IES CHAN DO MONTE
C.S. de Desarrollo de Aplicaciones Multiplataforma
Módulo Base de datos

UNIDAD 8 : PROCEDIMIENTOS ALMACENADOS Y FUNCIONES DEFINIDAS POR EL USUARIO

Índice

1. Programación con TRANSACT SQL

Con Transact SQL vamos a poder programar:

- Procedimientos almacenados
- Funciones
- Triggers
- Cursores
- Scripts

2. Variables

Una **variable** es una **zona de memoria**, caracterizada por un **nombre y un tipo**, que permite almacenar un valor.

Las variables Transact-SQL **deben declararse obligatoriamente antes de su utilización**.

Declaración de variables

```
DECLARE @nombre_variable tipo [,...]
```

nombre_variable._ Nombre precedido del carácter **@.**

tipo._ Tipo de sistema o definido por el usuario.

Asignación de valores a las variables

En Transact SQL podemos **asignar valores a una variable** de varias formas:

- A través de la instruncción **SET.**
- Utilizando una sentencia **SELECT**.

El alcance de una variable

Es el conjunto de instrucciones Transact-SQL desde las que se puede hacer referencia a la variable.

El **alcance de una variable** se extiende desde el **punto en el que se declara hasta el final del lote o procedimiento almacenado en el que se ha declarado**.

Ejemplo: El siguiente ejemplo muestra como asignar una variable utilizando la instrucción SET.

```
DECLARE @nombre VARCHAR(100)
-- La consulta debe devolver un único registro
SET @nombre = (SELECT nombre FROM CLIENTES WHERE ID = 1)
PRINT @nombre
```

El siguiente ejemplo muestra como asignar variables utilizando una sentencia SELECT.

```
DECLARE @nombre VARCHAR(100),
        @apellido1 VARCHAR(100),
        @apellido2 VARCHAR(100)
SELECT  @nombre=nombre , @apellido1=Apellido1,   @apellido2=Apellido2
FROM CLIENTES WHERE ID = 1
PRINT @nombre
PRINT @apellido1
PRINT @apellido2 Alcance de una variable
```

Por ejemplo, esta secuencia de comandos **genera un error de sintaxis porque la variable se declara en un lote y se hace referencia a la misma en otro**:

```
DECLARE @Var int;
SET @Var = 27;
GO   -- Terminamos el lote
/* @Var ya no existe a partir de aquí. Este SELECT da error debido a que ya
   no existe @Var*/
SELECT Nombre, habitants   FROM Provincia
WHERE Codigo = @Var;  --ERROR---
```

3. Operadores en TRANSACT SQL

La siguiente tabla ilustra los operadores de Transact SQL :

TIPO DE OPERADOR	OPERADORES
Operador de asignación	**=**
Operadores aritméticos	**+** (suma) **-** (resta) ***** (multiplicación) **/** (división) ****** (exponente) **%** (módulo)
Operadores relacionales o de comparación	**=** (igual a) **<>**(distinto de) **!=** (distinto de) **<**(menor que) **>**(mayor que) **>=** (mayor o igual a) **<=** (menor o igual a) **!>** (no mayor a) **!<** (no menor a)
Operadores lógicos	**AND** (y lógico) **NOT** (negacion) **OR** (o lógico) **&** (AND a nivel de bit) **|** (OR a nivel de bit) **^** (OR exclusivo a nivel de bit)
Operador de concatenación	**+**
Otros	**ALL** (Devuelve TRUE si el conjunto completo de comparaciones es TRUE) **ANY**(Devuelve TRUE si cualquier elemento del conjunto de comparaciones es TRUE) **BETWEEN** (Devuelve TRUE si el operando está dentro del intervalo) **EXISTS** (TRUE si una subconsulta contiene filas) **IN** (TRUE si el operando está en la lista) **LIKE** (TRUE si el operando coincide con un patron) **NOT** (Invierte el valor de cualquier operador booleano) **SOME**(Devuelve TRUE si alguna de las comparaciones de un conjunto es TRUE)

4. Estructuras de control en TRANSACT SQL

4.1 Bloques BEGIN...END

- ✓ Los bloques BEGIN…END contienen **una serie de instrucciones Transact-SQL** que **son tratadas como una unidad** llamada bloque de instrucción.
- ✓ **Se usan para agrupar varias instrucciones** Transact-SQL en un bloque lógico.
- ✓ Se pueden usar en cualquier parte cuando una instrucción de control de flujo deba ejecutar un bloque **con dos o más instrucciones.**
  - Cuando es necesario que un bucle WHILE incluya un bloque de instrucciones.
  - Cuando es necesario que un elemento de una función CASE incluya un bloque de instrucciones.
  - Cuando es necesario que una cláusula IF o ELSE incluya un bloque de instrucciones.

Sintaxis:

```
BEGIN
{instrucción_sql | bloque_instrucciones}
END
```

4.2 Estructura condicional IF

La **estructura condicional IF** permite **evaluar una expresión booleana (resultado SI-NO),** y ejecutar las operaciones contenidas en el bloque formado por BEGIN END.

La sintaxis general de IF es:

```
IF (<expresion>)
 BEGIN
    ...
 END
ELSE IF (<expresion>)
 BEGIN
    ...
 END
ELSE
 BEGIN
    ...
 END
```

A continuación vemos un ejemplo del uso de la estructura condicional IF:

```
DECLARE @Web varchar(100), @diminutivo varchar(3)
SET @diminutivo = 'VOZ'
IF @diminutivo = 'VOZ'
 BEGIN
        PRINT 'www.lavozdegalicia.es'
 END
ELSE
 BEGIN
       PRINT 'Otra Web '
 END
```

La estructura **IF admite el uso de subconsultas:**

```
DECLARE @codPais int, @descripcion varchar(255)
set @codPais = 5
set @descripcion = 'España'
IF EXISTS(SELECT * FROM PAIS WHERE COD_PAIS = @codPais)
 BEGIN
    UPDATE PAIS
    SET DESCRIPCION = @descripcion
    WHERE COD_PAIS = @codPais
 END
ELSE
 BEGIN
    INSERT INTO PAIS(COD_PAIS, DESCRIPCION)
    VALUES(@codPais, @descripcion)
 END
```

4.3 Estructura condicional CASE

La **estructura condicional CASE** permite evaluar **una expresión y devolver un valor u otro**.

La **sintaxis general** de case es:

```
CASE <expresion>
   WHEN <valor_expresion> THEN <valor_devuelto>
   WHEN <valor_expresion> THEN <valor_devuelto>
   ELSE <valor_devuelto>     -- Valor por defecto
END
```

A continuación vemos un ejemplo del uso de la estructura condicional CASE:

```
DECLARE @Web varchar(100),   @diminutivo varchar(3)
SET @diminutivo = 'VOZ'
SET @Web = (CASE @diminutivo
                    WHEN 'VOZ' THEN 'www.lavozdegalicia.es'

                    WHEN 'CG' THEN 'www.elcorreogallego.es'
                    ELSE 'www.elprogreso.es'
            END)
 PRINT @Web
```

Otra sintaxis de CASE nos permite evaluar diferentes expresiones:

```
CASE
   WHEN <expresion> = <valor_expresion> THEN <valor_devuelto>
   WHEN <expresion> = <valor_expresion> THEN <valor_devuelto>
   ELSE <valor_devuelto> -- Valor por defecto
END
```

El mismo ejemplo aplicando esta sintaxis:

```
DECLARE @Web varchar(100), @diminutivo varchar(3)
SET @diminutivo = 'VOZ'
SET @Web = (CASE
                  WHEN @diminutivo = 'VOZ' THEN 'www.lavozdegalicia.es'
                  WHEN @diminutivo = 'CG' THEN 'www.elcorreogallego.es'
             ELSE 'www..elprogreso.es'
            END)
 PRINT @Web
```

Otro **aspecto muy interesante de CASE es que permite el uso de subconsultas**:

```
DECLARE @Web varchar(100), @diminutivo varchar(3)
SET @diminutivo = 'VOZ'
 SET @Web = (CASE
                 WHEN @diminutivo = 'VOZ' THEN (SELECT web FROM WEBS WHERE id=1)
                 WHEN @diminutivo = 'CG' THEN (SELECT web FROM WEBS  WHERE id=2)
                  ELSE 'www.elprogreso.es'
          END)
 PRINT @Web
```

4.4 Bucle WHILE

- ✓ El bucle **WHILE se repite mientras expresión se evalúe como verdadera**.
- ✓ Es el **único tipo de bucle del que dispone Transact SQL**.

Sintaxis:

```
WHILE <expresion>
  BEGIN
    ...
  END
```

A continuación vemos un ejemplo de uso del bucle WHILE:

```
DECLARE @contador int
SET @contador = 0
WHILE (@contador < 100)
   BEGIN
     SET @contador = @contador + 1
     PRINT 'Iteracion del bucle ' + cast(@contador AS varchar)
   END
```

Podemos **pasar a la siguiente iteración del bucle utilizando CONTINUE**.

```
DECLARE @contador int
SET @contador = 0
  WHILE
    BEGIN
      SET @contador = @contador + 1
      IF (@contador % 2 = 0)
            CONTINUE
      PRINT 'Iteracion del bucle ' + cast(@contador AS varchar)
    END
```

El bucle **se dejará de repetir con la instrucción BREAK**.

```
DECLARE @contador int
SET @contador = 0
WHILE (@contador < 100)
  BEGIN
    SET @contador = @contador + 1
    IF (@contador % 50 = 0)
          BREAK
    PRINT 'Iteracion del bucle ' + cast(@contador AS varchar)
  END
```

También podemos **utilizar el bucle WHILE conjuntamente con subconsultas**.

```
DECLARE @codRecibo int
--Ojo, la subconsulta se ejecuta una vez por cada iteracion del bucle!
WHILE  EXISTS (SELECT * FROM RECIBOS WHERE PENDIENTE = 'S')
  BEGIN
       SET @codRecibo = (SELECT TOP 1 COD_RECIBO       FROM RECIBOS
                                 WHERE PENDIENTE = 'S')
      UPDATE RECIBOS
           SET PENDIENTE = 'N'
           WHERE COD_RECIBO = @codRecibo
  END
```

5. Control de errores en TRANSACT SQL

5.1 Uso de TRY CATCH

A partir d**e la versión 2005,** SQL Server proporciona el **control de errores a través de las instrucciones TRY y CATCH**. Estas nuevas instrucciones suponen un gran paso adelante en el control de errores en SQL Server, un tanto precario en las versiones anteriores.

La sintaxis de TRY CATCH es la siguiente:

```
BEGIN TRY
      ...
END TRY
BEGIN CATCH
      ...
END CATCH
```

El siguiente ejemplo ilustra el uso de TRY – CATCH:

```
BEGIN TRY
  DECLARE @divisor int, @dividendo int, @resultado int
  SET @dividendo = 100
  SET @divisor = 0
  SET @resultado = @dividendo/@divisor -- Esta linea provoca un error de division por 0
  PRINT 'No hay error'
END TRY
```

```
BEGIN CATCH
  PRINT 'Se ha producido un error'
END CATCH
```

5.2 Funciones especiales de Error

- ✓ Las funciones especiales de error **están disponibles únicamente en el bloque CATCH para la obtención de información detallada del error**. Son:
  - **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 auditoria de errores.
- ✓ 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.

Ejemplo:

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

5.3 La variable de sistema @@ERROR

- ✓ 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**.

```
DECLARE @divisor   int , @dividendo int ,  @resultado int
SET @dividendo = 100
SET @divisor = 0
SET @resultado = @dividendo/@divisor       -- Esta linea provoca un error de división por 0
IF @@ERROR = 0
    BEGIN
            PRINT 'No hay error'
    END
ELSE
    BEGIN
            PRINT 'Hay error'
    END
```

El uso de @@ERROR para controlar errores puede provocar multitud de problemas.**Uno de los más habituales es sin duda, incluir una nueva sentencia Transact SQL entre la línea que provoco el error y la que lo controla. Esa nueva instrucción restaura el valor de @@ERROR y no controlaremos el error**.

El siguiente ejemplo ilustra esta situación:

```
DECLARE @divisor   int,   @dividendo int, @resultado int
SET @dividendo = 100
SET @divisor = 0
SET @resultado = @dividendo/@divisor          -- Esta linea provoca un error de division por 0
PRINT 'Controlando el error ...'
IF @@ERROR = 0  -- Esta linea establece  @@ERROR a cero  (no da error)
   BEGIN
      -- Se ejecuta esta parte!!!!
           PRINT 'No hay error'
   END
ELSE
   BEGIN
           PRINT 'Hay error'
   END
```

5.4 Generar un error con RAISERROR

- ✓ En ocasiones es necesario **provocar voluntariamente un error**, por ejemplo nos puede interesar que se genere un error cuando los datos incumplen una regla de negocio.
- ✓ La **función RAISERROR recibe tres parámetros**, el **mensaje del error (o código de error predefinido), la severidad y el estado**.
  - La **severidad** indica el **grado de criticidad del error**. Admite valores de 0 al 25, pero solo podemos asignar valores del 0 al 18. Los errores el 20 al 25 son considerados fatales por el sistema, y cerraran la conexión que ejecuta el comando RAISERROR. Para asignar valores del 19 al 25 necesitares ser miembros de la función de SQL Server sysadmin.
  - El **estado** es un valor para permitir que el programador **identifique el mismo error desde diferentes partes del código**. Admite valores entre 1 y 127.

Podemos provocar **un error en tiempo de ejecución a través de la función RAISERROR**.

```
DECLARE @tipo int, @clasificacion int
SET @tipo = 1
SET @clasificacion = 3
IF (@tipo = 1 AND @clasificacion = 3)
 BEGIN
   RAISERROR ('El tipo no puede valer uno y la clasificacion 3',
           16,  -- Severidad
           1    -- Estado           )
 END
```

6. Lotes de instrucciones y scripts

- ✓ Un lote de instrucciones es un **conjunto de instrucciones Transact-SQL que será compilado y ejecutado en una sola unidad**.
- ✓ **Termina con la instrucción GO** .
- ✓ Un **lote puede incluir cualquier instrucción o serie de instrucciones, así como transacciones**.
- ✓ El interés de los lotes reside en la mejora del rendimiento y en la compilación única: en el caso de que se produzca un error de sintaxis, no se ejecuta ninguna instrucción. Sin embargo, los lotes están sometidos a ciertas restricciones:
  - No se pueden **combinar ciertas instrucciones en un mismo lote**: CREATE PROCEDURE, CREATE RULE, CREATE DEFAULT, CREATE TRIGGER, CREATE VIEW.

- ✓ **Cada archivo de procesamiento por lotes se procesa de forma independiente**, con lo cual un error en un archivo de procesamiento por lotes no impide que se ejecute otro archivo de procesamiento por lotes.
- ✓ Los scripts **son conjuntos de lotes que se ejecutarán sucesivamente a partir de un archivo de texto**. Estos archivos tienen, por convención, la extensión '.sql'.

En el siguiente ejemplo podemos ver el funcionamiento:

```
Declare @texto varchar(50);
SET @texto='*** Mi variable toma un valor ***';
PRINT '*** Realizado con el primer batch ***';
GO
-- la siguiente línea da error porque la variable no se ha declarado en este archivo de procesamiento por lotes
PRINT @texto;
PRINT '*** Realizado con el segundo batch ***';
GO
-- La siguiente línea se ejecuta
PRINT '*** Realizado con el tercer batch ***';
GO
```

Cada **archivo de procesamiento por lotes es autónomo en cuanto a los problemas que puedan tener en tiempo de ejecución**. Por ello, el resultado sería el siguiente:

```
*** Realizado con el primer batch ***
    Mens. 137, Nivel 15, Estado 2, Línea 4
    Debe declarar la variable escalar "@texto".
*** Realizado con el tercer batch ***
```

7. Procedimientos almacenados

- ✓ Un **procedimiento almacenado** es **una colección con nombre de instrucciones SQL** que está almacenado en un servidor. Los procedimientos almacenados son un método eficaz de **recopilar instrucciones para su repetida ejecución**.
- ✓ Soportan **variables declaradas** por el usuario, **ejecución condicional** y otras características potentes de programación.
- ✓ Son parecidos a los procedimientos en otros lenguajes de programación, ya que pueden:
  - Aceptar **parámetros de entrada**
  - **Devolver varios valores en forma de parámetros de salida** al lote o al procedimiento que realiza la llamada.
  - **Contener instrucciones de programación** que **realicen operaciones en la base de datos**, incluidas las llamadas a otros procedimientos.
  - **Devolver un valor de estado** a un lote o a un procedimiento que realiza una llamada para **indicar si la operación se ha realizado correctamente o se han producido errores** (y el motivo de éstos).

7.1 Ventajas de procedimientos almacenados

Los procedimientos almacenados ofrecen numerosas **ventajas**.

- **Compartir lógica de aplicación con otras aplicaciones**. Todos los clientes pueden **utilizar los mismos procedimientos almacenados para un acceso y modificación de datos consistente**.
- **Proporcionar mecanismos de seguridad**. Se puede **conceder permiso a los usuarios para ejecutar un procedimiento almacenado incluso si no tienen permiso para acceder a las tablas o vistas** a las que se hace referencia en el procedimiento almacenado.
- **Ejecutarse automáticamente al inicio** si la condición de inicio para el procedimiento almacenado está establecida en TRUE utilizando el procedimiento almacenado de sistema **sp_procoption**. Es una

buena idea el tener un solo procedimiento almacenado identificado en el inicio para su ejecución automática. Este procedimiento almacenado puede llamar a procedimientos almacenados adicionales para que no tenga que utilizar subprocesos innecesarios.

- **Aumentar el rendimiento**, ya que los planes de ejecución residen en la caché después de que son ejecutados por primera vez.
- **Reducir tráfico de red**. En lugar de enviar cientos de instrucciones Transact_SQL por la red, los usuarios **pueden realizar una operación compleja ejecutando un único procedimiento almacenado**, lo cual reduce el número de peticiones que pasan entre el cliente y el servidor.

7.2 Proceso en la creación de un procedimiento almacenado

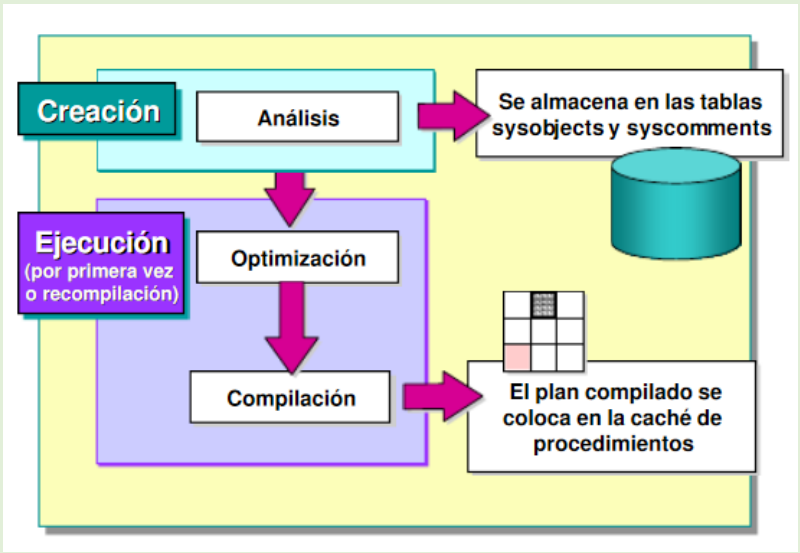
El **proceso en la ralización de un procedimiento almacenado** incluye:

1. **CREAR PROCEDIMIENTO**: **En el momento de la creación, sólo se realiza la comprobación de la sintaxis**.
   - Se almacena el **nombre del procedimiento** almacenado en la tabla de sistema **sysobjects**
   - y el **texto del procedimiento** almacenado en la tabla de sistema **syscomments** en la base de datos actual.
   - Se **devuelve un error si se encuentra un error de sintaxis y el procedimiento almacenado no es creado**.
2. **EJECUTARLO POR PRIMERA VEZ**: se **compila, se crea y se guarda en memoria** (**caché de procedimientos**) **un plan de ejecución que contiene el método más rápido de acceder a los datos.**
   - **Se resuelven todos los objetos referenciados en el procedimiento almacenado**. Ocurre algún error si se referencia a algún objeto inexistente
   - **Se utiliza el plan de ejecución generado cuando se vuelve a llamar al procedimiento sin volver a compilarlo nuevamente**.
   - **Fases**

   ✓ **Optimización**
   - Se **analizan las instrucciones TransactSQL** en el procedimiento almacenado **y crea el plan de ejución**. Para hacer esto, el optimizador de consultas considera lo siguiente:
     - La cantidad de datos en las tablas.
     - La presencia y naturaleza de los índices de las tablas y la distribución de los datos en las columnas indizadas.
     - Los operadores de comparación y los valores de comparación que son utilizados en condiciones de cláusula WHERE.
     - La presencia de combinaciones y las cláusulas UNION, GROUP BY u ORDER BY.

   ✓ **Compilación**
   - Después de que el optimizador de consultas coloca **el plan compilado en la caché de procedimientos, se ejecuta el procedimiento almacenado**.
   - **En los procesamientos posteriores del procedimiento almacenado**, **SQL Server utiliza el plan de consulta optimizado en la caché de procedimientos**.

La **caché de procedimientos** es **el área de la memoria que SQL Server utiliza para almacenar planes de consultas compilados para la ejecución de un procedimiento almacenado**. El tamaño de la caché de procedimientos fluctúa de acuerdo con los niveles de actividad. Cuando la caché de procedimientos se llena, los planes menos recientemente utilizados se eliminan para dejar espacio a los nuevos planes.



7.3 Creación de procedimientos almacenados

Hay que tener en cuenta lo siguiente al crear procedimientos almacenados:

- ✓ Los procedimientos almacenados pueden hacer referencia a **tablas, vistas, procedimientos almacenados y tablas temporales.**
- ✓ **Una instrucción CREATE PROCEDURE no puede ser combinada** con otras instrucciones SQL **en un único lote**.
- ✓ Los procedimientos **pueden no tener parámetros o tener varios**.
- ✓ Los parámetros pueden **ser de entrada** (informacion dada al procedimiento desde el exterior) o **pueden ser de salida** (información proporcionada por el procedimiento)

Sintaxis parcial

```
CREATE PROC[EDURE] nombre_procedimiento
[
{@parámetro tipo_datos} [=predeterminado ] [OUTPUT]
][,...n]
AS
instrucción_sql [,...n]
```

- **@parámetro:** Se trata de **un parámetro del procedimiento**.

  En una instrucción CREATE PROCEDURE se pueden declarar uno o más parámetros.
- **tipoDatos:** Es el **tipo de datos del parámetro**. Para los parámetros de entrada todos los tipos de datos, se pueden utilizar como un parámetro para un procedimiento almacenado.
- **Predeterminado:** **un valor predeterminado para el parámetro**. Si se define un valor predeterminado, el procedimiento se puede ejecutar sin especificar un valor para ese parámetro. El valor predeterminado debe ser una constante o puede ser NULL.
- **OUTPUT:** Indica que se trata de **un parámetro devuelto(de salida)**. El valor de esta opción puede devolverse a EXEC[UTE].

- ✓ Con **SET NOCOUNT {ON|OFF}**: Evita que **se devuelva el mensaje que muestra el recuento del número de filas afectadas**

  Si se establece SET NOCOUNT en **ON,** no se devuelve el recuento. Cuando SET NOCOUNT es **OFF,** sí se devuelve ese número.
- ✓ La función **@@ROWCOUNT** (**devuelve el número de filas afectadas**). Se actualiza incluso cuando SET NOCOUNT es ON.
- ✓ **Ejecutar un procedimiento almacenado** con **EXECUTE O EXEC**

Ejemplo: **Creación de un procedimiento sin parámetros**.

```
USE BDPERSONA
GO
CREATE PROC dbo.Persona
AS
 SET NOCOUNT ON
 SELECT Nombre, Apellido1, Apellido2, SEXO FROM Persona  WHERE SEXO='M'
GO
```

Ejecución del procedimiento. La siguiente instrucción **ejecuta el procedimiento almacenado** PersonA después de que ha sido creado.

```
EXEC dbo.Persona
```

Opciones RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION

Al la hora de crear un procedimiento almacenado se puede optar por las siguites opciones:

```
CREATE PROC[EDURE] nombre_procedimiento
[
{@parámetro tipo_datos} [VARYING] [=predeterminado ] [OUTPUT]
] [,...n]
[WITH {RECOMPILE | ENCRYPTION | RECOMPILE,ENCRYPTION}]
AS
instrucción_sql [,...n]
```

- **RECOMPILE** indica que **no se almacena en la caché un plan de ejucuión para este procedimiento**, con lo que el **procedimiento se vuelve a compilar cada vez que se ejecuta**.
- **ENCRYPTION** indica que SQL Server **codifica la entrada de la tabla syscomments** que contiene el **texto de la instrucción** CREATE PROCEDURE.

```
USE BDPERSONA
GO
CREATE PROC dbo.Persona
WITH RECOMPILE
AS
 SET NOCOUNT ON
 SELECT Nombre, Apellido1, Apellido2, SEXO FROM Persona  WHERE SEXO='M'
GO
```

7.4 Parámetros

Declarar parámetros

- ✓ La **declaración de un parámetro** requiere de **dos a cuatro** de estos fragmentos de información:
  - **El nombre**
  - **El tipo de datos**
  - **El valor predeterminados (opcional)**
  - **OUTPUT**

```
@nombre_parámetro [AS] tipo_de_datos [=valor_predeterminado|NULL] [OUTPUT|OUT]
```

- ✓ **Si se epecifica el valor predeterminado**, un **usuario puede ejecutar el procedimiento almacenado sin especificar un valor para ese parámetro**.
- ✓ Los valores predeterminados de los parámetros deben ser constantes o NULL.

Parámetros de entrada

- ✓ Los parámetros de entrada **permiten que se pase información a un procedimiento almacenado.**
- ✓ Para definir un procedimiento almacenado que acepte parámetros de entrada se declara **una o más variables como parámetros** en la instrucción CREATE PROCEDURE.

Sintaxis parcial

```
@parámetro tipo_datos [= predeterminado]
```

Ejemplo: procedimiento almacenado donde utilizamos parámetros para crear un nuevo registro en la tabla Grupo de la base de datos COCINA:

```
USE COCINA
GO
CREATE PROC spInsertarGrupo
@Codigo smallint, @Nome varchar(30)
AS
INSERT INTO GRUPO    VALUES (@Codigo, @Nome)
```

Ejecucion: Un ejemplo donde utilizamos el procedimiento almacenado para insertar en la tabla sería:

```
EXEC spInsertarGrupo 15,'MASAS'
```

Como no hemos suministrado ningún **valor predeterminado para ninguno de los parámetros, ambos se consideran requeridos, con lo cual debemos suministrar ambos parámetros o nos dará error**.

Ejemplo: Creación de un procedimiento donde se le pasa por parámetros el autor de un libro y nos visualiza los titulos de los libros de ese autor.

```
USE BIBLIOTECA
GO
CREATE PROC ObtenerTitulosDeAutor
    @autor nvarchar(20)
AS
SELECT titulo FROM libros   WHERE autor like '%'+@autor+'%'
```

Para ejecutar el procedimiento:

```
EXEC ObtenerTitulosDeAutor 'Cervantes'
```

Suministrar valores predeterminados

- ✓ Para crear un **parámetro opcional** debemos suministrar un **valor predeterminado.**
- ✓ Para ello, **sólo tenemos que añadir un signo = junto con el valor** que deseamos utilizar como predeterminado tras el tipo de datos.

Vamos a crear un ejemplo donde vamos a insertar en una tabla CHEF el código y nombre, pero el nombre va a ser opcional.

```
CREATE PROC spInsertarChef
  @Codigo smallint,  @Nome varchar(30) = NULL
AS
INSERT INTO CHEF
VALUES (@Codigo, @Nome)
```

Ahora funcionaría una llamada a este procedimiento donde no le pasamos el parámetro Nome.

```
EXEC spInsertarChef 3
```

Parámetros de salida

- ✓ Los procedimientos almacenados **pueden devolver información** al procedimiento almacenado o al cliente que les ha llamado con **parámetros de salida** (parámetros designados con la palabra clave **OUTPUT**).
- ✓ Para utilizar un parámetro de salida, la **palabra clave OUTPUT debe estar especificada en las instrucciones CREATE PROC y EXECUTE**.
- ✓ Si la palabra clave OUTPUT se omite cuando se ejecuta el procedimiento almacenado, el procedimiento almacenado se ejecuta de todas maneras, pero produce una condición de error.
- ✓ El parámetro puede ser de cualquier tipo de datos excepto texto o imagen.

Ejemplo: Vamos a crear un procedimiento almacenado que calcula el producto de dos números. Para ello vamos a declarar dos parámetros de entrada: @num1 y @num2 y un parámetro de salida @producto.

```
CREATE PROC spProducto
  @num1 smallint, @num2 smallint,
  @producto smallint OUTPUT
AS
SET @producto=@num1*@num2
GO
```

Para ejecutarlo:

```
DECLARE @resultado smallint
EXEC spProducto 10, 3, @resultado OUTPUT
SELECT 'El resultado es ', @resultado
```

Ejemplo: Vamos a hacer otro ejemplo donde vamos a insertar en una tabla PEDIDO que tiene un campo codigo Identity, Cod_cliente smallint y fechaPedido. Tenemos previsto realizar una tarea adicional utilizando el registro insertado. (Imaginemos que después fuésemos a insertar en la tabla detalle_pedido, para mantener la relación intacta debemos conocer la identidad del registro de Pedido antes de poder realizar nuestras inserciones en la tabla DetallePedido.

```
USE FERRETERIA
GO
CREATE PROC spInsertarPedido
  @Cod_cliente smallint,    @fechaPedido datetime=NULL,
  @Codigo int OUTPUT
AS
INSERT INTO Pedido
VALUES (@Cod_cliente, @fechaPedido)
/* Movemos el valor identidad desde el registro recién insertado en nuestra variable de salida */
SELECT @Codigo=@@IDENTITY
```

Ahora vamos a probar a recoger el valor que generó para la columna IDENTITY.

```
DECLARE @MiIdentidad int
EXEC spInsertarPedido 1029, @MiIdentidad OUTPUT
SELECT * FROM PEDIDO WHERE Codigo=@MiIdentidad
```

Confirmación del éxito o fallo con valores devueltos: Uso del Return

- ✓ Todos los valores devueltos indican el **éxito o el fallo del procedimiento almacenado** e incluso la extensión o naturaleza de dicho fallo.
- ✓ **De forma predeterminada**, SQL Server devueve automáticamente un valor de cero cuando se ha completado el procedimiento. Para **pasar un valor devuelto desde nuestro procedimiento almacenado al código se usa la instrucción RETURN**.

- ✓ La **instrucción RETURN sale incodicionalmente de nuestro procedimiento almacenado**. Es decir, independientemente del lugar en el que se encuentre en el procedimiento almacenado, **no se ejecutarán más líneas de código tras haber emitido una instrucción RETURN**. Su sintaxis es:

```
RETURN [<valor entero a devolver]
```

- ✓ **Puede devolver un valor de estado integer (código de retorno).**
- ✓ **Si no define un valor de retorno** definido por el usuario se devuelve **0**.

Vamos a ver un ejemplo:

```
USE COCINA
GO
CREATE PROC dbo.EsCocinero
    @cocinero  int
AS
IF NOT EXISTS(SELECT codigo FROM COCINERO  WHERE codigo = @cocinero)
  RETURN 100  --No existe este cocinero en nuestra base de datos
IF EXISTS(SELECT codigo  FROM CHEF
       WHERE codigo = @cocinero)
   RETURN 1 –- Existe y es Chef
IF EXISTS(SELECT codigo  FROM COCINERO
              WHERE cocinero= @cocinero)
   RETURN 2   –- Es cocinero pro no es Chef
```

Para capturar el valor de la instrucción RETURN, tendremos que asignarlo a una variable durante la instrucción EXEC. Por ejemplo:

```
DECLARE @valorRETURN int
EXEC @valorRETURN= EsCocinero 10
  SELECT @valorRETURN
```

Ejemplo: Procedimiento que selecciona todas las filas de la tabal socios y también de la tabla libros.

```
CREATE PROC CuentaSociosyLibros
   @CuentaSocios int OUTPUT,    @CuentaLibros int OUTPUT
AS
 SELECT * FROM Socios
 SELECT @CuentaSocios=@@ROWCOUNT
 SELECT * FROM Libros
 SELECT @CuentaLibros=@@ROWCOUNT
```

Después, el procedimiento se ejecutaría de la siguiente forma:

```
DECLARE @CuentaLib int, @CuentaSoc int
EXEC CuentaSociosyLibros @CuentaLib OUTPUT, @CuentaSoc OUTPUT
select Total_Libros=@CuentaLib, @CuentaSoc  as Total_Socios
```

7.5 Ejecutar un procedimiento almacenado

- ✓ Para **ejecutar un procedimiento almacenado** se utiliza instrucción **EXECUTE (o EXEC)** junto con el **nombre del procedimiento** almacenado y **los parámetros** (sin tienen).

Sintaxis

```
[EXEC[UTE]]
{
[@return_status =] {nombre_procedimiento
 | @nombre_procedimiento_var}
}
[[@parámetro =] {(valor | @variable [OUTPUT] | [DEFAULT]] [, ...n] }
[WITH RECOMPILE]
```

- ✓ **EXECUTE ……WITH RECOMPILE: crea un nuevo plan de ejecución** durante la ejecución del procedimiento almacenado. **El nuevo plan de ejecución se remplaza en la caché**.

Nota: *Si la instrucción de ejcutar el procedimiento es la primera línea de un lote, se puede omitir la la palabra EXEC[UTE], es decir, especificando solo el nombre y parámetros.*

*No obstante, **siempre debería utilizar la instrucción EXEC[UTE] para evitar que ocurran errores** si introduce otra línea al principio del lote. Si intenta ejecutar un procedimiento almacenado en otra línea que no sea la primera línea del lote y no especifica la instrucción EXEC[UTE], se devolverá un error «Sintaxis incorrecta».*

Establecer el valor de un parámetro en un procedimiento almacenado

Puede **establecer el valor de un parámetro** pasando el valor al procedimiento almacenado mediante **referencia o posición**. No se debería mezclar los distintos formatos cuando se proporcionen los valores.

Pasar valores mediante referencia

- ✓ El especificar un parámetro en una instrucción EXECUTE en el formato **@parámetro = valor** se denomina pasar mediante **referencia**.
- ✓ Cuando se pasa los valores mediante referencia, **los valores del parámetro pueden ser especificados en cualquier orden, y puede omitir parámetros que tengan un valor predeterminado.**

Sintaxis:

```
[EXEC[UTE]] nombre_procedimiento [@parámetro = valor] [,...n]
```

Ejemplo: El procedimiento almacenado Trabajadores devuelve el numero de empleados de que hay en una determinada oficina pasada por parámetros

```
USE Gestion
GO
CREATE PROC trabajadores
  @ofi INT, @num INT OUTPUT
  AS
   SELECT @num=(SELECT COUNT(*) FROM  empleados WHERE oficina=@ofi)
GO
```

Para ejecutar el procedimiento, por referencia:

```
DECLARE @cod int
EXEC trabajadores @ofi = 3, @num = @cod OUTPUT
```

Sugerencia: La mayoría de los lenguajes de programación utilizan el término pasar mediante referencia de una manera diferente que SQL Server. El pasar los parámetros por referencia en SQL Server no es lo mismo que el significado habitual del término, el cual es pasar una referencia o puntero a un parámetro.

Pasar valores mediante posición

- ✓ Pasar valores (**sin referencia a los nombres de los parámetros**) se denomina pasar valores mediante **posición**.
- ✓ Cuando sólo se especifican valores, deben **estar listados en el orden en el cual están definidos en la instrucción CREATE PROC.**
- ✓ Cuando pase valores mediante posición, **puede omitir parámetros cuando existan valores predeterminados, pero no puede interrumpir la secuencia**. Por ejemplo, si un procedimiento almacenado tiene cinco parámetros, puede omitir el cuarto y quinto parámetros, pero no puede omitir el cuarto parámetro y especificar el quinto.

Sintaxis parcial

```
[EXEC[UTE]] nombre_procedimiento [valor] [,...n]
```

Ejemplo

La siguiente secuencia de comandos pasa valores mediante posición al procedimiento almacenado trabajadores sería

```
DECLARE @cod int
EXEC trabajadores 3, @cod OUTPUT
```

Cómo anidar procedimientos almacenados

- ✓ Los procedimientos almacenados **pueden ser anidados**, que es cuando un procedimiento almacenado llama a otro.
- ✓ Las características de la anidación incluyen las siguientes:
  - Los procedimientos almacenados pueden ser anidados **hasta 32 niveles**.
  - El **nivel actual de anidación** es devuelto por la función **@@NESTLEVEL.**
- ✓ Si **un procedimiento almacenado llama a otro procedimiento** almacenado, el **segundo procedimiento almacenado puede acceder a todos los objetos que son creados por el primer procedimiento** almacenado, incluyendo las tablas temporales.

7.6 Ver información acerca de procedimientos almacenados

Se puede utilizar los procedimientos almacenados de sistema en la siguiente tabla para encontrar información adicional acerca de todos los tipos de procedimientos almacenados.

Procedimiento almacenado	Información
sp_help nombre_procedimiento	Muestra una lista de parámetros y sus tipos de datos para el procedimiento almacenado especificado.
sp_helptext nombre_procedimiento	Muestra el texto del procedimiento almacenado especificado si no está cifrado.
sp_depends nombre_procedimiento	Lista los objetos que dependen del procedimiento almacenado especificado y los objetos sobre los que depende el procedimiento almacenado.

7.7 Modificar y eliminar procedimientos almacenados

Modificar procedimientos almacenados

Para **modificar un procedimiento almacenado existente** y mantener la asignación de permisos, se utiliza la instrucción **ALTER PROCEDURE**.

Sintaxis:

```
ALTER PROC[EDURE]  nombre_procedimiento
[
{@parámetro tipo_datos} [VARYING] [=predeterminado ] [OUTPUT]
][,...n]
[WITH {RECOMPILE | ENCRYPTION | RECOMPILE,ENCRYPTION}]
AS
instrucción_sql [,...n]
```

Ejemplo:

```
ALTER PROC sp_cuentaVivienda @tipovivienda varchar(6)
AS
SELECT COUNT(*)  FROM VIVIENDA
WHERE tipo = @tipovivienda
GO
```

Cómo eliminar procedimientos almacenados

- ✓ La instrucción **DROP PROCEDURE** se utiliza para **eliminar procedimientos almacenados** definidos por el usuario de la base de datos actual.
- ✓ Antes de eliminar un procedimiento almacenado, hay que ejecutar el procedimiento almacenado **sp_depends** para determinar si hay objetos que dependen del procedimiento almacenado que quiere eliminar. S

Sintaxis

```
DROP PROC[EDURE] procedimiento [, ...n]
```

Ejemplo: Se elimina el procedimiento almacenado **ObtenerTitulosDeAutor**

```
USE BIBLIOTECA
GO
DROP PROC ObtenerTitulosDeAutor
```

8. Funciones definidas por el usuario.

- ✓ Son rutinas que **aceptan parámetros**, **realizan una acción** y **devuelven el resultado de esa acción como un valor**.
- ✓ El **valor devuelto** puede ser un **valor escalar único** o **un conjunto de resultados (tabla)**, por lo tanto tenemos **dos tipos de funciones definidas por el usuario**:
  - **Las que devuelven un valor escalar (funciones escalares)**
  - **Las que devuelven una tabla (funciones inline).**
- ✓ Una función definida por el usuario es muy parecida a un procedimiento almacenado, la **diferencia principal entre ellos se encuentra en el modo en que se devuelven los resultados**.

8.1 CREATE FUNCTION

- ✓ Crea una función definida por el usuario, que es una **rutina guardada de Transact-SQL** que **devuelve un valor** (un **valor escalar o una tabla**).

Sintaxis:

```
CREATE FUNCTION nombre_función
(
  -- Lista de parámetros de entrada
[ { @parameteros [AS] tipodedatosescalar [ = default ] } [ ,...n ] ] )
    -- Tipo de datos que devuelve la función.
RETURNS tipodedatosdevuelto
 [ AS ]
BEGIN
        Cuerpo de la funcion
 RETURN expression
END
```

Argumentos

- **Nombre_funcion:** Nombre de la función definida por el usuario. Los nombres de funciones deben seguir las reglas de los identificadores.
- **@parametros:** Valores de entrada a la función**.** Se pueden declarar uno o varios parámetros.
- **Tipodedatosdevuelto:** Es el tipo de valor de devuelto de una función. Puede ser de cualquiera de los tipos de datos escalares compatibles con SQL Server (excepto text, ntext, image y timestamp)
- **Cuerpo de la función:** Especifica que una serie de instrucciones Transact-SQL.

Funciones escalares

Las **funciones escalares** devuelven un valor escalar, eso quiere decir, que **devuelve un valor único**.

Ejemplo: Esta instrucción crea una función sencilla que devuelve un decimal:

```
CREATE FUNCTION Volumen
      -- Dimensiones en centímetros
  (@Largo decimal(4,1),
   @Ancho decimal(4,1),
   @Alto decimal(4,1) )
RETURNS decimal(12,3) – tipo de datos que devuelve la función
AS
BEGIN
  RETURN ( @Largo * @Ancho * @Alto )
END
```

Funciones inline

- ✓ Las **funciones inline** son las funciones que devuelven **un conjunto de resultados correspondientes a la ejecución de una sentencia SELECT**.
- ✓ **No se puede utilizar la clausula ORDER BY en la sentencia de una función el línea**

Ejemplo de creación de una función de tabla inline que permite conocer los artículos que poseen un precio inferior al tomado como parámetro:

```
CREATE FUNCTION ArticulosBajoPrecio (@maximo int)
 RETURNS TABLE
AS
BEGIN
  RETURN (SELECT * FROM ARTICULOS WHERE PRECIO<@maximo)
END
```

Funciones inline de múltiples sentencias

Éste tipo de función es muy similar al anterior, con la diferencia que está enfocado a implementar lógicas más fuertes, que requieran varias consultas SELECT para armar el resultado que deseamos que devuelva.

```
CREATE FUNCTION fx_insertarPersonas
( @Codigo int, @Edad int)
 RETURNS @credito TABLE
  (Codigo int primary key,
      NombreApellido varchar(50),
      Edad int )
AS
```

```
BEGIN
   INSERT INTO @credito
   SELECT Codigo, Nombre + " "+Apellido, datediff(year, P.FechaNacimiento, getdate())
   FROM Persona
   WHERE datediff(year, P.FechaNacimiento, getdate()) > @Edad

  INSERT INTO @credito
  SELECT Codigo, Nombre + Apellido, datediff(year, P.FechaNacimiento, getdate())
   FROM Persona
   WHERE Codigo = @Codigo;

    RETURN
END
GO
```

8.2 Llamar a funciones definidas por el usuario

- ✓ La **funciones** definidas por el usuario **pueden ser utilizadas en cualquier sentencia Transact SQL**.
- ✓ Cuando **se llama una función** definida por el usuario, debe **especificarse el nombre de la función** seguido de **paréntesis** y deben proporcionarse **valores de argumentos para todos los parámetros** en la misma secuencia en que están definidos los parámetros en la instrucción CREATE FUNCTION.
- ✓ Tanto las funciones definidas por el usuario, como las funciones de sistema, se pueden llamar desde una consulta. También se pueden ejecutar mediante una instrucción EXECUTE como los procedimientos almacenados.

  - **Las funciones de valores escalares** se pueden llamar **en aquellos lugares donde se utilizan expresiones escalares**, incluidas las columnas calculadas y las definiciones de restricciones CHECK.

    Por ejemplo, si se define una función llamada **fn_MiEntero** que devuelve un entero con un parámetro entero y un parámetro nchar(20), se puede invocar utilizando:

```
SELECT *  FROM Tabla  WHERE Clave = dbo.fn_MiEntero( 1, 'Ana' )
```

Es posible utilizar la función Volumen que creamos anteriormente en cualquier parte en la que se permita una expresión decimal, como en una columna calculada de una tabla:

```
CREATE TABLE Envase
  (
   ID   int PRIMARY KEY,
   Color    varchar(20),
   Alto     decimal(4,1),
   Largo    decimal(4,1),
   Ancho    decimal(4,1),
   Volumen AS    dbo.Volumen(Largo, Ancho, Alto )
  )
```

  - **Las funciones inline** se puede invocar una función definida por el **usuario que devuelve una tabla**, donde se permiten **expresiones de tabla** en la cláusula **FROM** de instrucciones SELECT, INSERT, UPDATE o DELETE.

```
SELECT * FROM  dbo. fx_insertarPersonas (1, 30)
```

8.3 ALTER FUNCTION

Para modificar una función creada anteriormente por la ejecución de la instrucción CREATE FUNCTION, sin cambiar los permisos y sin que afecte a ninguna otra función, procedimiento almacenado o desencadenador dependientes tenemos la función **ALTER TABLE**.

```
ALTER FUNCTION nombre_función
(
  -- Lista de parámetros
[ { @parameteros [AS] tipodedatosescalar [ = default ] } [ ,...n ] ] )
    -- Tipo de datos que devuelve la función.
RETURNS tipodedatosdevuelto
[ AS ]
BEGIN
 Cuerpo de la funcion
 RETURN expression
END
```

Por ejemplo:

```
ALTER FUNCTION ArticulosBajoPrecio (@maximo int)
RETURNS TABLE
AS
  RETURN (SELECT * FROM ARTICULOS WHERE PRECIO<@maximo AND tipo='A')
```

8.4 DROP FUNCTION

Elimina una o más funciones definidas por el usuario de la base de datos actual.

Sintaxis

```
DROP FUNCTION nombre_funcion  [ ,...n ]
```

8.5 Obtener información acerca de las funciones

Diversos objetos de catálogo proporcionan información acerca de las funciones definidas por el usuario:

- **sp_help** proporciona información acerca de las funciones definidas por el usuario.
- **sp_helptext** proporciona información acerca del origen de las funciones definidas por el usuario.

Existen vistas de esquemas que proporcionan información acerca de las funciones definidas por el usuario: ROUTINES y PARAMETERS. Estas vistas de esquemas de información también informan acerca de los procedimientos almacenados.