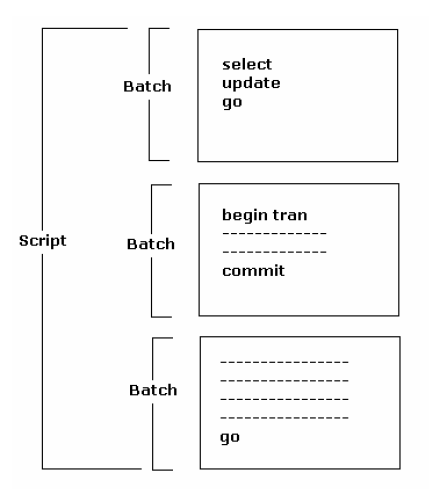
**BATCHS y SCRIPTS**



**BATCHS**

Un Batch es un lote de una o más sentencias Transact-SQL, enviadas en una sola vez desde una aplicación a Microsoft SQL Server para ser ejecutada en un solo intento. La cláusula GO indica el fin de un Batch.

Lo utilitarios de SQL Server interpretan el GO como un signo de que el Batch (conjunto de sentencias) debe ser enviado para su ejecución. El Batch actual, está compuesto por todas las sentencias antes de la cláusula GO.

Una sentencia Transact-SQL no puede ocupar en una misma línea el comando GO

Las variables locales definidas por el usuario son limitadas dentro del Batch, esto significa que no pueden ser referenciadas después del comando GO.

**SCRIPTS**

Es un programa, o sea un conjunto de comandos, que se le da al motor SQL para decirle lo que debe hacer y en qué orden debe hacerlo. Los Scripts son series de Batchs ejecutados uno después de otro.

**Comando GO**

Indica a SQL Server el final de un lote de instrucciones Transact-SQL.  GO no es una instrucción Transact-SQL, sino un comando

El SQL Server interpreta el GO como una señal de que deben enviar el lote actual de instrucciones Transact-SQL a una instancia de SQL Server. El lote actual de instrucciones está formado por todas las instrucciones especificadas desde el último comando GO o desde el comienzo de la sesión.

**Variables en SQL Server**

**Introducción**

Las variables nos permiten almacenar un valor y recuperarlo más adelante para emplearlos en otras sentencias. Las variables de usuario son específicas de cada conexión y son liberadas automáticamente al abandonar la conexión. Las variables de usuario comienzan con "@" (arroba) seguido del nombre (sin espacios), dicho nombre puede contener cualquier carácter.

**Declaración**

Una variable debe ser declarada antes de usarse. Una variable local se declara así:

declare @NOMBREVARIABLE TIPO

Colocando "declare" el nombre de la variable que comienza con el símbolo arroba (@) y el tipo de dato. Ejemplo:

         declare @nombre varchar(20)

Puede declarar varias variables en una misma sentencia:

declare @nombre varchar(20), @edad int

No existen variables globales en SQL Server. Una variable declarada existe dentro del entorno en que se declara; debemos declarar y emplear la variable en el mismo lote de sentencias, porque si declaramos una variable y luego, en otro bloque de sentencias pretendemos emplear, dicha variable ya no existe. Por ejemplo, si ejecutamos estas sentencias en diferentes lotes:

declare @variable varchar(10); select @variable;

Aparece un mensaje indicando que la variable "@variable" debe ser declarada.

Debemos tipear:

declare @variable varchar(10)

select @variable;

Disponemos punto y coma solo al final de la última instrucción del lote. Una variable a la cual no se le ha asignado un valor contiene "null".

La asignación de variables se realiza con la sentencia SELECT o SET

set @edad=45

Para almacenar un valor en una variable se coloca el signo igual (=) entre la variable y el valor a asignar. Si le asignamos un valor resultado de una consulta, la sintaxis es:

select @nombre = name from Production.Product where ProductID = 998

Podemos ver el contenido de una variable con:

         select @nombre;

La utilidad de las variables consiste en que almacenan valores para utilizarlos en otras consultas. Por ejemplo, queremos saber todos los datos de productos con mayor precio de la tabla "Product". Para ello podemos emplear una variable para almacenar el precio más alto:

 declare @mayorprecio

 select @mayorprecio = max(ListPrice) from Production.Product

y luego mostrar todos los datos de dicho producto empleando la variable anterior:

select \*from libros

where precio=@mayorprecio;

Una variable puede ser definida con cualquier tipo de dato, excepto text, ntext e image; incluso de un tipo de dato definido por el usuario.

**Variables de tipo Tabla**

Podemos declarar variables de tipo table. Este tipo de variables tienen una serie de ventajas sobre las tablas temporales por lo que siempre que podamos escogeremos usar variables de tabla frente a tablas temporales. Usar variables temporales es sencillo:

DECLARE @VariableTabla TABLE (Campo1 int, Campo2 char(50))  
INSERT INTO @VariableTabla VALUES (1,'Primer campo')  
INSERT INTO @VariableTabla VALUES (2,'Segundo campo')  
SELECT \* FROM @VariableTabla

Ventajas que encontraremos al usar variables de tipo tabla:

* Tienen un ámbito bien definido. El procedimiento almacenado, la función o el batch en el que se declaran.
* Las variables de tipo tabla producen menos recopilaciones de los procedimientos almacenados en los que se encuentran que si utilizamos tablas temporales.
* Las variables de tabla no necesitan de bloqueos ni de tantos recursos como las tablas temporales.

Pero también tienen inconvenientes:

* No podemos cambiar la definición de la tabla una vez declarada
* No podemos utilizar índices que no sean agrupados
* No se pueden utilizar en INSERT INTO ni en SELECT INTO
* No podemos utilizar funciones en las restricciones

Si ponemos en una balanza las ventajas y los inconvenientes vemos que en general es mejor utilizar las variables de tipo tabla que las tablas temporales. Solo en el caso de tener gran cantidad de datos en una tabla temporal y si la vamos a usar varias veces es preferible la opción de tablas temporales porque en ellas podemos definir índices.

**Variables Globales pre declaradas**

Como habíamos visto las variables es SQL se declaran con la cláusula DECLARE y el nombre de la variable debe preceder el símbolo @

SQL proporciona variables del sistema predefinidas

Los nombres de estas variables comienzan con el doble símbolo @@

No pueden asignarse valores. Solamente consultar.

SELECT convert(money, substring(context\_info, 1, 8))

FROM master..sysprocesses

WHERE spid = @@spid

En este ejemplo @@spid devuelve el identificador de la conexión actual.

**Control de Flujo**

**Bloque BEGIN – END**

Encierra un conjunto de instrucciones Transact-SQL de forma que se pueda ejecutar un grupo de instrucciones Transact-SQL. BEGIN y END son palabras clave del lenguaje de control de flujo.

Se utilizan para controles de flujos como IF, ELSE, WHILE.

**Bloque IF … ELSE**

Impone condiciones en la ejecución de una instrucción Transact-SQL.

La instrucción Transact-SQL (sql\_statement) que sigue a Boolean\_expression se ejecuta si Boolean\_expression se evalúa como TRUE.

La palabra clave opcional ELSE es una instrucción Transact-SQL alternativa que se ejecuta cuando Boolean\_expression se evalúa como FALSE o NULL.

**Construcción WHILE, BREAK y CONTINUE**

Establece una condición para la ejecución repetida de una instrucción o bloque de instrucciones SQL.

Las instrucciones se ejecutan repetidamente siempre que la condición especificada sea verdadera. Se puede controlar la ejecución de instrucciones en el bucle WHILE con las palabras clave BREAK y CONTINUE.

BREAK: Produce la salida del bucle WHILE más interno. Se ejecutan las instrucciones que aparecen después de la palabra clave END, que marca el final del bucle.

CONTINUE: Hace que se reinicie el bucle WHILE y omite las instrucciones que haya después de la palabra clave CONTINUE.

**GOTO**

La instrucción GOTO provoca que, en la ejecución de un lote de Transact-SQL, se salte a una etiqueta.

Ninguna de las instrucciones situadas entre la instrucción GOTO y la etiqueta se ejecutarán. El nombre de la etiqueta se define con la sintaxis:

GOTO label\_name

…

…

…

label\_name:

La etiqueta que constituye el objetivo de un GOTO sólo identifica el destino del salto

**WAITFOR**

La instrucción WAITFOR suspende la ejecución de un lote, un procedimiento almacenado o una transacción hasta que:

Haya pasado un intervalo de tiempo especificado.

Se haya alcanzado una hora del día especificada.

Una instrucción RECEIVE especificada modifique o devuelva como mínimo una fila en una cola de Service Broker.

WAITFOR DELAY '00:00:02'

print 'pasaron 2 segundos'

**Sentencia EXECUTE**

Ejecuta uno de los siguientes módulos: procedimiento almacenado del sistema, procedimiento almacenado definido por el usuario, función con valores escalares definida por el usuario o procedimiento almacenado extendido.

Ejecuta una cadena de comandos o una cadena de caracteres dentro de un proceso por lotes de Transact-SQL.

EXECUTE ('select \* from titles')

**Expresión CASE**

**Descripción**

Evalúa una lista de condiciones y devuelve una de las varias expresiones de resultado posibles. La función "case" compara 2 o más valores y devuelve un resultado.

**Función: Simple CASE:**

La función CASE sencilla compara una expresión con un conjunto de expresiones sencillas para determinar el resultado.

La sintaxis es la siguiente:

case VALORACOMPARAR  
 when VALOR1 then RESULTADO1  
 when VALOR2 then RESULTADO2  
 ...  
 else RESULTADO3  
end

Por cada valor hay un "when" y un "then"; si encuentra un valor coincidente en algún "where" ejecuta el "then" correspondiente a ese "where", si no encuentra ninguna coincidencia, se ejecuta el "else"; si no hay parte "else" retorna "null". Finalmente se coloca "end" para indicar que el "case" ha finalizado.

Ejemplo:

Un profesor guarda las notas de sus alumnos de un curso en una tabla llamada "alumnos" que consta de los siguientes campos:

- nombre (30 caracteres),  
- nota (valor entero entre 0 y 10, puede ser nulo).  
Mostrar los nombres, notas de los alumnos y en una columna extra llamada "resultado" empleamos un case que testee la nota y muestre un mensaje diferente si en dicho campo hay un valor:

- 0, 1, 2 ó 3: 'libre';  
- 4, 5 ó 6: 'regular';  
- 7, 8, 9 ó 10: 'promocionado';

select nombre,nota,  
case nota  
 when 0 then 'libre'  
 when 1 then 'libre'  
 when 2 then 'libre'  
 when 3 then 'libre'  
 when 4 then 'regular'  
 when 5 then 'regular'  
 when 6 then 'regular'  
 when 7 then 'promocionado'  
 when 8 then 'promocionado'  
 when 9 then 'promocionado'  
 when 10 then 'promocionado'  
end  
from alumnos

Note que cada "where" compara un valor puntual, por ello los valores devueltos son iguales para algunos casos. Note que como omitimos la parte "else", en caso de que el valor no encuentra coincidencia con ninguno valor "when", retorna "null".

**Función: Searched CASE:**

La función CASE buscada evalúa un conjunto de expresiones booleanas para determinar el resultado.

Podemos realizar comparaciones en cada "where". La sintaxis es la siguiente:

case  
 when VALORACOMPARAR OPERADOR VALOR1 then RESULTADO1  
 when VALORACOMPARAR OPERADOR VALOR2 then RESULTADO2  
 ...  
 else RESULTADO3  
end

Mostramos los nombres de los alumnos y en una columna extra llamada "resultado" empleamos un case que teste si la nota es menor a 4, está entre 4 y 7 o supera el 7:

select nombre, nota, resultado=  
  case   
  when nota<4 then 'libre'  
  when nota >=4 and nota<7 then 'regular'  
  when nota>=7 then 'promocionado'  
  else 'sin nota'  
 end  
from alumnos;

**Ordenamiento con CASE**

Se puede utilizar la función CASE realizar ordenamientos que no sean alfabéticos o numéricos, tanto ascendentes como descendentes. Por ejemplo, queremos realizar un reporte de todos los autores ordenado por el estado en el que viven pero con el siguiente criterio: Primero los que vivan en California, luego los que viven en Washington y el resto ordenado alfabéticamente.

select state, au\_fname, au\_lname

from authors

order by        case when state = “CA” then 1

                         when state = “WA” then 2

else 3 end,

state

**Manejo de Errores con SQL**

**TRY - CATCH**

**Description**

Además del @@ERROR existe una sentencia más práctica a la hora de detectar que se ha producido un error en un procedimiento y ejecutar un ROLLBACK en consecuencia. Al igual que en otros lenguajes como C#.NET se puede abrir un TRY para la secuencia de comandos que podrían dar error y un CATCH para realizar un “deshacer” si esto ocurre.

**Características**

La a partir de la versión 2005 implementa un mecanismo de control de errores para T-SQL.

Se puede incluir un grupo de instrucciones T-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.

Un bloque TRY debe ir seguido inmediatamente por un bloque CATCH. Un bloque CATCH se inicia con la instrucción BEGIN CATCH y finaliza con la instrucción END CATCH. En Transact-SQL, cada bloque TRY se asocia a un sólo bloque CATCH.

**Sintaxis**

BEGIN TRY

     { sql\_statement | statement\_block }

END TRY

BEGIN CATCH

     { sql\_statement | statement\_block }

END CATCH

Cuando utilice la construcción TRY…CATCH, tenga en cuenta las siguientes directrices y sugerencias:

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.

Cada construcción TRY…CATCH debe encontrarse en un solo lote, procedimiento almacenado o desencadenador.

Por ejemplo, no se puede colocar un bloque TRY en un lote y el bloque asociado CATCH en otro. La siguiente secuencia de comandos generaría un error:

BEGIN TRY

    SELECT \*

        FROM sys.messages

        WHERE message\_id = 21;

END TRY

GO   --Error!!!!!!

BEGIN CATCH

    SELECT ERROR\_NUMBER () AS ErrorNumber;

END CATCH;

GO

Un bloque CATCH debe seguir inmediatamente a un bloque TRY.

Las construcciones TRY…CATCH pueden estar anidadas. Esto significa que las construcciones TRY…CATCH se pueden colocar dentro de otros bloques TRY y CATCH. Cuando se produce un error dentro de un bloque TRY anidado, el control del programa se transfiere al bloque CATCH que está asociado al bloque TRY anidado.

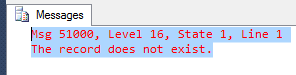
**Sentencia THROW**

Produce una excepción y transfiere la ejecución a un bloque CATCH de una construcción TRY...CATCH.

Ejemplo:

THROW 51000,'The record does not exist.', 1;

Resultado:



Lo nuevo en TRY - CATCH es el THROW en la que ahora podemos manejar los errores internos de SQL Server. Cuando poseemos una estructura TRY CATCH, al fallar la sentencia TRY se maneja un error interno de SQL Server, THROW permite capturar el error interno que se ejecuta en el TRY y nos permite mostrarlo como mensaje dentro del bloque CATCH, lo que nos va permitir administrar y poder almacenar los errores internos que se originan en nuestras transacciones programadas, podemos manejar nuestro propio log de errores y de esa manera generar estadísticas por errores, a fin de mejorar nuestra programación y como tal también nuestras aplicaciones.

**TRY CATCH THROW: La captura de errores ha cambiado T-SQL**

Cuando SQL Server 2005 introdujo la sintaxis BEGIN TRY y BEGIN CATCH, que era una gran mejora con respecto al manejo de errores anterior sobre la base de comprobación @@ERROR después de cada declaración. T-SQL, se unió a la fila de los lenguajes de programación, no más que un lenguaje de acceso a datos. La experiencia ha demostrado que el control de excepciones conduce a un mejor código en comparación con los controles de error.

Pero cuando se trata de utilizar el nuevo TRY / CATCH de manejo de excepciones en el código T-SQL, se hizo evidente rápidamente uno de los problemas: el bloque CATCH se enmascara los metadatos de error original: número de error / gravedad / estado, el texto de error, la línea de origen y así sucesivamente. Dentro de un bloque CATCH el código sólo se le permitió subir un nuevo archivo \* error \*. Claro, la información del error original podría ser transmitida en el mensaje de error generado, pero sólo como un mensaje. El código de todos los errores importantes se ha cambiado. Esto puede parecer un tema menor, pero resulta que tiene un efecto cascada muy en serio: la persona que llama ahora tiene que entender los nuevos códigos de error, en lugar de los códigos de error del sistema original. Si el código de la aplicación se preparó para controlar interbloqueos (código de error 1205) de una manera determinada (por ejemplo, vuelva a intentar la operación), con un try de T-SQL / catch el código de error estancamiento que de repente se traduce en algo por encima de 50000.

**Sentencia RAISERROR (deprecado)**

Con la introducción de THROW, RAISERROR fue declarado obsoleto. Hasta el momento coexisten ambos.

**Sentencia PRINT**

PRINT: Devuelve al cliente un mensaje definido por el usuario. Devuelve una cadena de caracteres o una constante de cadena Unicode

**Transacciones**

**Administración de Transacciones**

Las aplicaciones controlan las transacciones principalmente al especificar cuándo se inicia y finaliza una transacción.

El sistema también debe ser capaz de controlar correctamente los errores que terminan una transacción antes de que se concluya.

De manera predeterminada, las transacciones se administran en las conexiones. Cuando se inicia una transacción en una conexión, todas las instrucciones Transact-SQL ejecutadas en esa conexión forman parte de la transacción hasta que ésta finaliza. No obstante, en una sesión de conjunto de resultados activos múltiples (MARS), una transacción de Transact-SQL explícita o implícita se convierte en una transacción de lote que se administra en los lotes. Cuando se termina el lote, si la transacción de lote no se confirma ni se revierte, SQL Server la revierte automáticamente.

Existen dos tipos de transacciones. Transacciones Implícitas y Transacciones Explícitas.

**Transacciones Explícitas**

Una transacción explícita es aquella en que se define explícitamente el inicio y el final de la transacción. Las transacciones explícitas también recibían el nombre de transacciones definidas por el usuario.

Las secuencias de comandos Transact-SQL utilizan las instrucciones BEGIN TRANSACTION, COMMIT TRANSACTION, ROLLBACK TRANSACTION de Transact-SQL para definir transacciones explícitas.

BEGIN TRANSACTION: Marca el punto de inicio de una transacción explícita para una conexión.

COMMIT TRANSACTION: Se utiliza para finalizar una transacción correctamente si no hubo errores. Todas las modificaciones de datos realizadas en la transacción se convierten en parte permanente de la base de datos. Se liberan los recursos ocupados por la transacción.

ROLLBACK TRANSACTION: Se utiliza para eliminar una transacción en la que se encontraron errores. Todos los datos modificados por la transacción vuelven al estado en el que estaban al inicio de la transacción. Se liberan los recursos ocupados por la transacción.

**Transacciones Implícitas**

Cuando una conexión funciona en modo de transacciones implícitas, SQL Server Database Engine (Motor de base de datos de SQL Server) inicia automáticamente una nueva transacción después de confirmar o revertir la transacción actual.

No tiene que realizar ninguna acción para delinear el inicio de una transacción, sólo tiene que confirmar o revertir cada transacción. El modo de transacciones implícitas genera una cadena continua de transacciones.

SET IMPLICIT\_TRANSACTIONS ON

**Chequeo de Errores dentro de la Transacción**

La variable @@ERROR devuelve el código de error producido

USE PUBS

BEGIN TRAN

UPDATE authors

SET state = 'FL'

WHERE state = 'KS'

IF @@ERROR <> 0 BEGIN

    ROLLBACK TRAN

    GOTO ON\_ERROR

 END

UPDATE jobs

SET min\_lvl = min\_lvl - 10

IF @@ERROR <> 0 BEGIN

    ROLLBACK TRAN

    GOTO ON\_ERROR

 END

COMMIT TRAN

ON\_ERROR:

SELECT \* FROM authors

WHERE state = 'FL'

**Transacciones: Nivel de Aislamiento**

El nivel de aislamiento con el que se ejecuta una instrucción Transact-SQL determina su comportamiento de bloqueo y de versión de fila.

READ UNCOMMITTED es el nivel de aislamiento menos restrictivo porque omite los bloqueos realizados por otras transacciones. Las transacciones que se ejecutan con READ UNCOMMITTED pueden leer valores de datos modificados que aún no han confirmado otras transacciones; éstos se conocen como lecturas no confirmadas.

READ COMMITTED es el nivel de aislamiento predeterminado en SQL Server. Impide las lecturas no confirmadas al especificar que las instrucciones no pueden leer valores de datos modificados que aún no hayan confirmado otras transacciones.

REPEATABLE READ es un nivel de aislamiento más restrictivo que READ COMMITTED. Incluye READ COMMITTED y además especifica que ninguna otra transacción puede modificar ni eliminar datos que la transacción actual haya leído hasta que ésta no se confirme. La simultaneidad es menor que en READ COMMITTED porque durante la transacción se mantienen bloqueos compartidos en los datos leídos en lugar de liberarlos al final de cada instrucción.

SNAPSHOT La transacción únicamente puede reconocer las modificaciones de datos confirmadas antes del comienzo de la misma. Las instrucciones que se ejecuten en la transacción actual no verán las modificaciones de datos efectuadas por otras transacciones después del inicio de la transacción actual. El efecto es el mismo que se obtendría si las instrucciones de una transacción obtuviesen una instantánea de los datos confirmados tal como se encontraban al comienzo de la transacción.

SERIALIZABLE es el nivel de aislamiento más restrictivo, dado que bloquea intervalos enteros de claves y mantiene los bloqueos hasta que la transacción finaliza. Incluye REPEATABLE READ y agrega la restricción de que otras transacciones no pueden insertar filas nuevas en intervalos que haya leído la transacción hasta que ésta no finalice.

**Sentencias de Transacción: ROLLBACK y SAVE**

Se puede evitar tener que deshacer toda la transacción empleando un punto de almacenamiento para deshacer hasta cierto punto de la transacción, en lugar de llegar hasta el comienzo de la transacción.

Todas las modificaciones producidas hasta el punto de almacenamiento siguen siendo válidas y no se deshacen, pero las instrucciones ejecutadas después del punto de almacenamiento (que debe especificarse en la transacción) y hasta la instrucción ROLLBACK se deshacen.

Las instrucciones situadas a continuación de la instrucción ROLLBACK seguirán ejecutándose. Si se deshace posteriormente la transacción sin especificar un punto de almacenamiento, todas las modificaciones se anulan hasta el principio de la transacción, como siempre; se deshace toda la transacción.

Hay que tener en cuenta que cuando se deshace una transacción hasta un punto de almacenamiento, SQL Server no libera los recursos bloqueados. Se liberan cuando se confirma la transacción o al producirse una cancelación de toda la transacción.

Para especificar un punto de almacenamiento en una transacción se utiliza la instrucción siguiente:

SAVE TRAN[SACTION] nombre

Se añade de un punto de almacenamiento en la transacción en la posición hasta la que se desea deshacer. Para deshacer hasta el punto de almacenamiento hay que utilizar ROLLBACK TRAN con el nombre del punto de almacenamiento, como se muestra a continuación:

ROLLBACK TRAN nombre

Se pueden tener más instrucciones de T-SQL después de la instrucción ROLLBACK para continuar la transacción. Hay que recordar incluir una instrucción COMMIT u otra instrucción ROLLBACK después de la primera instrucción ROLLBACK para que se complete toda la transacción.