytes 1 byte por carácter almacenado hasta un máximo de 8000
texto	bytes 1 byte por carácter almacenado hasta un máximo de 2 GB
nchar (n)	2 bytes por carácter definido por n hasta un máximo de 4000 bytes
nvarchar (n)	2 bytes por carácter almacenado hasta un máximo de 4000 bytes
ntext	2 bytes por carácter almacenado hasta un máximo de 2 GB

Tipos de Datos Carácter

Datos de caracteres de longitud fija y variable

SQL Server le permite definir los datos de caracteres, ya sea como longitud fija o variable.
El número de caracteres definidos establece el número máximo de caracteres que se permiten almacenar en una columna.

Cuando los datos se almacenan en un tipo de datos char o nchar, la cantidad de almacenamiento consumido es igual a la definición de almacenamiento del tipo de datos, independientemente del número de caracteres colocados en la columna. Cualquier espacio que no es consumido por los datos se rellena con espacios.

Cuando los datos se almacenan en un tipo de datos nvarchar o varchar, la cantidad de almacenamiento consumido es igual al número de caracteres efectivamente almacenados

Datos Unicode

Los datos de caracteres se pueden almacenar usando un conjunto ANSI o uno de caracteres Unicode. El conjunto de caracteres ANSI abarca la mayor parte de los caracteres que se utilizan en la mayoría de los idiomas en todo el mundo. Sin embargo, ANSI sólo abarca un poco más de 32.000 caracteres. Varios idiomas como el árabe, el hebreo y algunos dialectos chinos contienen más de 32.000 caracteres del alfabeto estándar. Con el fin de almacenar la amplia gama de caracteres, se necesitan 2 bytes de almacenamiento para cada uno.

SQL Server le permite especificar si el almacenamiento de datos de una columna es Unicode o no Unicode. ***Los tipos de datos Unicode*** comienzan con una n y son **nchar (n), nvarchar (n), y ntext**.

Tipos de Datos Fecha y Hora

Tipo de datos	Rango de Valores	Precisión	Espacio de almacenamiento
smalldatetime	01/01/1900 hasta 06/06/2079	1 minuto	4 bytes
datetime	01/01/1753 a 12/31/9999	0,00333 segundos	8 bytes
datetime2	01/01/0001 a 12/31/9999	100 nanosegundos	6 a 8 bytes
datetimeoffset	01/01/0001 a 12/31/9999	100 nanosegundos	8 a 10 bytes
date	01/01/0001 a 12/31/9999	1 día	3 bytes
time	00:00:00.0000000 a 23:59:59.9999999	100 nanosegundos	3 a 5 bytes

smalldatetime y datetime

Los tipos de datos smalldatetime y datetime almacenan ambos una fecha y una hora como un valor único. SQL Server 2008 introduce el tipo de datos datetime2 que debería sustituir tanto al tipo de datos smalldatetime y el datetime, debido a una mejor precisión, así como la capacidad de manejar un mayor rango de fechas.

El datetimeoffset le permite almacenar una zona horaria para aplicaciones que necesitan localizar fechas y horas.

smalldatetime y datetime

```
declare @horacorta smalldatetime
set @horacorta=(Select CONVERT(nvarchar(10), GETDATE(), 112) AS Hora)
print 'Hora Corta '+ cast(@horacorta as varchar)
```

Pruebe cambiando el tipo de dato smalldatetime por datetime y el valor 112 por 103 y 108

Funciones para fecha y hora

HH:MM:SS

```
Select CONVERT(nvarchar(10), GETDATE(), 108) AS Hora
```

Fecha dd/mm/yyyy

```
SELECT CONVERT(CHAR(10), GETDATE(), 103)
```

Sin formato aniomesdia

```
SELECT CONVERT(CHAR(10), GETDATE(), 112)
```

Horas y Minutos

```
SELECT CONVERT(CHAR(5), GETDATE(), 108)
```

Tipos de Datos Binarios

Tipos de datos	Rango de valores	Espacio de almacenamiento
bit	Null, 0 y 1	1 bit
binary	Fijo con datos binarios	Hasta 8000 bytes
varbinary	Datos binarios de longitud variable	Hasta 8000 bytes
image	Datos binarios de longitud variable	Hasta 2 GB

Tipo de dato image

Está obsoleto a partir de SQL Server 2005. No se debería usar image en ningún desarrollo. Las tablas que tienen datos de tipo image deben ser modificados para usar varbinary (max) en su lugar.

Al igual que los tipos de datos caracteres de longitud variable, se puede aplicar la palabra clave max para el tipo de datos varbinary para permitir el almacenamiento de hasta 2 GB de datos, a la vez que soporta todas las funciones de programación disponibles para la manipulación de datos binarios

XML

Permite almacenar y manipular documentos XML de forma nativa. Cuando almacena documentos XML, está limitado a un ***máximo de 2 GB*** así como un ***máximo de 128 niveles*** dentro de un documento.
Los documentos XML son útiles porque almacenan los datos y la estructura en un único archivo, que permite la transferencia de datos independiente de la plataforma. La estructura de un documento XML es llamado XML schema.

Además de definir una columna con un tipo de datos XML, también puede limitar los tipos de documentos XML que están permitidos para ser almacenados dentro de la columna. Usted limita los tipos de documentos XML creando una colección de esquemas XML y asociando la colección de esquema a la columna XML. Al insertar o actualizar un documento XML, se validará contra la colección de esquemas XML para asegurarse de que sólo está insertando documentos XML que cumplen con uno de los esquemas permitidos.

XML

```
DECLARE @myxml XML

 set @myxml = (SELECT @@LANGUAGE idioma FOR XML RAW, TYPE)

 print cast(@myxml as varchar(max))


 SELECT  NOMBRESUCURSAL NS, CIUDADSUCURSAL CS, ACTIVOS ACT
FROM SUCURSAL
FOR XML AUTO
```

Tipos de Datos Definidos por el usuario

```
CREATE TYPE [tipo_definido_usuario] FROM [tipo_dato];
CREATE TYPE iva FROM decimal(10,2) NOT NULL;

CREATE RULE vporcentaje AS @p>=0 AND @p<=100
go
CREATE DEFAULT d_iva AS 16
go
EXEC sp_addtype iva2, 'decimal(10,2)','not null'
go
EXEC sp_bindrule 'vporcentaje','iva2'
go
EXEC sp_bindefault 'd_iva','iva2'
go
sp_help iva2
```

Tipos de Datos Definidos por el usuario

Ejercicio:

1. Crear una tabla donde se emplee el tipo de datos iva2
2. Emplear el procedimiento sp_helpconstraint [tabla] para determinar que se afectó el valor predeterminado del campo iva2.
3. Ingresar un registro sin valor y verificar que se almacenó

Tipos de Datos Definidos por el usuario

```
create table datos(
impuesto iva2,
id int
);
go
sp_helpconstraint datos;
```

Resultados | Mensajes

	Object Name
1	datos

	constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constraint_keys
1	DEFAULT on column impuesto (bound with sp_bindef...	d_iva	(n/a)	(n/a)	(n/a)	(n/a)	CREATE DEFAULT d_iva AS 16
2	RULE on column impuesto (bound with sp_bindrule)	vporcentaje	(n/a)	(n/a)	(n/a)	(n/a)	CREATE RULE vporcentaje AS @p>=0 AND @p<=100

```
insert into datos default values;
select * from datos;
```

Resultados | Mensajes

	impuesto	id
1	16.00	NULL

Declaraciones de Variables y Estructuras de Control

Declaración de Variables

Las variables proporcionan una manera de manipular, almacenar y transmitir datos dentro de un procedimiento almacenado, así como entre los procedimientos almacenados y funciones. SQL Server tiene dos tipos de variables:

- local y
- global.

La variable local se designa mediante un solo símbolo @, mientras que una variable global se designa con un doble símbolo @@.

Además, puede crear, leer y escribir variables locales mientras que no puede crear o escribir variables globales.

Declaraciones de Variables y Estructuras de Control

Variables Globales

Variable Global	Definición
@ @ ERROR	Código de error de la última sentencia ejecutada
@ @ IDENTITY	Valor del último número de identidad insertada dentro de la conexión
@ @ ROWCOUNT	El número de filas afectadas por la última instrucción
@ @ TRANCOUNT	El número de transacciones abiertas dentro de la conexión
@ @ VERSION	La versión de SQL Server

Declaración de Variables

Para instanciar una variable se usa la cláusula DECLARE donde se especifica el nombre y el tipo de datos de la variable. Una variable puede ser definida usando cualquier tipo de datos excepto text, ntext e image. Por ejemplo:

```
DECLARE @intvariable INT,
@datevvariable DATE

DECLARE @tablevar TABLE
(ID INT NOT NULL,
Cliente VARCHAR(50) NOT NULL)
```

Nota Los tipos de datos text, ntext e image han quedado en desuso y no deben ser utilizados

Declaración de Variables

Aunque una sola declaración DECLARE se puede utilizar para crear instancias de múltiples variables, la instanciación de una variable de tabla tiene que estar en una DECLARE separada.

Asignación de Variables

Se puede asignar a una variable un valor estático o un valor único devuelto desde una declaración SELECT.
Cualquiera de las dos SET o SELECT se puede utilizar para asignar un valor, sin embargo, si se está ejecutando
una consulta para asignar un valor, debe utilizar una sentencia SELECT. SELECT también se utiliza para devolver el
valor de una variable. Una variable puede ser utilizada para realizar cálculos, procesamiento de control, o como una
SARG en una consulta.

Asignación de Variables

Además de asignar un valor utilizando una declaración SET o SELECT, también se le puede asignar un valor en el momento que una variable es instanciada.

```
DECLARE @intvariable INT = 2,
@datevvariable DATE = GETDATE(),
@maxorderdate DATE = (SELECT MAX(OrderDate) FROM OrderHeader),
@counter1 INT,
@counter2 INT

SET @counter1 = 1
SELECT @counter2 = -1
SELECT @intvariable, @datevariable, @maxorderdate, @counter1, @counter2
```

Asignación de Variables

Se pueden realizar cálculos con variables utilizando ya sea una sentencia SET o SELECT. SQL Server 2008 introduce una forma más compacta para la asignación de valores a las variables utilizando un cálculo.

Declaraciones de Variables y Estructuras de Control

Asignación de Variables en SQL 2005

```
--SQL Server 2005 and below
DECLARE @var INT
SET @var = 1
SET @var = @var + 1
SELECT @var

SET @var = @var * 2
SELECT @var

SET @var = @var / 4
SELECT @var
GO
```

Asignación de Variables en SQL 2008

```
--SQL Server 2008
DECLARE @var INT
SET @var = 1
SET @var += 1
SELECT @var

SET @var *= 2
SELECT @var

SET @var /= 4
SELECT @var
GO
```

Asignación de Variables

Con la excepción de una variable de tabla, todas las otras variables contienen un solo valor.

Aunque se puede asignar el resultado de una declaración SELECT a una variable, si hay más de una fila devuelta desde la declaración SELECT, usted no recibirá un error. La variable contendrá sólo el último valor del conjunto de resultados. Cualquier otro valor será descartado.

Cláusula OUTPUT

En el curso de las Instrucciones SQL, es posible que necesite recuperar el estado de las filas que están siendo modificadas. En el marco de las operaciones de DML, SQL Server ofrece dos tablas especiales denominadas ***inserted*** y ***deleted***. Estas tablas son creadas automáticamente Las tablas inserted y deleted están en el ámbito de una conexión y no pueden ser accedidas por cualquier otro usuario.
Estas tablas existen siempre que una ***modificación de datos*** está en curso, la transacción está abierta, y además son válidas sólo para la modificación de los datos que actualmente está siendo ejecutada.

La tabla inserted contiene el estado de la(s) fila(s) modificada(s) después que ya se producido la modificación, denominado como “la imagen después”.
La tabla deleted contiene el estado de la(s) fila(s) modificada(s) antes de la modificación de los datos, conocida como “la imagen antes”.

Cláusula OUTPUT

Dado que una declaración INSERT es sólo la adición de nuevas filas a una tabla, la tabla ***deleted*** siempre estará vacía. Cuando se ejecuta una instrucción DELETE, las filas se eliminan de la tabla, y la tabla ***inserted*** siempre estará vacía. Una declaración UPDATE, dado que modifica una fila ya existente, tendrá entradas tanto en la tabla ***inserted*** como en ***deleted***. Para una declaración MERGE, las tablas ***inserted*** y ***deleted***, se rellenan en función de si se ejecuta un INSERT, UPDATE, o DELETE desde la sentencia MERGE.

Antes de SQL Server 2005, sólo se podía acceder a las tablas inserted y deleted desde dentro un trigger.
Ahora puede acceder a estas dos tablas directamente en el ámbito de una declaración INSERT, UPDATE,DELETE, o MERGE mediante la utilización de la cláusula OUTPUT.

Cláusula OUTPUT

La cláusula OUTPUT se puede utilizar de dos maneras dentro de una operación DML:

1. Para devolver como resultado el contenido de las tablas insertadas o eliminados directamente a una aplicación.
2. Para insertar el contenido de las tablas inserted y/o deleted en una tabla o variable de tabla.

Estructura Condicional IF

IF. . . ELSE proporciona la capacidad de ejecutar código condicionalmente. La declaración IF comprobará la
condición suministrada y ejecutará el siguiente bloque de código cuando la condición sea verdadera. La sentencia
opcional ELSE permite ejecutar código cuando el chequeo de la condición es falso.

```
DECLARE @var INT
SET @var = 1
        IF @var = 1
        PRINT 'Este código es ejecutado cuando es true.'
        ELSE
        PRINT ' Este código es ejecutado cuando es false.'
```

Estructura Condicional IF

Independientemente de la rama que toma el código para un IF. . . ELSE, sólo la declaración siguiente se ejecutara condicionalmente.

```
DECLARE @var INT
SET @var = 1

    IF @var = 2
    PRINT 'Este código es ejecutado cuando es true.'
    PRINT 'Esto siempre se ejecutará.'
```

Estructura Condicional IF

Dado que una declaración IF condicionalmente ejecutará solo la siguiente línea de código, usted tiene un problema cuando quiere ejecutar un bloque de código condicional. El BEGIN. . . END permite delimitar bloques de código que se ejecutan como una unidad.

```
DECLARE @var INT
SET @var = 1

    IF @var = 2
    BEGIN
    PRINT 'Este código es ejecutado cuando es true.'
    PRINT ' Este código también es ejecutado solo cuando la condición sea true.'
    END
```

Estructura Condicional CASE

Si usted necesita devolver condicionalmente un valor en un conjunto de resultados, puede utilizar la función CASE que tiene una sintaxis genérica de:

```
CASE input_expression
WHEN when_expression THEN result_expression
[ ...n ]
[ ELSE else_result_expression ]
END
```

Estructura Condicional CASE

```
--Ejemplo
DECLARE @Titulo varchar(100),
        @Carrera varchar(3)
    SET @Carrera = 'INF'

    SET @Titulo = (CASE @Carrera
                     WHEN 'ISC' THEN 'Ing en Sistemas'
                     WHEN 'IINF' THEN 'Ing Informática'
                     ELSE 'Otra carrera'
                END)
       PRINT @Titulo
```

Estructura Condicional CASE

La siguiente consulta devuelve un valor diferente en el conjunto de resultados basándose en el valor de la columna fxpago de la tabla PAGO.

```
SELECT FXPAGO,CASE
WHEN MONTH(FXPAGO)= 5 THEN 'PAGOS DE MAYO'
WHEN MONTH(FXPAGO)= 6 THEN 'PAGOS DE JUNIO'
ELSE 'NO ENCONTRADO'
END FXPAGO
    FROM PAGO ORDER BY PAGO.FXPAGO
```

Estructura WHILE

WHILE se utiliza para ejecutar iterativamente un bloque de código mientras una condición especificada sea verdadera.

```
DECLARE @var1 INT,
                    @var2 VARCHAR(30)
SET @var1 = 1
WHILE @var1 <= 10
BEGIN
        SET @var2 = 'Iteration #' + CAST(@var1 AS VARCHAR(2))
        PRINT @var2
        SET @var1 += 1
END
```

Estructura WHILE (BREAK/CONTINUE)

BREAK se utiliza junto con un bucle while. Si usted necesita terminar la ejecución en un bucle WHILE, puede utilizar la declaración BREAK para poner fin a la repetición del bucle. Una vez que BREAK es ejecutada, el código seguirá ejecutando, desde la siguiente línea de código que sigue al bucle WHILE.

CONTINUE se utiliza dentro de un bucle WHILE para que el código se siga ejecutando dentro del bucle.

Nota BREAK/CONTINUE casi nunca se utilizan. Un bucle while terminará tan pronto como la condición para el bucle WHILE ya no es verdadera. En lugar de incorporar una prueba condicional junto con una declaración BREAK, los bucles WHILE son controlados normalmente mediante el uso de un adecuado condicional para el WHILE. Mientras el condicional para el WHILE es verdadero, el bucle continuará en ejecución. Por lo tanto, nunca debe tener la necesidad de utilizar una sentencia CONTINUE.

Estructura WAITFOR

WAITFOR se utiliza para permitir que en la ejecución de código se haga una pausa. Tiene tres permutaciones diferentes: WAITFOR DELAY, WAITFOR TIME y WAITFOR RECEIVE.

- WAITFOR RECEIVE se utiliza junto con el Agente de servicio (Service Broker).
- WAITFOR TIME detiene la ejecución de código hasta que se alcanza una hora determinada.
- WAITFOR DELAY interrumpe la ejecución de código por un intervalo de tiempo especificado

Estructura WAITFOR

```
DECLARE @var1 INT,
@var2 VARCHAR(30)
SET @var1 = 1
--Genera una pausa de 2 segundos
WAITFOR DELAY '00:00:02'
--WAITFOR TIME '13:01:59'
WHILE @var1 <= 10
BEGIN
SET @var2 = 'Bucle #' + CAST(@var1 AS VARCHAR(2))
PRINT @var2
SET @var1 += 1
END
```

Estructura GOTO

GOTO permite pasar la ejecución a una etiqueta incorporada dentro de un procedimiento. En ningún código que pueda encontrar, es aconsejable incluir construcciones tales como GOTO.

Declaraciones de Variables y Estructuras de Control

Ejercicios

- Crear 2 variables coSucursal y ciudadSucursal
- Asignar los valores para dichas variables SAT y San Andrés Respectivamente
- Realizar la consulta para verificar si existe la Sucursal SAT
  - Si existe actualizar la ciudad de la Sucursal por San Andrés
- Si no existe insertar dicha Sucursal con la ciudad sucursal correspondiente en una tabla denominada PRUEBA que contendrá los campos (ID,N) con los mismos tipos de datos que la tabla SUCURSAL de la base de datos BANCO.

Control de Errores y Transacciones

Estructura TRY CATCH

Sintaxis

```
BEGIN TRY
    ...
END TRY

BEGIN CATCH
    ...
END CATCH
```

Estructura TRY CATCH

En un mundo perfecto, cada bloque de código que se ejecuta siempre se ejecute sin errores.
Sin embargo, todo código siempre estará sujeto a fallos. Por lo tanto, es necesario incluir un manejo de errores en los procedimientos almacenados.

Antes de SQL Server 2005, la única manera de realizar el tratamiento de errores era poner a prueba el valor de la variable global @@ Error. Ahora tiene una forma de realizar un control de errores estructurado similar a otros lenguajes de programación, mediante la utilización de un bloque TRY. . . CATCH.

El bloque TRY. . . CATCH tiene dos componentes. El bloque TRY se utiliza para englobar cualquier código que puede recibir un error y que se desea atrapar y manejar. El bloque CATCH se utiliza para controlar el error.

Estructura TRY CATCH

El siguiente código crea un error debido a la violación de una restricción de clave primaria. Usted puede ser que espere que este código deje la tabla vacía debido al error en la transacción, sin embargo, usted encontrará que las instrucciones insert primera y tercera se ejecutan con éxito y dejan dos filas en la tabla.

```
CREATE TABLE dbo.mytable
(ID INT NOT NULL PRIMARY KEY)

BEGIN TRAN
INSERT INTO dbo.mytable VALUES(1)
INSERT INTO dbo.mytable VALUES(1)
INSERT INTO dbo.mytable VALUES(2)
COMMIT TRAN

SELECT * FROM dbo.mytable
```

Estructura TRY CATCH

La razón por la que tiene dos filas insertadas en la tabla se debe a que de forma predeterminada, SQL Server no revierte una transacción que tiene un error. Si desea que la transacción sea completada en su totalidad o que falle toda la transacción completa, se puede utilizar el comando SET para cambiar la configuración del parámetro XACT_ABORT de la conexión, de la siguiente manera:

```
SET XACT_ABORT ON;
BEGIN TRAN
INSERT INTO dbo.mytable VALUES(1)
INSERT INTO dbo.mytable VALUES(1)
INSERT INTO dbo.mytable VALUES(2)
COMMIT TRAN

SET XACT_ABORT OFF;
SELECT * FROM dbo.mytable
```

Estructura TRY CATCH

Aunque la declaración SET logra su objetivo, al cambiar la configuración de una conexión, puede tener resultados impredecibles para una aplicación si el código no reinicia correctamente las opciones. Una mejor solución es usar un controlador estructurado de errores para atrapar y decidir cómo manejar el error.

Estructura TRY CATCH

```
--TRY...CATCH
TRUNCATE TABLE dbo.mytable

BEGIN TRY
        BEGIN TRAN
                INSERT INTO dbo.mytable VALUES(1)
                INSERT INTO dbo.mytable VALUES(1)
                INSERT INTO dbo.mytable VALUES(2)
        COMMIT TRAN
END TRY

BEGIN CATCH
        ROLLBACK TRAN
        PRINT 'Catch'
END CATCH
```

Funciones de Error

Además de proporcionar una rutina de control de errores estructurada, también puede eliminar la devolución de códigos de error fatal que podrían hacer que el código falle inesperadamente. Usted notará en el código anterior que, si bien aún se lanza un error, el bloque CATH atrapa el error, se encarga del manejo del problema y, a continuación, devuelve el control sin el mensaje de error, que ha visto antes en los dos bloques de código.

Las funciones especiales de error, están disponibles únicamente en el bloque CATCH para la obtención de información detallada del error

Funciones de Error

- ERROR_NUMBER(), devuelve el número de error.
- ERROR_SEVERITY(), devuelve la severidad del error.
- ERROR_STATE(), devuelve el estado del error.
- ERROR_PROCEDURE(), devuelve el nombre del procedimiento almacenado que ha provocado el error.
- ERROR_LINE(), devuelve el número de línea en el que se ha producido el error.
- ERROR_MESSAGE(), devuelve el mensaje de error.

Son extremadamente útiles para realizar una auditoría de errores.

Funciones de Error

```
BEGIN TRY
    DECLARE @divisor int,@dividendo int,@resultado int
    SET @dividendo = 100
    SET @divisor = 0
    -- Esta linea provoca un error de division por 0
    SET @resultado = @dividendo/@divisor
    PRINT 'No hay error'
 END TRY
 BEGIN CATCH
    PRINT ERROR_NUMBER()
    PRINT ERROR_SEVERITY()
    PRINT ERROR_STATE()
    PRINT ERROR_PROCEDURE()
    PRINT ERROR_LINE()
    PRINT ERROR_MESSAGE()
 END CATCH
```

Variable del sistema @@ERROR

Lógicamente, podemos utilizar estas funciones para almacenar esta información en una tabla de la base de datos y registrar todos los errores que se produzcan.

En versiones anteriores a SQL Server 2005, no estaban disponibles las instrucciones TRY CATCH. En estas versiones se controlaban los errores utilizando la variable global de sistema @@ERROR, que almacena el número de error producido por la última sentencia Transact SQL ejecutada.

Variable del sistema @@ERROR

```
DECLARE @divisor   int ,
        @dividendo int ,
        @resultado int
SET @dividendo = 100
SET @divisor = 0
-- Esta linea provoca un error de division por 0
SET @resultado = @dividendo/@divisor

IF @@ERROR = 0
   BEGIN
      PRINT 'No hay error'
   END
ELSE
   BEGIN
    PRINT 'Hay error'
   END
```

Conceptos

Los Motores de base de datos no están diseñados para simplemente almacenar datos. Una característica fundamental de cualquier motor de base de datos es garantizar la consistencia de datos al tiempo que permite el máximo acceso simultáneo a los mismos. Si nunca cambiaran los datos dentro de las bases de datos de SQL Server, el tema de la consistencia nunca ocurriría. Sin embargo, un usuario puede cambiar los datos mientras otros usuarios están intentando leer los mismos datos.

Cada vez que un usuario realiza un cambio a los datos, puede decidir ya sea guardarlos, COMMIT, o descartarlos, ROLLBACK. SQL Server tiene que garantizar que los usuarios que leen los datos siempre reciban datos que se han confirmado en la base (commit), con el fin de asegurar resultados consistentes.

Conceptos

Cada unidad de trabajo que usted envíe a SQL Server es una transacción. Cada transacción está delimitada ya sea implícita o explícitamente. Una transacción se construye con BEGIN TRAN, seguido por uno o más comandos a ejecutar. La operación es finalizada por un COMMIT TRAN o ROLLBACK TRAN. Una transacción explícita se produce cuando usted escribe el código BEGIN TRAN. . . COMMIT TRAN / ROLLBACK TRAN.

Una transacción implícita se produce cuando sólo hay que ejecutar un comando tal como un INSERT, UPDATE, o DELETE y deja en manos de SQL Server anteponer el o los comando(s) con un BEGIN TRAN y finalizar el lote con un COMMIT TRAN.

Por ejemplo, todos los scripts que se han presentado hasta ahora ha sido una transacción implícita.

Sintaxis para Definir una Transacción

Las transacciones se pueden anidar. Por ejemplo

```
BEGIN TRAN
<do something>
        BEGIN TRAN
        <do something>
                BEGIN TRAN
                <do something>
                COMMIT TRAN
        COMMIT TRAN
COMMIT TRAN
```

Transacciones

BEGIN TRAN iniciará una nueva transacción. COMMIT TRAN guarda la transacción más interna.
ROLLBACK TRAN revertirá todas las transacciones para la conexión, deshaciendo cualquier cambio que se haya hecho, y liberando todos los bloqueos.

BEGIN TRAN y COMMIT TRAN delimitan una transacción. SQL Server aplica bloqueos sobre los datos para asegurar que las aplicaciones siempre puedan recuperar un conjunto de datos consistente. Los bloqueos se apoderan de un recurso cuando comienza la transacción y se libera cuando la transacción se confirma o se revierte. Los tipos de bloqueo utilizados por SQL Server son:

- Compartido (shared)
- Exclusivo (exclusive)
- Actualización (update)

Ejemplo de Transacciones

```
BEGIN TRY
        BEGIN TRANSACTION;
DECLARE @coSucursal INT,
        @ciudadSucursal varchar(30)
set @coSucursal = 'SAT'
set @ciudadSucursal = 'San Andrés'

                IF EXISTS(SELECT * FROM SUCURSAL
                                WHERE NOMBRESUCURSAL = @coSucursal)
                 BEGIN
                        UPDATE PRUEBA
                        SET N = @ciudadSucursal
                        WHERE ID = @coSucursal
                        COMMIT TRANSACTION;
                 END
```

Ejemplo de Transacciones (Continua)

```
ELSE
                    BEGIN
                            INSERT INTO PRUEBA
                            (ID, N) VALUES
                            (@coSucursal, @ciudadSucursal)
                            COMMIT TRANSACTION;
                    END
END TRY
BEGIN CATCH
            PRINT 'Numero de Error: '+ CAST(ERROR_NUMBER() AS VARCHAR(10))
    PRINT 'Grado de Error:' + CAST(ERROR_SEVERITY() AS VARCHAR(10))
    PRINT 'Estado de Error: '+ CAST(ERROR_STATE() AS VARCHAR(10))
    PRINT 'Proc. de Error: '+COALESCE(ERROR_PROCEDURE(),'NADA')
    PRINT 'Error en linea: '+ CAST(ERROR_LINE() AS VARCHAR(10))
    PRINT 'Mensaje de error: '+ ERROR_MESSAGE()
ROLLBACK TRANSACTION
```

Tipos de Transacciones

- Transacciones explícitas.

Cada transacción se inicia explícitamente con la instrucción BEGIN TRANSACTION y se termina explícitamente con una instrucción COMMIT o ROLLBACK.

- Transacciones implícitas.

Se inicia automáticamente una nueva transacción cuando se ejecuta una instrucción que realiza modificaciones en los datos, pero cada transacción se completa explícitamente con una instrucción COMMIT o ROLLBACK.

Para activar-desactivar el modo de transacciones implícitas debemos ejecutar la siguiente instrucción.

```
--Activamos el modo de transacciones implícitas
SET IMPLICIT_TRANSACTIONS ON

--Desactivamos el modo de transacciones implícitas
SET IMPLICIT_TRANSACTIONS OFF
```

Ejemplo Transacciones explícitas

```
select * from cuenta
go
IF OBJECT_ID (N'dbo.MOVIMIENTOS', N'U') IS NOT NULL
DROP TABLE dbo.MOVIMIENTOS
CREATE TABLE MOVIMIENTOS(
IDCUENTA VARCHAR(5),
SALDO_ANTERIOR DECIMAL(10,2),
SALDO_POSTERIOR DECIMAL(10,2),
IMPORTE DECIMAL(10,2),
FXMOVIMIENTO DATE
)
GO
```

Ejemplo Transacciones explícitas

```
DECLARE @importe DECIMAL(18,2),
        @CuentaOrigen VARCHAR(12),
        @CuentaDestino VARCHAR(12)

/* Asig el importe de la transferencia y las cuentas de origen y destino*/
SET @importe = 50
SET @CuentaOrigen = 'C-101'
SET @CuentaDestino = 'C-102'
```

Ejemplo Transacciones explícitas

```
BEGIN TRANSACTION -- O solo BEGIN TRAN
BEGIN TRY
/* Descontamos el importe de la cuenta origen */
UPDATE CUENTA
SET SALDO = SALDO - @importe
WHERE NUMCUENTA = @CuentaOrigen

/* Registramos el movimiento */
INSERT INTO MOVIMIENTOS
(IDCUENTA, SALDO_ANTERIOR, SALDO_POSTERIOR,
 IMPORTE, FXMOVIMIENTO)
SELECT
NUMCUENTA, SALDO + @importe, SALDO, @importe, getdate()
FROM CUENTA
WHERE NUMCUENTA = @CuentaOrigen
```

Ejemplo Transacciones explícitas

```
/* Incrementamos el importe de la cuenta destino */
UPDATE CUENTA
SET SALDO = SALDO + @importe
WHERE NUMCUENTA = @CuentaDestino

/* Registramos el movimiento */
INSERT INTO MOVIMIENTOS
(IDCUENTA, SALDO_ANTERIOR, SALDO_POSTERIOR,
 IMPORTE, FXMOVIMIENTO)
SELECT
NUMCUENTA, SALDO - @importe, SALDO, @importe, getdate()
FROM CUENTA
WHERE NUMCUENTA = @CuentaDestino
```

Ejemplo Transacciones explícitas

```
/* Confirmamos la transaccion*/
COMMIT TRANSACTION -- O solo COMMIT

END TRY
BEGIN CATCH
/* Hay un error, deshacemos los cambios*/
ROLLBACK TRANSACTION -- O solo ROLLBACK
PRINT 'Se ha producido un error!'
END CATCH
go
select * from cuenta
go
```

INSTITUTO TECNOLÓGICO SUPERIOR DE SAN ANDRÉS TUXTLA

Ejemplo de Transacciones

Parte 1

```
select * from cuenta
go
IF OBJECT_ID (N'dbo.MOVIMIENTOS', N'U') IS NOT NULL
DROP TABLE dbo.MOVIMIENTOS
CREATE TABLE MOVIMIENTOS(
IDCUENTA VARCHAR(5),
SALDO_ANTERIOR DECIMAL(10,2),
SALDO_POSTERIOR DECIMAL(10,2),
IMPORTE DECIMAL(10,2),
FXMOVIMIENTO DATE
)
GO
DECLARE @importe DECIMAL(18,2),
                 @CuentaOrigen VARCHAR(12),
                 @CuentaDestino VARCHAR(12)

/* Asig el importe de la transf y las cuentas de origen y destino*/
SET @importe = 50
SET @CuentaOrigen = 'C-101'
SET @CuentaDestino = 'C-102'

BEGIN TRANSACTION -- O solo BEGIN TRAN
BEGIN TRY
/* Descontamos el importe de la cuenta origen */
UPDATE CUENTA
SET SALDO = SALDO - @importe
WHERE NUMCUENTA = @CuentaOrigen
```

Parte 2

```
/* Registramos el movimiento */
INSERT INTO MOVIMIENTOS
(IDCUENTA, SALDO_ANTERIOR, SALDO_POSTERIOR,
 IMPORTE, FXMOVIMIENTO)
SELECT
NUMCUENTA, SALDO + @importe, SALDO, @importe, getdate()
FROM CUENTA
WHERE NUMCUENTA = @CuentaOrigen

/* Incrementamos el importe de la cuenta destino */
UPDATE CUENTA
SET SALDO = SALDO + @importe
WHERE NUMCUENTA = @CuentaDestino

/* Registramos el movimiento */
INSERT INTO MOVIMIENTOS
(IDCUENTA, SALDO_ANTERIOR, SALDO_POSTERIOR,
 IMPORTE, FXMOVIMIENTO)
SELECT
NUMCUENTA, SALDO - @importe, SALDO, @importe, getdate()
FROM CUENTA
WHERE NUMCUENTA = @CuentaDestino

/* Confirmamos la transaccion*/
COMMIT TRANSACTION -- O solo COMMIT

END TRY
BEGIN CATCH
/* Hay un error, deshacemos los cambios*/
ROLLBACK TRANSACTION -- O solo ROLLBACK
PRINT 'Se ha producido un error!'
END CATCH
go
select * from cuenta
go
```

SavePoint y Rollback

Los puntos de recuperación (SavePoints) permiten manejar las transacciones por pasos, pudiendo hacer rollbacks hasta un punto marcado por el savepoint y no por toda la transacción. Ejemplo:

```
BEGIN TRAN
DECLARE @coSucursal varchar(30),
      @ciudadSucursal varchar(30)
      set @coSucursal = 'SAT'
      set @ciudadSucursal = 'Acayucan'
                  UPDATE PRUEBA
                  SET N = @ciudadSucursal
                  WHERE ID = @coSucursal
                  SAVE TRANSACTION T1;
```

SavePoint y Rollback (Continúa Ejemplo)

```
                INSERT INTO PRUEBA
                (ID, N) VALUES
                (@coSucursal, @ciudadSucursal)
      -- ESTE ROLLBACK SOLO AFECTA AL INSERT  QUE ESTA
DESPUES
      -- DEL UPDATE
      ROLLBACK TRANSACTION T1;
-- CONFIRMA LA TRANSACCION QUE ESTA EL EL SAVEPOINT
COMMIT
```

SavePoint y Rollback

Ejemplo Completo de savepoint

```
BEGIN TRAN
DECLARE @coSucursal varchar(30),
            @ciudadSucursal varchar(30)
            set @coSucursal = 'SAT'
            set @ciudadSucursal = 'Acayucan'

                                    UPDATE PRUEBA
                                    SET N = @ciudadSucursal
                                    WHERE ID = @coSucursal
                                    SAVE TRANSACTION T1;


                                    INSERT INTO PRUEBA
                                    (ID, N) VALUES
                                    (@coSucursal, @ciudadSucursal)
            -- ESTE ROLLBACK SOLO AFECTA AL INSERT  QUE ESTA DESPUES
            -- DEL UPDATE
            ROLLBACK TRANSACTION T1;
-- CONFIRMA LA TRANSACCION QUE ESTA EL EL SAVEPOINT
COMMIT
```

Funciones y Procedimientos

Funciones del Sistema

SQL server viene con una amplia gama de funciones que se pueden utilizar para realizar muchas operaciones. Las funciones integradas se pueden dividir en 15 categorías diferentes, como son:

- Agregado
- Configuración
- Criptográficas
- Cursor
- Date and Time
- Management
- Mathematical
- Metadata
- Ranking
- Rowset
- Security
- String
- System
- System statistics
- Text and image

Creación de una Funciones

Aunque las funciones se utilizan para realizar cálculos, una función no puede cambiar el estado de una base de datos o instancia. Las funciones no pueden:

- Realizar una acción que cambie el estado de una instancia o base de datos.
- Modificar datos en una tabla.
- Llamar a una función que tiene un efecto externo, tal como la función RAND ().
- Crear o acceder a tablas temporales.
- Ejecutar código dinámicamente.

Las funciones pueden devolver un valor escalar o una tabla. Con valores de tabla las funciones pueden ser de dos diferentes tipos: en línea (inline) y multi-declaración (multi-statement).

Sintaxis de una Función

```
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ][ type_schema_name. ] parameter_data_type
        [ = default ] [ READONLY ] } [ ,...n ] ])
RETURNS return_data_type
        [ WITH <function_option> [ ,...n ] ]
        [ AS]
BEGIN
                function_body
        RETURN scalar_expression
END
```

Sintaxis de una Función

Una ***función con valores de tabla en línea*** (inline) contiene una única sentencia SELECT que devuelve una tabla. Puesto que una función con valores de tabla en línea no realiza ninguna otra operación, el optimizador la trata igual que una vista.

La sintaxis general de una función con valores de tabla en línea (inline) es la siguiente:

```
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
        [ = default ] [ READONLY ] } [ ,...n ] ])
RETURNS TABLE
        [ WITH <function_option> [ ,...n ] ]
        [ AS ]
        RETURN [ ( ] select_stmt [ ) ][
```

Sintaxis de una Función

La sintaxis general para *una función múlti-sentencia* con valores de tabla es la siguiente:

```
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
        [ = default ] [READONLY] } [ ,...n ] ])
RETURNS @return_variable TABLE <table_type_definition>
        [ WITH <function_option> [ ,...n ] ]
        [ AS ]
BEGIN
        function_body
        RETURN
END
```

Sintaxis de una Función

Sin importar el tipo de función, hay cuatro opciones que se pueden especificar: ENCRYPTION, SCHEMABINDING, RETURNS NULL ON NULL INPUT / CALLED ON NULL INPUT, y EXECUTE AS.

ENCRYPTION, SCHEMABINDING y EXECUTE AS son opciones que también están disponibles para procedimientos almacenados. ENCRYPTION y SCHEMABINDING pueden además usarse para triggers y vistas

Ejemplo Función Escalar(Parte1)

Ejecute el siguiente código:

```
use BANCO
go
drop function numpagos;
go

CREATE FUNCTION numpagos(@NPRESTAMO
AS VARCHAR(4))
RETURNS INT
AS
BEGIN
DECLARE @NPAGO INT
```

Ejemplo Función Escalar(Parte2)

```
SELECT @NPAGO=COUNT(NUMPAGO) FROM
PAGO
GROUP BY NUMPRESTAMO
HAVING NUMPRESTAMO=@NPRESTAMO
ORDER BY NUMPRESTAMO


RETURN @NPAGO
END
GO

select dbo.numpagos('P-17');
go
```

Uso de Procedimientos

Cada declaración que usted ejecute en un servidor SQL se puede encapsular dentro de un procedimiento almacenado. En pocas palabras, un procedimiento almacenado no es más que un lote de T-SQL al que se le ha dado un nombre y se almacenó dentro de una base de datos.

La sintaxis genérica para crear un procedimiento almacenado es la siguiente:

```
CREATE { PROC | PROCEDURE } [schema_name.] procedure_name [ ; number
]
        [ { @parameter [ type_schema_name. ] data_type }
                [ VARYING ] [ = default ] [ OUT | OUTPUT ] [READONLY]
        ] [ ,...n ]
[ WITH <procedure_option> [ ,...n ] ]
[ FOR REPLICATION ]
AS { <sql_statement> [;][ ...n ] | <method_specifier> } [;]
<procedure_option> ::=
        [ ENCRYPTION ] [ RECOMPILE ] [ EXECUTE AS Clause ]
```

Crear Procedimientos

Lo que diferencia a un procedimiento almacenado de un simple lote de T-SQL son todas las estructuras de código que pueden ser empleadas, tales como variables, parametrización, manejo de error, y construcciones de flujo de control.

Procedimientos sin parámetros

```
CREATE PROCEDURE up_obtenerClientes
AS
SELECT CID, CNOMBRE FROM CLIENTE;
GO
EXEC up_obtenerClientes;
```

Procedimientos con parámetros

```
CREATE PROCEDURE up_ClientePorCiudad(@ciudad  varchar(50))
AS
SELECT * FROM CLIENTE WHERE CCIUDAD=@ciudad;
GO
EXEC up_ClientePorCiudad 'Peguerinos';
```

Ejercicios

1. Crear una función de tipo tabla que permita obtener la fecha de todos los sábados que hay entre el 16/03/2003 hasta la fecha actual

2. Modifique el ejemplo anterior para que muestre además de la fecha, también el día

Resultados | Mensaj

	Fecha
1	2017-07-22 00:00:00
2	2017-07-15 00:00:00
3	2017-07-08 00:00:00
4	2017-07-01 00:00:00
5	2017-06-24 00:00:00
6	2017-06-17 00:00:00
7	2017-06-10 00:00:00
8	2017-06-03 00:00:00
9	2017-05-27 00:00:00
10	2017-05-20 00:00:00

Resultados

	Fecha
1	22/Jul/2017
2	15/Jul/2017
3	08/Jul/2017
4	01/Jul/2017
5	24/Jun/2017
6	17/Jun/2017
7	10/Jun/2017
8	03/Jun/2017
9	27/May/2017
10	20/May/2017

Ejercicios (Solución 1)

```
use BANCO
go
drop function Sabados;
go
CREATE FUNCTION Sabados(@FechaI smalldatetime)
RETURNS @Tabla TABLE (Fecha smalldatetime)
BEGIN
     declare @i int
     declare @dia varchar(20)
     declare @fecha datetime
     select @i = 0
     while @i > - DateDiff(Day, '16/03/2003', @FechaI)
     begin
          select @dia = DateName(dw, dateadd(d, @i, @FechaI))
          select @fecha = DateAdd(d, @i, @FechaI)
          if @dia in ('Sábado')
          begin
               INSERT INTO @Tabla Select convert(nvarchar, @fecha, 103)
          end
```

Ejercicios (Solución 1)

```
select @i = @i - 1
     end
 RETURN
END

go
select * from dbo.Sabados(GETDATE());
```

Ejercicios (Solución 2)

```
use BANCO
go
drop function Sabados;
go

CREATE FUNCTION Sabados(@FechaI smalldatetime)
RETURNS @Tabla TABLE (Fecha smalldatetime,
            tdia varchar(10))
BEGIN
    declare @i int
    declare @dia varchar(20)
    declare @fecha datetime
    select @i = 0
    while @i > - DateDiff(Day, '16/03/2003', @FechaI)
    begin
        select @dia = DateName(dw, dateadd(d, @i, @FechaI))
```

Ejercicios (Solución 2)

```
select @fecha = DateAdd(d, @i, @FechaI)
        if @dia in ('Sábado')
        begin
            INSERT INTO @Tabla Select convert(nvarchar, @fecha, 103),@dia
        end
        select @i = @i - 1
    end
 RETURN
END

go
select replace(convert(varchar,Fecha,106),' ','/') AS 'Fecha',tdia from
dbo.Sabados(GETDATE());
```

Resultados | Mensajes

	Fecha	tdia
1	22/Jul/2017	Sábado
2	15/Jul/2017	Sábado
3	08/Jul/2017	Sábado
4	01/Jul/2017	Sábado
5	24/Jun/2017	Sábado
6	17/Jun/2017	Sábado
7	10/Jun/2017	Sábado
8	03/Jun/2017	Sábado
9	27/May/2017	Sábado
10	20/May/2017	Sábado

Disparadores y Cursores

Disparador DML

Aunque las funciones y los procedimientos almacenados son objetos independientes, usted no puede ejecutar directamente un trigger. Los desencadenadores DML se crean contra una tabla o una vista, y se definen para un evento específico INSERT, UPDATE, o DELETE. Cuando ejecuta el evento para el que se definió un disparador, SQL Server ejecuta automáticamente el código incluido en el trigger, también conocido como "disparar" el trigger.

```
CREATE TRIGGER [ schema_name . ]trigger_name
ON { table | view }
[ WITH <dml_trigger_option> [ ,...n ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS { sql_statement [ ; ] [ ,...n ] | EXTERNAL NAME <method specifier [ ; ] > }
```

Disparador DML

Cuando el trigger se define como AFTER, el desencadenador se dispara después que la modificación ha pasado todas las restricciones (constraints). Si en una modificación falla una comprobación de restricción, tal como un check, clave principal o clave foránea, el trigger no se ejecuta. Los desencadenadores AFTER sólo se definen para las tablas. Puede definir varios desencadenadores AFTER para la misma acción.

Un disparador definido con la cláusula INSTEAD OF hace que el código del trigger sea ejecutado como un reemplazo para INSERT, UPDATE, o DELETE. Es posible definir un solo desencadenador INSTEAD OF para una acción determinada. A pesar de que los desencadenadores INSTEAD OF se pueden crear sobre tablas y vistas, estos son casi siempre creados sobre vistas.
Independientemente del número de filas que se ven afectadas, un disparador sólo se activa una vez para una acción.

Disparador DML

Como se explica en el tema de la Clausula OUTPUT, "Manipulación de datos" SQL Server tiene un par de tablas denominadas ***inserted*** y ***deleted*** disponibles cuando se ejecutan cambios.

Veamos un ejemplo donde se active un disparador al momento de que se modifique (UPDATE) el valor del campo ACTIVOS de la tabla SUCURSAL, para ello se deberá crear una tabla de histórico de la sucursal (HSUCURSAL) con los campos (HSUCURSAL,HCIUDAD,HACTIVOSA,HACTIVON,HFX) el campo HFX es para guardar la fecha en la que se actualizó dicho valor

Disparador DML Ejemplo

```
USE BANCO
GO
IF OBJECT_ID (N'dbo.HSUCURSAL', N'U') IS NOT NULL
DROP TABLE dbo.HSUCURSAL
GO
CREATE TABLE HSUCURSAL(
HSUCURSAL VARCHAR(30),
HCIUDAD VARCHAR(30),
HACTIVOSA DECIMAL(10,3),
HACTIVOSN DECIMAL(10,3),
HFX DATE
)
GO
IF OBJECT_ID ('tr_movSucursal', 'TR') IS NOT NULL
   DROP TRIGGER tr_movSucursal;
GO
```

Disparador DML Ejemplo

```
CREATE TRIGGER tr_movSucursal ON SUCURSAL AFTER UPDATE
AS
BEGIN
DECLARE @VALORNUEVO DECIMAL(10,3)

        SET NOCOUNT ON;
        IF UPDATE(ACTIVOS)
        BEGIN
        SET @VALORNUEVO = (SELECT ACTIVOS FROM DELETED)
        INSERT INTO HSUCURSAL(HSUCURSAL,HCIUDAD,HACTIVOSA,HACTIVOSN,HFX)
        SELECT NOMBRESUCURSAL,CIUDADSUCURSAL,@VALORNUEVO,ACTIVOS,GETDATE() FROM
INSERTED
        END
END
GO

UPDATE SUCURSAL
SET ACTIVOS = ACTIVOS * 2 WHERE NOMBRESUCURSAL='Becerril'
```

Ejercicios de Disparador DML

Cambie el tipo de dato para que muestre la fecha, horas, minutos, segundos y milésimas de segundo

Solución

```
USE BANCO
GO
IF OBJECT_ID (N'dbo.HSUCURSAL', N'U') IS NOT NULL
DROP TABLE dbo.HSUCURSAL
GO
CREATE TABLE HSUCURSAL(
HSUCURSAL VARCHAR(30),
HCIUDAD VARCHAR(30),
HACTIVOSA DECIMAL(10,3),
HACTIVOSN DECIMAL(10,3),
HFX datetime
)
GO
IF OBJECT_ID ('tr_movSucursal', 'TR') IS NOT NULL
  DROP TRIGGER tr_movSucursal;
GO
CREATE TRIGGER tr_movSucursal ON SUCURSAL AFTER UPDATE
AS
BEGIN
DECLARE @VALORNUEVO DECIMAL(10,3)

                        SET NOCOUNT ON;
                        IF UPDATE(ACTIVOS)
                        BEGIN
                        SET @VALORNUEVO = (SELECT ACTIVOS FROM DELETED)
                        INSERT INTO HSUCURSAL(HSUCURSAL,HCIUDAD,HACTIVOSA,HACTIVOSN,HFX)
                        SELECT NOMBRESUCURSAL,CIUDADSUCURSAL,@VALORNUEVO,ACTIVOS,GETDATE() FROM INSERTED
                        END
END
GO

UPDATE SUCURSAL
SET ACTIVOS = ACTIVOS * 2 WHERE NOMBRESUCURSAL='Becerril'
go
select * from SUCURSAL
go
select * from HSUCURSAL
```

GRACIAS