

Capítulo 9

Lenguaje Transact SQL

Al finalizar el capítulo, el alumno podrá:

- Identificar los conceptos de programación con T-SQL.
- Construir rutinas sencillas según requerimientos puntuales.

Temas:

1. Definición de variables
2. Estructuras de programación
3. Funciones del usuario
4. Manejo de transacciones

1. Definición de variables

Definición de variables

¿Qué es T-SQL?

- Es una extensión del lenguaje SQL creado por Microsoft.
- Añade programación procedural, variables locales, variables globales, funciones de cadenas, fechas y números.
- Además, amplía las funcionalidades de las sentencias DELETE y UPDATE incorporándole nuevas características que permite implementar el concepto de transacciones.

```
graph BT; SetTheory[Set Theory] --> RelationalModel[Relational Model]; PredicateLogic[Predicate Logic] --> RelationalModel; RelationalModel --> SQL[SQL]; SQL --> TSQL[T-SQL]
```

9 - 4

Copyright © Todos los Derechos Reservados - Cibertec Perú S.A.C.

1.1 Variables locales

Una variable local en Transact SQL es un objeto que contiene un valor individual de datos de un tipo específico. Normalmente, las variables se utilizan en lotes y scripts:

- Como contadores, para contar el número de veces que se realiza un bucle o controlar cuántas veces debe ejecutarse.
- Para contener un valor de datos que desea probar mediante una instrucción de flujo.
- Para guardar el valor de un dato que se va a devolver en un código de retorno de un procedimiento almacenado o un valor devuelto en una función.

Las variables se declaran en el cuerpo de un proceso por lotes o procedimiento con la instrucción **DECLARE**, y se les asignan valores con una instrucción **SELECT** o **SET**. Después de la declaración, todas las variables se inicializan en **NULL**, a menos que se proporcione un valor como parte de la declaración.

Los nombres de variable deben comenzar con un signo de arroba (@), seguido del tipo de dato definido por el usuario; una variable no puede ser del tipo de dato **text**, **ntext**, ni **image**.

El siguiente es un ejemplo de declaración de una variable del tipo VARCHAR de 20 caracteres de longitud.

```
DECLARE @Nombre VARCHAR(20)
```

A continuación, presentamos algunos ejemplos que demuestran la funcionalidad básica de la declaración de variables:

a. Usar DECLARE

En el ejemplo siguiente se utiliza una variable local denominada @find para recuperar la información de clientes para todos los contactos que comiencen con An.

```
DECLARE @find varchar(30);

SET @find = 'Ana%';

SELECT CustomerID, CompanyName, ContactName, ContactTitle
FROM Customers
WHERE ContactName LIKE @find
```

b. Usar DECLARE con dos variables

El ejemplo siguiente recupera los clientes que se encuentran en la ciudad de México D.F. y su código postal sea 05033.

```
DECLARE @City varchar(50), @PostalCode varchar(10)

SET @City = 'México D.F.'
SET @PostalCode = '05033'

SELECT CustomerID, CompanyName, ContactName, ContactTitle
FROM Customers
WHERE City = @City
AND PostalCode = @PostalCode
```

1.2 Variables globales

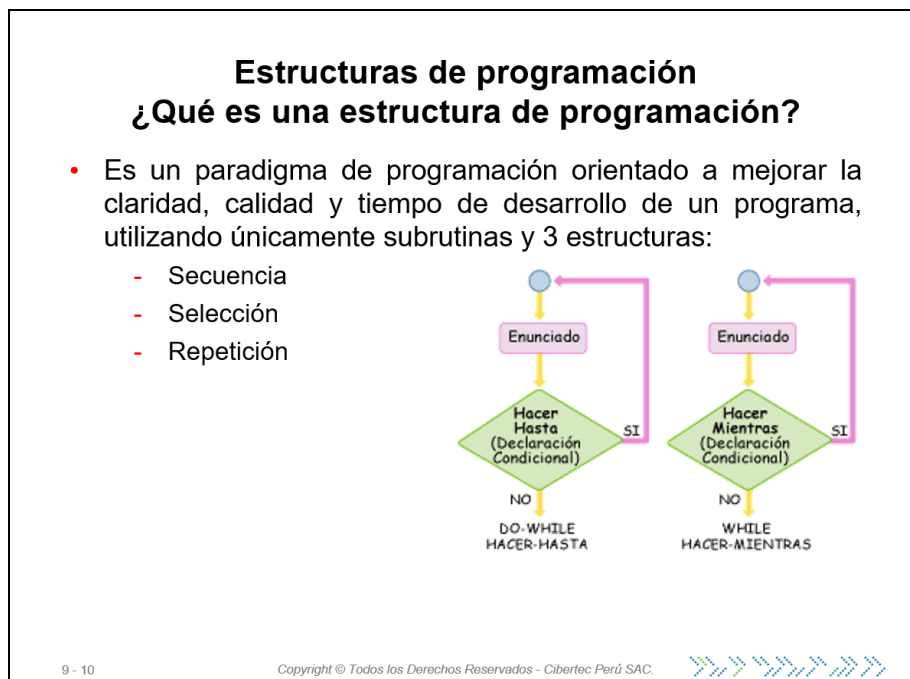
Son variables predefinidas suministradas por el sistema. Se distinguen de las variables locales por tener dos símbolos "@" precediendo a sus nombres; por ejemplo, @@ error.

Variables globales de SQL Server	
Variable	Contenido
@@version	Retorna la versión del SQL Server.
@@language	Retorna el nombre del lenguaje que se está usando.
@@error	Contiene 0 si la última transacción se ejecutó de forma correcta; caso contrario, contiene el último número de error generado por el sistema. Se utiliza para verificar el estado de error de la última instrucción emitida, se haya ejecutado correctamente o no. Una instrucción como if @@error != 0 seguida de return, origina una salida por error.
@@identity	Contiene el último valor insertado en una columna IDENTITY mediante una instrucción INSERT o SELECT INTO. Además, se define cada vez que se inserta una fila en una tabla. Si una instrucción inserta múltiples filas, @@ identity refleja el valor IDENTITY de la última fila insertada. Si la tabla afectada no contiene una columna IDENTITY, se define en 0. El valor de @@identity no se ve afectado por el fallo de una instrucción insert o select into, ni por la reversión de la

	transacción que la contenía, pues conserva el último valor insertado en una columna IDENTITY, aunque la instrucción que la haya insertado no se consigne.
@@rowcount	Devuelve el número de filas afectadas por la última instrucción.

Si un usuario declara una variable local que tiene el mismo nombre que una variable global, dicha variable se trata como una variable local.

2. Estructuras de programación



2.1 Conceptos previos

Transact-SQL permite agrupar una serie de instrucciones como un lote, ya sea de forma interactiva o desde un archivo del sistema operativo. También, se pueden utilizar las estructuras de control de flujo para conectar las instrucciones utilizando estructuras del tipo de programación.

Además, proporciona palabras clave especiales, llamadas lenguaje de control de flujo, que permiten controlar el flujo de ejecución de las instrucciones. El lenguaje de control de flujo se puede utilizar en instrucciones sencillas, lotes, procedimientos almacenados y disparadores. Sin este lenguaje, las instrucciones SQL se llevan a cabo de forma secuencial, conforme se producen.

Las subconsultas correlacionadas son una excepción parcial. El lenguaje de control de flujo permite que las instrucciones se conecten y se relacionen entre sí, utilizando estructuras de tipo de programación.

El lenguaje de control de flujo (if...else para la ejecución condicional de comandos y while para la ejecución repetitiva), permite refinar y controlar el funcionamiento de las instrucciones SQL. El lenguaje de control de flujo de Transact-SQL, transforma el SQL estándar en un lenguaje de programación de muy alto nivel.

Reglas asociadas a lotes

Existen reglas que controlan qué instrucciones SQL pueden combinarse en un solo lote. Estas reglas de lotes incluyen lo siguiente:

- Algunos comandos de base de datos no pueden combinarse con otras instrucciones en un lote. Se trata de los siguientes: create procedure, create rule, create default, create trigger, create view.
- Los comandos que sí pueden combinarse con otras instrucciones SQL en un lote incluyen: create database (salvo que no puede crear una base de datos y generar o acceder a los objetos de la base de datos nueva en un solo lote) create table, create index.
- Las reglas y valores predeterminados no pueden vincularse a columnas y utilizarse durante el mismo lote. **sp_bindrule** y **sp_bindefault** no pueden estar en el mismo lote que las instrucciones **insert** que ejecutan la regla o valor predeterminado.
- **USE** debe ejecutarse en un lote anterior antes que las instrucciones que hacen referencia a los objetos de dicha base de datos.
- No es posible realizar una operación **DROP** con un objeto y después hacer referencia a este o volver a crearlo en el mismo lote.
- Cualquier opción definida con una instrucción **set** tendrá efecto al final del lote. Es posible combinar instrucciones set y consultas en el mismo lote, pero las opciones de set no se aplicarán a las consultas de dicho lote.

Uso del lenguaje de control de flujo

El lenguaje de control de flujo se puede utilizar con instrucciones interactivas, en lotes y en procedimientos almacenados. El control de flujo y las palabras clave relacionadas y sus funciones son:

Control de flujo y palabras clave relacionadas	
Palabra clave	Función
BEGIN	Comienzo de un bloque de instrucciones.
...END	Final de un bloque de instrucciones.
IF	Define una ejecución condicional.
...ELSE	Define una ejecución alternativa cuando la condición if es falsa.
WHILE	Repite la ejecución de instrucciones mientras la condición es verdadera.
BREAK	Salida del final del siguiente bucle while más exterior.
...CONTINUE	Reinicio del bucle while.
DECLARE	Declara variables locales.
GOTO LABEL	Va a un rótulo (label:), una posición en un bloque de instrucciones.
RETURN	Salida de forma incondicional.
WAITFOR	Define el retardo para la ejecución del comando.

PRINT	Imprime un mensaje definido por el usuario o una variable local en la pantalla del usuario.
RAISERROR	Imprime un mensaje definido por el usuario o una variable local en la pantalla del usuario y define un indicador del sistema en la variable global @@ error.
/*COMENTARIO */	Inserta un comentario en cualquier punto de una instrucción SQL.
CASE	Permite que se muestre un valor alternativo.

2.2 Estructura BEGIN - END

Las palabras clave BEGIN y END se utilizan para englobar una serie de instrucciones, a fin de que sean tratadas como una unidad por las estructuras de control de flujo como if...else. Una serie de instrucciones englobadas por BEGIN y END se denomina **bloque de instrucciones**.

La sintaxis es:

```
BEGIN
    BLOQUE DE INSTRUCCIONES
END
```

En el siguiente ejemplo, **BEGIN** y **END** definen un conjunto de instrucciones Transact SQL que se ejecutan juntas. Si el bloque **BEGIN...END** no se incluye, se devolverán ambos mensajes del **PRINT**.

```
IF @@TRANCOUNT = 0
BEGIN
    SELECT LastName, FirstName
    FROM Employees WHERE LastName = 'Fuller'

    PRINT 'Rolling back the transaction two times would cause an error.'
END

PRINT 'Rolled back the transaction.'
```

2.3 Estructura IF... ELSE

La palabra clave IF, con o sin la compañía de ELSE, se utiliza para introducir una condición que determina si se ejecutará la instrucción siguiente. La instrucción SQL se ejecuta si la condición se cumple, es decir, si devuelve TRUE (verdadero).

La palabra clave ELSE introduce una instrucción SQL alternativa que se ejecuta cuando la condición IF devuelve FALSE (falso).

La sintaxis para ambas es:

```
IF EXPRESION_BOLEANA
  EXPRESIÓN _SQL
[ELSE
  [ IF EXPRESION_BOLEANA ]
    EXPRESIÓN _SQL ]
```

Una expresión booleana es una expresión que devuelve TRUE o FALSE. En ella se puede incluir un nombre de columna, una constante, cualquier combinación de nombres de columna y constantes conectados por operadores aritméticos, o basados en bits o una subconsulta, siempre que esta devuelva un solo valor. Si la expresión booleana contiene una instrucción SELECT, debe incluirse entre paréntesis y devolver un único valor.

A continuación, se muestra un ejemplo del uso de IF:

```
IF (SELECT COUNT(*) FROM Products
    WHERE ProductName LIKE 'Sir Rodney%') > 1
    PRINT 'There are more than 1 presentation Sir Rodney's.'
```

En el ejemplo se ejecuta una consulta en la expresión booleana. Dado que hay 2 presentaciones del producto Sir Rodney's en la tabla **Products** que cumplen con la cláusula **WHERE** se ejecuta el **PRINT**.

En el ejemplo siguiente hay una expresión booleana (1=1) que es TRUE y, por lo tanto, se imprime la primera instrucción.

```
IF 1 = 1
    PRINT 'Boolean_expression is true.'
ELSE
    PRINT 'Boolean_expression is false.'
```

En el ejemplo siguiente hay una expresión booleana simple (1=2) que es FALSE y, por tanto, se imprime la segunda instrucción.

```
IF 1 = 2
    PRINT 'Boolean_expression is true.'
ELSE
    PRINT 'Boolean_expression is false.'
```

Las estructuras **IF...ELSE** se utilizan frecuentemente, en procedimientos almacenados en los que se verifica la existencia de algún parámetro.

Las pruebas **IF** pueden estar anidadas dentro de otras pruebas **IF**, ya sea dentro de otra construcción **IF** o después de una **ELSE**. La expresión de la prueba **IF** solo puede devolver un valor. Además, para cada estructura **IF...ELSE**, puede existir una instrucción **SELECT** para **IF** y otra para **ELSE**. Para incluir más de una instrucción **SELECT**, hay que utilizar las palabras clave **BEGIN...END**. El número máximo de pruebas **IF** que pueden anidarse varía con la complejidad de las instrucciones **SELECT** (u otras estructuras de lenguaje) que se incluyan con cada estructura **IF...ELSE**.

En el ejemplo siguiente se muestra cómo se puede anidar una instrucción **IF...ELSE** dentro de otra. Establezca la variable @Number en 5, 50 y 500 para probar cada instrucción.


```
DECLARE @Number int;
SET @Number = 50;
IF @Number > 100
    PRINT 'The number is large.';
ELSE
    BEGIN
        IF @Number < 10
            PRINT 'The number is small.';
        ELSE
            PRINT 'The number is medium.';
    END
```

2.4 Estructura WHILE Y BREAK...CONTINUE

WHILE se utiliza para definir una condición para la ejecución repetida de una instrucción o un bloque de instrucciones. Las instrucciones se ejecutan reiteradamente, siempre que la condición especificada sea verdadera.

La sintaxis se muestra a continuación.

```
WHILE BOOLEAN_EXPRESSION
    EXPRESIÓN _SQL
```

En el ejemplo siguiente, si el precio de venta promedio de un producto es inferior a \$300, el bucle WHILE dobla los precios y, a continuación, selecciona el precio máximo. Este bucle continúa doblando los precios hasta que el promedio sea menor que \$300, después de lo cual sale del bucle WHILE e imprime un mensaje.

```
WHILE (SELECT AVG(UnitPrice) FROM Products) < 100
BEGIN
    UPDATE Products
        SET UnitPrice = UnitPrice * 2

    SELECT MAX(UnitPrice) FROM Products
END
PRINT 'Too much for the market to bear'
```

BREAK y CONTINUE controlan el funcionamiento de las instrucciones dentro de un bucle while. BREAK permite salir del bucle while. Todas las instrucciones que aparecen después de la palabra clave END, que marca el final del bucle, se ejecutan.

CONTINUE hace que el bucle WHILE se inicie de nuevo, omitiendo cualquier instrucción después de CONTINUE, menos dentro del bucle. BREAK y CONTINUE se activan frecuentemente mediante una prueba IF.

La sintaxis de break...continue es la siguiente.

```
WHILE BOOLEAN EXPRESSION
BEGIN
    EXPRESIÓN _SQL
    [EXPRESIÓN _SQL]...
    BREAK
    [EXPRESIÓN _SQL]...
    CONTINUE
    [EXPRESIÓN _SQL]...
END
```

En el ejemplo siguiente, si el precio de venta promedio de un producto es inferior a 300, el bucle WHILE dobla los precios y, a continuación, selecciona el precio máximo. Si el precio máximo es menor o igual que 500, el bucle WHILE se reinicia y vuelve a doblar los precios. Este bucle continúa doblando los precios hasta que el precio máximo es mayor que 500, después de lo cual sale del bucle WHILE e imprime un mensaje.

```
WHILE (SELECT AVG(UnitPrice) FROM Products) < 300
BEGIN
    UPDATE Products
    SET UnitPrice = UnitPrice * 2

    SELECT MAX(UnitPrice) FROM Products

    IF (SELECT MAX(UnitPrice) FROM Products) > 500
        BREAK
    ELSE
        CONTINUE
END

PRINT 'Too much for the market to bear'
```

2.5 **GOTO**

Altera el flujo de ejecución y lo dirige a una etiqueta. Las instrucciones Transact-SQL que siguen a una instrucción GOTO se pasan por alto y el procesamiento continúa en el punto que marca la etiqueta. Las instrucciones GOTO y las etiquetas se pueden utilizar en cualquier punto de un procedimiento, lote o bloque de instrucciones. Las instrucciones GOTO se pueden anidar.

A continuación, se muestra la sintaxis.

```
LABEL:
    GOTO LABEL
```

En el ejemplo siguiente se muestra cómo usar **GOTO** como mecanismo de bifurcación.

```
DECLARE @Counter int
SET @Counter = 1

WHILE @Counter < 10
BEGIN
    SELECT @Counter
    SET @Counter = @Counter + 1
    GOTO LABEL
```

```
IF @Counter = 4
    GOTO Branch_One --Jumps to the first branch.

IF @Counter = 5
    GOTO Branch_Two --This will never execute.
END

Branch_One:
    SELECT 'Jumping To Branch One.'
    GOTO Branch_Three --This will prevent Branch_Two from executing.
Branch_Two:
    SELECT 'Jumping To Branch Two.'
Branch_Three:
    SELECT 'Jumping To Branch Three.'
```

2.6 RETURN

Salte incondicionalmente de una consulta o procedimiento. RETURN es inmediata y completa, y se puede utilizar en cualquier punto para salir de un procedimiento, lote o bloque de instrucciones. Las instrucciones que siguen a RETURN no se ejecutan.

La sintaxis es la que se muestra líneas abajo.

```
RETURN [EXPRESION ENTERA]
```

En el siguiente ejemplo se muestra que si no se proporciona ningún nombre de usuario como parámetro al ejecutar findjobs, RETURN provoca la salida del procedimiento tras enviar un mensaje a la pantalla del usuario. Si se especifica un nombre de usuario, se obtienen de las tablas del sistema adecuadas, los nombres de todos los objetos creados por este usuario en la base de datos actual.

```
CREATE PROCEDURE findjobs @nm sysname = NULL
AS
    IF @nm IS NULL
    BEGIN
        PRINT 'You must give a user name'
        RETURN
    END
    ELSE
    BEGIN
        SELECT o.name, o.id, o.uid
        FROM sysobjects o
        INNER JOIN master..syslogins l ON o.uid = l.sid
        WHERE l.name = @nm
    END
```

En el siguiente ejemplo se comprueba el país del Id. de un proveedor especificado. Si el país es USA, se devuelve un estado de 1. En caso contrario, se devuelve 2 para cualquier otra condición (un valor distinto de USA para Country o SupplierID que no coincida con una fila).

```
CREATE OR ALTER PROCEDURE checkstate @param int
AS
    IF (SELECT Country FROM Suppliers
        WHERE SupplierID = @param) = 'USA'
        RETURN 1
    ELSE
        RETURN 2
```

2.7 Manejo de excepciones TRY...CATCH

Implementa un mecanismo de control de errores para Transact-SQL, que es similar al control de excepciones en los lenguajes Microsoft Visual C# y Microsoft Visual C++. Se puede incluir un grupo de instrucciones Transact-SQL en un bloque TRY. Si se produce un error en el bloque TRY, el control se transfiere a otro grupo de instrucciones que está incluido en un bloque CATCH.

La sintaxis es la que se muestra líneas abajo.

```
BEGIN TRY
    {expresión SQL}
END TRY
BEGIN CATCH
    [{expresión SQL}]
END CATCH
```

Una construcción TRY...CATCH detecta todos los errores de ejecución que tienen una gravedad mayor de 10 y que no cierran la conexión de la base de datos.

Un bloque TRY debe ir seguido inmediatamente por un bloque CATCH asociado. Si se incluye cualquier otra instrucción entre las instrucciones END TRY y BEGIN CATCH se genera un error de sintaxis.

Una construcción TRY...CATCH no puede abarcar varios bloques de instrucciones Transact-SQL. Por ejemplo, una construcción TRY...CATCH no puede abarcar dos bloques BEGIN...END de instrucciones Transact-SQL y no puede abarcar una construcción IF...ELSE.

Si no hay errores en el código incluido en un bloque TRY, cuando la última instrucción de este bloque ha terminado de ejecutarse, el control se transfiere a la instrucción inmediatamente posterior a la instrucción END CATCH asociada. Si hay un error en el código incluido en un bloque TRY, el control se transfiere a la primera instrucción del bloque CATCH asociado. Si la instrucción END CATCH es la última instrucción de un procedimiento almacenado o desencadenador, el control se devuelve a la instrucción que llamó al procedimiento almacenado o activó el desencadenador.

Cuando finaliza el código del bloque CATCH, el control se transfiere a la instrucción inmediatamente posterior a la instrucción END CATCH. Los errores capturados por un bloque CATCH no se devuelven a la aplicación que realiza la llamada. Si es necesario devolver cualquier parte de la información sobre el error a la aplicación, debe hacerlo el código del bloque CATCH a través de mecanismos como los conjuntos de resultados SELECT o las instrucciones RAISERROR y PRINT.

Las construcciones TRY...CATCH pueden estar anidadas. Un bloque TRY o un bloque CATCH puede contener construcciones TRY...CATCH anidadas. Por ejemplo, un bloque CATCH puede contener una construcción TRY...CATCH incrustada para controlar los errores detectados por el código de CATCH.

Los errores que se encuentren en un bloque CATCH se tratan como los errores generados en otros lugares. Si el bloque CATCH contiene una construcción TRY...CATCH anidada, los errores del bloque TRY anidado transferirán el control al bloque CATCH anidado. Si no hay ninguna construcción TRY...CATCH anidada, el error se devuelve al autor de la llamada.

Las construcciones TRY...CATCH capturan los errores no controlados de los procedimientos almacenados o desencadenadores ejecutados por el código del bloque TRY. Alternativamente, los procedimientos almacenados o desencadenadores pueden contener sus propias construcciones TRY...CATCH para controlar los errores generados por su código. Por ejemplo, cuando un bloque TRY ejecuta un procedimiento almacenado y se produce un error en este, el error se puede controlar de las formas siguientes:

- Si el procedimiento almacenado no contiene su propia construcción TRY...CATCH, el error devuelve el control al bloque CATCH asociado al bloque TRY que contiene la instrucción EXECUTE.
- Si el procedimiento almacenado contiene una construcción TRY...CATCH, el error transfiere el control al bloque CATCH del procedimiento almacenado. Cuando finaliza el código del bloque CATCH, el control se devuelve a la instrucción inmediatamente posterior a la instrucción EXECUTE que llamó al procedimiento almacenado.

No se pueden utilizar instrucciones GOTO para entrar en un bloque TRY o CATCH. Estas instrucciones se pueden utilizar para saltar a una etiqueta dentro del mismo bloque TRY o CATCH, o bien para salir de un bloque TRY o CATCH.

La construcción TRY...CATCH no se puede utilizar en una función definida por el usuario.

En el ámbito de un bloque CATCH, se pueden utilizar las siguientes funciones del sistema para obtener información acerca del error que provocó la ejecución del bloque CATCH:

- ERROR_NUMBER() devuelve el número del error.
- ERROR_SEVERITY() devuelve la gravedad.
- ERROR_STATE() devuelve el número de estado del error.
- ERROR_PROCEDURE() devuelve el nombre del procedimiento almacenado o desencadenador donde se produjo el error.
- ERROR_LINE() devuelve el número de línea de la rutina que provocó el error.
- ERROR_MESSAGE() devuelve el texto completo del mensaje de error. Este texto incluye los valores suministrados para los parámetros reemplazables, como longitudes, nombres de objetos u horas.

Por ejemplo, en el siguiente script se muestra un procedimiento almacenado que contiene funciones de control de errores. Se llama al procedimiento almacenado en el bloque CATCH de una construcción TRY...CATCH y se devuelve información sobre el error.

```
-- Create procedure to retrieve error information.
CREATE PROCEDURE usp_GetErrorInfo
AS
    SELECT ERROR_NUMBER() AS ErrorNumber,
           ERROR_SEVERITY() AS ErrorSeverity,
           ERROR_STATE() AS ErrorState,
```

```
        ERROR_PROCEDURE() AS ErrorProcedure,  
        ERROR_LINE() AS ErrorLine,  
        ERROR_MESSAGE() AS ErrorMessage  
GO  
  
BEGIN TRY  
    -- Generate divide-by-zero error.  
    SELECT 1/0;  
END TRY  
  
BEGIN CATCH  
    -- Execute error retrieval routine.  
    EXECUTE usp_GetErrorInfo;  
END CATCH
```

2.8 **WAITFOR**

Bloquea la ejecución de un lote, un procedimiento almacenado o una transacción hasta alcanzar la hora o el intervalo de tiempo especificado, o hasta que una instrucción especificada modifique o devuelva al menos una fila.

```
WAITFOR  
{  
    DELAY 'time_to_pass'  
    | TIME 'time_to_execute'  
    | [ ( receive_statement ) | ( get_conversation_group_statement ) ]  
    [ , TIMEOUT timeout ]  
}
```

La transacción se está ejecutando mientras se ejecuta la instrucción WAITFOR y no se puede ejecutar ninguna otra solicitud para la misma transacción.

El retardo de tiempo real puede variar del tiempo especificado en time_to_pass, time_to_execute o timeout, y depende del nivel de actividad del servidor. El contador de tiempo se inicia cuando se programa el subproceso asociado a la instrucción WAITFOR. Si el servidor está ocupado, es posible que no se programe el subproceso inmediatamente; por tanto, el retardo de tiempo puede ser mayor que el tiempo especificado.

WAITFOR no cambia la semántica de una consulta. Si una consulta no devuelve ninguna fila, WAITFOR esperará indefinidamente o hasta que se alcance el valor de TIMEOUT, si está especificado.

No se pueden abrir cursores ni vistas en las instrucciones WAITFOR.

Cuando una consulta supera la opción query wait, se puede completar el argumento de la instrucción WAITFOR sin ejecutarse.

Cada instrucción WAITFOR tiene un subproceso asociado. Si se especifica un gran número de instrucciones WAITFOR en el mismo servidor, se pueden acumular muchos subprocesos a la espera de que se ejecuten estas instrucciones. SQL Server supervisa el número de subprocesos asociados con las instrucciones WAITFOR y selecciona aleatoriamente algunos de estos subprocesos para salir si el servidor empieza a experimentar la falta de subprocesos.

Puede crear un interbloqueo ejecutando una consulta con WAITFOR en una transacción que también tenga bloqueos que impidan realizar cambios en el conjunto de filas al que intenta tener acceso la instrucción WAITFOR. SQL Server identifica estos escenarios y devuelve un conjunto de resultados vacío si existe la posibilidad de un interbloqueo de este tipo.

En el ejemplo siguiente se ejecuta el procedimiento almacenado `sp_update_job` en la base de datos `msdb` a las 10:20 p.m. (22:20).

```
EXECUTE sp_add_job @job_name = 'TestJob';
BEGIN
    WAITFOR TIME '22:20';
    EXECUTE sp_update_job @job_name = 'TestJob', @new_name = 'UpdatedJob';
END
```

En el ejemplo siguiente se ejecuta el procedimiento almacenado después de un retardo de dos horas.

```
BEGIN
    WAITFOR DELAY '02:00';
    EXECUTE sp_helpdb;
END
```

2.9 Expresión CASE

Es una expresión especial de Transact-SQL que permite mostrar un valor alternativo dependiendo del valor de una columna. Este cambio es temporal, con lo que no hay cambios permanentes en los datos.

Por ejemplo, la función CASE puede mostrar 'Eastern' o 'Western' o 'Northern' o 'Southern' dependiendo de la columna `RegionID` de la tabla `Territories`.

La función CASE está compuesta de:

- La palabra clave **CASE**.
- El nombre de columna que se va a transformar.
- Cláusulas **WHEN** que especifican las expresiones que se van a buscar y cláusulas **THEN** que especifican las expresiones que las van a reemplazar.
- La palabra clave **END**.
- Una cláusula **AS** opcional que define un alias de la función CASE.

En este ejemplo se utiliza la expresión **CASE** para cambiar la presentación de la región de los territorios con el fin de hacerla más comprensible.

```
SELECT TerritoryID, TerritoryDescription,
       CASE RegionID WHEN 1 THEN 'Eastern'
                     WHEN 2 THEN 'Western'
                     WHEN 3 THEN 'Northern'
                     WHEN 4 THEN 'Southern'
                     ELSE 'No Region' END
FROM Territories
ORDER BY TerritoryID
```

2.10 Función IFF

La función IFF es una forma abreviada de escribir una expresión CASE. Evalúa la expresión booleana pasada como primer argumento, y luego devuelve cualquiera de los otros dos argumentos en función del resultado de la evaluación. Es decir, se

devuelve el *true_value* si la expresión booleana es verdadera y se devuelve *false_value* si la expresión booleana es falsa o desconocida.

```
IIF (boolean_expression, true_value, false_value)
```

Por ejemplo, se tiene esta sentencia con una expresión CASE:

```
DECLARE @a int = 45, @b int = 40
SELECT CASE WHEN @a > @b THEN 'TRUE' ELSE 'FALSE' END AS Result
```

Se puede sustituir por esta versión más clara y compacta:

```
DECLARE @a int = 45, @b int = 40
SELECT IIF ( @a > @b, 'TRUE', 'FALSE' ) AS Result
```

2.11 Función CHOOSE

La función CHOOSE actúa como un índice en una matriz, en donde la matriz se compone de los argumentos que siguen al argumento del índice. El argumento del índice determina cuál de los siguientes valores será devuelto.

```
CHOOSE ( index, val_1, val_2 [, val_n ] )
```

Por ejemplo, se tiene esta sentencia con una expresión CASE:

```
SELECT CASE DATEPART(month, getdate())
    WHEN 1 THEN 'Ene'
    WHEN 2 THEN 'Feb'
    WHEN 3 THEN 'Mar'
    WHEN 4 THEN 'Abr'
    WHEN 5 THEN 'May'
    WHEN 6 THEN 'Jun'
    WHEN 7 THEN 'Jul'
    WHEN 8 THEN 'Ago'
    WHEN 9 THEN 'Sep'
    WHEN 10 THEN 'Oct'
    WHEN 11 THEN 'Nov'
    WHEN 12 THEN 'Dic'
    ELSE '' END as x_mes
```


Se puede sustituir por esta versión más clara y compacta, porque puede simplificar sentencias CASE mucho más complejas:

```
SELECT CHOOSE(DATEPART(month, getdate()),
    'Ene', 'Feb', 'Mar', 'Abr', 'May', 'Jun', 'Jul', 'Ago', 'Sep', 'Oct', 'Nov', 'Dic')
```


3. Funciones del usuario

Funciones del usuario
¿Qué es una función de usuario?

- Permite al usuario crear sus propias funciones.
- El T-SQL soluciona problemas de reutilización de código.
- La lógica de la función es almacenada en el servidor.



9 - 20 Copyright © Todos los Derechos Reservados - Cibertec Perú S.A.C.

3.1 Introducción

Microsoft agregó nuevas características a su producto SQL Server desde la versión 2000 y lo más interesante para los programadores es la posibilidad de hacer funciones definidas por el usuario. La adición de funciones al lenguaje de SQL, solucionará los problemas de reutilización del código y dará mayor flexibilidad al programar las consultas de SQL.

Limitaciones

Las funciones definidas por el usuario tienen algunas restricciones. No todas las sentencias SQL son válidas dentro de una función. Las listas siguientes enumeran las operaciones válidas e inválidas de las funciones.

Válido:

- Las sentencias de asignación.
- Las sentencias de control de flujo.
- Sentencias SELECT y modificación de variables locales
- Operaciones de cursores, sobre variables locales
- Sentencias INSERT, UPDATE, DELETE con variables locales.

Inválido:

- Armar funciones no determinadas como GetDate().
- Sentencias de modificación o actualización de tablas o vistas
- Operaciones CURSOR FETCH que devuelven datos del cliente.

3.2 Funciones escalares

Las funciones escalares vuelven un tipo de los datos, tal como int, money, varchar, real, etc. Pueden ser utilizadas en cualquier lugar, incluso incorporadas dentro de sentencias SQL. La sintaxis para una función escalar es como se muestra a continuación, acompañada de 2 ejemplos: una función que devuelve el cubo de un número y otra función que devuelve la factorial de un número.

Seguidamente, se muestra la sintaxis para una función escalar:

```
CREATE FUNCTION [owner_name.] function_name
( [{ @parameter_name scalar_parameter_type [ = default]] [,..n] })
RETURNS scalar_return_type
[WITH <function_option> >::={SCHEMABINDING | ENCRYPTION}]
[AS]
BEGIN
    CUERPO DE LA FUNCIÓN
    RETURN Valor Escalar
END
```

A continuación, presentamos algunos ejemplos que demuestran la funcionalidad básica del uso de funciones escalares.

En el ejemplo siguiente se crea una función escalar, en donde la función toma un valor de entrada fNumber y devuelve el cubo.

```
/* Cubo de un número */
CREATE FUNCTION dbo.Cubo( @fNumber float)
RETURNS float
AS
BEGIN
    RETURN(@fNumber * @fNumber * @fNumber)
END
```

En el ejemplo siguiente se crea una función escalar, en donde la función toma un valor de entrada iNumber y devuelve el factorial.

```
/* Factorial de un número */
CREATE FUNCTION dbo.Factorial ( @iNumber int )
RETURNS INT
AS
BEGIN
    DECLARE @i int
    IF @iNumber <= 1
        SET @i = 1
    ELSE
        SET @i = @iNumber * dbo.Factorial( @iNumber - 1 )
    RETURN (@i)
END
```

En el ejemplo se crea una función escalar, en donde la función toma un valor de entrada, ProductID, y devuelve las unidades de stock.

```
CREATE FUNCTION dbo.ufnGetUnitsInStock (@ProductID int)
RETURNS int
AS
    -- Returns the stock level for the product
BEGIN
    DECLARE @ret    int

    SELECT @ret = UnitsInStock
    FROM Products p
    WHERE p.ProductID = @ProductID

    IF @ret IS NULL
        SET @ret = 0

    RETURN @ret
END
```

3.3 Funciones de tabla en línea

Son funciones que devuelven la salida de una simple declaración SELECT; esta salida se puede utilizar dentro de JOINS o QUERYS, como si fuera una tabla estándar.

Seguidamente, se muestra la sintaxis para una función de tabla en línea.

```
CREATE FUNCTION [owner_name.] function_name
( [{ @parameter_name scalar_parameter_type [ = default]} [,..n]])
RETURNS TABLE
[WITH <function_option>::={SCHEMABINDING | ENCRYPTION}]
RETURN [(] select_statement [)]
```

En el ejemplo siguiente se crea una función insertada con valores de tabla. La función toma como parámetros de entrada, Id de Cliente (almacén) y devuelve las columnas ProductID, Name, y el agregado de las ventas del año hasta la fecha como total por cada producto vendido en el almacén.

```
CREATE FUNCTION dbo.ufn_SalesByCustomer (@CustomerID char(05))
RETURNS TABLE
AS
RETURN
(
    SELECT P.ProductID, P.ProductName, (OD.Quantity * OD.UnitPrice) as Total
    FROM Orders O
    INNER JOIN OrderDetails OD ON O.OrderID = OD.OrderID
    INNER JOIN Products P ON P.ProductID = OD.ProductID
    WHERE O.CustomerID = @CustomerID
    GROUP BY P.ProductID, P.ProductName)
)
```

3.4 Funciones de tabla de multi sentencias

Son similares a los procedimientos almacenados, excepto que devuelven una tabla. Este tipo de función se usa en situaciones donde se requiere más lógica y proceso. Lo que sigue es la sintaxis para unas funciones de tabla de multi sentencias.

```

CREATE FUNCTION [owner_name.] function_name
( [{ @parameter_name scalar_parameter_type [ = default]] [,..n]])
RETURNS TABLE
[WITH <function_option> >::={SCHEMABINDING | ENCRYPTION}]
[AS]
BEGIN
    function_body
RETURN
END

```

En el ejemplo siguiente se crea una función de tabla ufn_FindReports (InEmpID) en donde se suministra un indicador de empleado válido, la función devuelve una tabla de todos los empleados que están bajo las órdenes de ese empleado tanto directa como indirectamente.

```

CREATE FUNCTION dbo.ufn_ReportEmployee (@Country varchar(15))
RETURNS @retReportEmployee TABLE
(
    EmployeeID        int NOT NULL,
    TitleOfCourtesy   varchar(25) NOT NULL,
    FirstName          varchar(10) NOT NULL,
    LastName           varchar(20) NOT NULL,
    Title              varchar(30) NOT NULL,
    ReportsTo          varchar(50) NULL
)
AS
BEGIN
    INSERT INTO @retReportEmployee
    SELECT E1.EmployeeID,
           E1.TitleOfCourtesy,
           E1.FirstName,
           E1.LastName,
           E1.Title,
           ISNULL(E2.TitleOfCourtesy + ' '+E2.FirstName+' '+E2.LastName, '')
           as ReportsTo
    FROM Employees E1
    LEFT JOIN Employees E2 ON E2.EmployeeID = E1.ReportsTo
    WHERE E1.Country = @Country
    ORDER BY ReportsTo ASC

    RETURN
END

```

Recuerde que...



Llamando a funciones

La diferencia substancial de las funciones frente a los procedimientos almacenados es que pueden ser invocados dentro de una sentencia select:

```
select * from funcion(parametros_de_la_funcion)
```

Por tanto, en una simple sentencia se puede devolver un conjunto de registros, obtenidos a partir de una serie de sentencias con más o menos complejidad.


Hay algunas idiosincrasias de la sintaxis al invocar funciones definidas por el usuario.

SQL Server proporciona algunas funciones definidas por el usuario a nivel sistema en la base de datos Master. Estas funciones del sistema se invocan con una sintaxis levemente distinta a las que usted puede crear.

4. Manejo de transacciones


Manejo de transacciones

¿Qué es una transacción?



- Conjunto de operaciones ejecutadas como una única unidad y puede contener varias instrucciones.
- Cumplen 4 propiedades conocidas como ACID:
 - Atomicidad
 - Coherencia
 - Asilamiento
 - Durabilidad
- Al enviar el T-SQL al servidor, éste lo escribe en el fichero de transacciones y realiza los cambios solicitados.
- Ante un problema, el servidor de base de datos puede leer este fichero de transacciones y deshacer los cambios.

[https://technet.microsoft.com/es-es/library/ms191165\(v=sql.105\).aspx](https://technet.microsoft.com/es-es/library/ms191165(v=sql.105).aspx)

9 - 24 Copyright © Todos los Derechos Reservados - Cibertec Perú S.A.C. 

4.1 Definición

Una transacción es un conjunto de operaciones que van a ser tratadas como una unidad. Estas transacciones deben cumplir 4 propiedades fundamentales, comúnmente conocidas como ACID: atomicidad, coherencia, asilamiento y durabilidad.

La transacción más simple en SQL Server es una única sentencia SQL. Por ejemplo, una sentencia como esta.

```
UPDATE Products SET UnitPrice = 20.00 WHERE ProductName = 'Chai'
```

Esta es una transacción 'autocommit', una transacción autocompletada.

Cuando se envía esta sentencia a SQL Server, se escribe en el fichero de transacciones lo que va a ocurrir y, a continuación, realiza los cambios necesarios en la base de datos. Si hay algún tipo de problema al hacer esta operación, SQL Server puede leer en el fichero de transacciones lo que se estaba haciendo y si es necesario, puede devolver la base de datos al estado en el que se encontraba antes de recibir la sentencia.

Por supuesto, este tipo de transacciones no requieren de una intervención del usuario, puesto que el sistema se encarga de todo. Sin embargo, sí hay que realizar varias operaciones y crear esas transacciones de manera explícita.

4.2 Sentencias para una transacción

Un conjunto de operaciones debe marcarse como transacción para que todas las operaciones que la conforman tengan éxito o fracasen.

La sentencia que se utiliza para indicar el comienzo de una transacción es 'BEGIN TRAN'. Si alguna de las operaciones de una transacción falla, hay que deshacer la transacción en su totalidad para volver al estado inicial, la base de datos antes de empezar. Esto se consigue con la sentencia 'ROLLBACK TRAN'.

Si todas las operaciones de una transacción se completan con éxito, se marca el fin de una transacción, para que la base de datos vuelva a estar en un estado consistente, con la sentencia 'COMMIT TRAN'.

Ejemplos de aplicación

Se trabajará con la base de datos Northwind. Se realizará una transacción que modifica el precio de dos productos de la base de datos.

```
Use Northwind
/* Se declara una variable que será utilizada para almacenar un posible código
de error*/
DECLARE @li_error    int

-- Iniciar la transacción
BEGIN TRAN
-- Ejecutar la primera sentencia
UPDATE Products SET UnitPrice = 20.00 WHERE ProductName = 'Chai'
-- Si ocurre un error, almacenar su código en @li_error
-- y saltar al trozo de código que cancelará la transacción.
-- Se puede mejorar el código con Try Catch
IF (@Error<>0) GOTO TratarError

-- Si la primera sentencia se ejecuta con éxito, pasar a la segunda
UPDATE Products SET UnitPrice = 20.00 WHERE ProductName = 'Chang'
SET @Error=@@ERROR
--Y si hay un error se hace como antes
IF (@Error<>0) GOTO TratarError

--Si se llega hasta aquí es que los dos UPDATE se han completado con
--éxito y se confirma la transacción en la base de datos
COMMIT TRAN

TratarError:
--Si ha ocurrido algún error, se finaliza
IF @@Error<>0
BEGIN
    PRINT 'Ha ocurrido un error. Abortar la transacción'
    --Se comunica al usuario y se deshace la transacción
    --todo volverá a estar como si nada hubiera ocurrido
    ROLLBACK TRAN
END
```

Como se puede ver para cada sentencia que se ejecuta, se observa si se ha producido o no un error; si un error es detectado, se ejecuta el bloque de código que deshace la transacción.

Hay una interpretación incorrecta en cuanto al funcionamiento de las transacciones que está bastante extendida. Mucha gente cree que si se tienen varias sentencias dentro de una transacción y una de ellas falla, la transacción se aborta en su totalidad. Esto no es correcto.

```
BEGIN TRAN
UPDATE Products SET UnitPrice = 20.00 WHERE Name = 'Chai'
UPDATE Products SET UnitPrice = 20.00 WHERE Name = 'Chang'
COMMIT TRAN
```

Estas dos sentencias se ejecutarán como una sola. Si, por ejemplo, en medio de la transacción (después del primer UPDATE y antes del segundo) hay un corte de electricidad, cuando SQL Server se recupere se encontrará en medio de una transacción y en este caso, o bien la termina o bien la deshace, pero no se quedará a medias.

El error está en pensar que si la ejecución de la primera sentencia da un error se cancelará la transacción. SQL Server se preocupa de ejecutar las sentencias, no de averiguar si lo hacen correctamente o si la lógica de la transacción es correcta.

4.3 Transacciones anidadas

Otra de las posibilidades que ofrece SQL Server es utilizar transacciones anidadas. Esto quiere decir que se pueden tener transacciones dentro de transacciones, es decir, empezar una nueva transacción sin haber terminado la anterior.

Asociada a esta idea de anidamiento existe una variable global @@TRANCOUNT que tiene valor 0, si no existe ningún nivel de anidamiento, 1 si hay una transacción anidada, 2 si está en el segundo nivel de anidamiento, y así sucesivamente.

No obstante, el trabajar con transacciones anidadas tiene dificultades en el comportamiento que tienen ahora las sentencias 'COMMIT TRAN' y 'ROLLBACK TRAN'

- **ROLLBACK TRAN:** dentro de una transacción anidada esta sentencia deshace todas las transacciones internas hasta la instrucción BEGIN TRANSACTION más externa.
- **COMMIT TRAN:** dentro de una transacción anidada esta sentencia reduce en 1 el valor de @@TRANCOUNT, pero no finaliza ninguna transacción, ni guarda los cambios. En el caso en el que @@TRANCOUNT=1 (en la última transacción) COMMIT TRAN hace que todas las modificaciones efectuadas sobre los datos desde el inicio de la transacción sean parte permanente de la base de datos, libera los recursos mantenidos por la conexión y reduce @@TRANCOUNT a 0.

Un ejemplo para entender cómo funciona.

```
CREATE TABLE Test (Columna int)
BEGIN TRAN TranExterna -- @@TRANCOUNT ahora es 1
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
INSERT INTO Test VALUES (1)
BEGIN TRAN TranInterna1 -- @@TRANCOUNT ahora es 2.
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
INSERT INTO Test VALUES (2)
BEGIN TRAN TranInterna2 -- @@TRANCOUNT ahora es 3.
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
INSERT INTO Test VALUES (3)
COMMIT TRAN TranInterna2 -- Reduce @@TRANCOUNT a 2.
-- Pero no se guarda nada en la base de datos.
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
COMMIT TRAN TranInterna1 -- Reduce @@TRANCOUNT a 1.
```

```
-- Pero no se guarda nada en la base de datos.
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
COMMIT TRAN TranExterna -- Reduce @@TRANCOUNT a 0.
-- Se lleva a cabo la transacción externa y todo lo que conlleva.
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
SELECT * FROM Test
```

A continuación, un ejemplo de transacción anidada con ROLLBACK TRAN.

```
BEGIN TRAN TranExterna -- @@TRANCOUNT ahora es 1
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
INSERT INTO Test VALUES (1)
BEGIN TRAN TranInterna1 -- @@TRANCOUNT ahora es 2.
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
INSERT INTO Test VALUES (2)
BEGIN TRAN TranInterna2 -- @@TRANCOUNT ahora es 3.
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
INSERT INTO Test VALUES (3)
ROLLBACK TRAN --@@TRANCOUNT es 0 y se deshace
--la transacción externa y todas las internas
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
SELECT * FROM Test
```

En este caso no se inserta nada puesto que el ROLLBACK TRAN deshace todas las transacciones dentro del anidamiento hasta la transacción más externa y además, hace @@TRANCOUNT=0

Las transacciones por SQL Server permiten programar de manera natural los anidamientos.

4.4 Sentencia Save Tran

Esta sentencia crea un punto de almacenamiento dentro de una transacción. Esta marca sirve para deshacer una transacción en curso solo hasta ese punto. Por supuesto la transacción debe continuar y terminar con un COMMIT TRAN (o los que hagan falta) para que todo se guarde o con un ROLLBACK TRAN para volver al estado previo al primer BEGIN TRAN.

```
BEGIN TRAN TranExterna -- @@TRANCOUNT ahora es 1
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
INSERT INTO Test VALUES (1)
BEGIN TRAN TranInterna1 -- @@TRANCOUNT ahora es 2.
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
INSERT INTO Test VALUES (2)
SAVE TRAN Guardada
BEGIN TRAN TranInterna2 -- @@TRANCOUNT ahora es 3.
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
INSERT INTO Test VALUES (3)
ROLLBACK TRAN Guardada -- se deshace lo hecho el punto guardado.
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
--Ahora se podrá decidir si la transacción se lleva a cabo
--o se deshace completamente
--Para deshacerla un ROLLBACK bastará como se ha visto
--Pero para guardar la transacción hace falta reducir @@TRANCOUNT a 0
COMMIT TRAN TranInterna1 -- Reduce @@TRANCOUNT a 2.
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
COMMIT TRAN TranInterna2 -- Reduce @@TRANCOUNT a 1.
-- Pero no se guarda nada en la base de datos.
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
COMMIT TRAN TranExterna -- Reduce @@TRANCOUNT a 0.
-- Se lleva a cabo la transacción externa y todo lo que conlleva.
SELECT 'El nivel de anidamiento es', @@TRANCOUNT
SELECT * FROM Test
```


Si no se pone el nombre del punto salvado con SAVE TRAN, al hacer un ROLLBACK TRAN se deshace la transacción más externa y @@TRANCOUNT se pone a 0.