TEMA 5. PROGRAMACIÓN DE BASES DE DATOS
(Parte I - Procedimientos Almacenados)

1. PROCEDIMIENTOS ALMACENADOS EN Microsoft SQL Server

2. DESENCADENADORES O TRIGGERS EN Microsoft SQL Server

1. PROCEDIMIENTOS ALMACENADOS EN Microsoft SQL Server

DEFINICIÓN

- Los procedimientos almacenados son una serie de **sentencias Transact-SQL que son posible llamarlos mediante un identificador** y **pueden recibir** y **devolver valores**.
- Hay **procedimientos almacenados que vienen definidos por el propio sistema** (procedimientos almacenados de sistema) y **otros que los define el propio usuario** (procedimientos almacenados de usuario).
- Dentro de un procedimiento almacenado se puede:
  - Definir variables: DECLARE @NOMBRE TIPO
  - Utilizar estructuras de control de flujo.
  - Llamar a otro procedimiento almacenado.
  - Devuelven un valor de estado (status value) comprendidos entre el 0 y -14, por ejemplo el valor 0 es ejecución con éxito.

(Ver " lenguaje de control de flujo" en enlace Manual de Referencia de Transact-SQL)

PRINCIPALES VENTAJA

- Se **optimizan** en el momento de su creación, por lo tanto mejora el rendimiento de las consultas.
- Están **compilados y almacenados en el servidor** para que su ejecución sea más rápida, que el conjunto de sentencias individuales que lo integran.

CREACIÓN

La sentencia Transact-SQL que permite crear procedimientos almacenados es CREATE PROCEDURE. Una sintaxis reducida es la siguiente:

```
CREATE PROCEDURE nombre_procedimiento
 [ {@parámetro tipoDatos }[= predeterminado] [OUTPUT] ]  [ , …n]
AS
[BEGIN]
        instrucciónSQL [...n]
        [RETURN N]
[END]
```

Parámetros

[{*@parámetro tipoDatos* }[**=** *predeterminado*] [OUTPUT]]

Ejemplo:

```
CREATE PROCEDURE calcular @p1 int=1, @p2 char, @p3 varchar(8)='Tomas' AS ...
```

En la llamada existen muchas posibilidades, veamos algunas:

```
EXEC calcular @p2='A'
EXEC calcular 3,'B'
EXEC calcular 3, 'B', 'JAVIER'
EXEC calcular @p1=DEFAULT, @p2='D'
```

Retorno de información

Los procedimientos almacenados pueden retornar información al procedimiento que los llamó o al usuario de **tres modos** diferentes:

- ✔ **Mediante el retorno de un valor de estado**.
- ✔ **Mediante parámetros de retorno o de salida** (Cláusula **OUTPUT**).
- ✔ Mediante la cláusula **RETURN**.

EJECUCIÓN

[EXEC[UTE]]
[@*valor de retorno* **=**] *nombreProcedimiento*
[[@*parámetro* **=**] {*valor* | @*variable* [OUTPUT] | [DEFAULT] }] [**,**…*n*]

EXEC o EXECUTE es necesario en los siguientes casos:

- ✔ Si llamamos al procedimiento desde otro procedimiento almacenado, desde un trigger, desde el código de un programa donde la llamada se encuentre inmersa.
- ✔ Si queremos obtener un valor de estado.
- ✔ Si tiene parámetro de salida.
- ✔ Si hemos utilizado la cláusula RETURN.

Ejemplos:

Creación del procedimiento:

```
CREATE PROCEDURE INFOEMPLEADOS AS

    SELECT * FROM TEMPLE
```

Ejecución del procedimiento:

```
EXEC INFOEMPLEADOS
```

Borrado de procedimientos almacenados

```
    DROP PROCEDURE nombre_procedimiento
```

Cambiar nombre a un procedimiento almacenado

```
    sp_rename nombre_antiguo, nombre_nuevo
```

Aquí utilizamos un procedimiento almacenado (store procedure) del sistema.

Veamos un ejemplo de **procedimiento almacenado simple** que se encuentra en "Ej1 pa simple.sql". El archivo pertenece a: EJEMPLOS_PROCEDIMIENTOS_ALMACENADOS.

```
--Creación de un procedimiento almacenado sencillo.

/*Deseamos conocer para todos los empleados su número, nombre y el
nombre del departamento en el que se encuentra.*/
USE empresa;

CREATE PROCEDURE p1
AS
BEGIN

SELECT  numem,nomem,nomde
FROM tdepto d  RIGHT JOIN temple e ON (e.numde=d.numde)

END;

--EJECUCIÓN.
p1;

EXEC p1;

EXECUTE p1;

--Podríamos consultar el valor de estado.
--Podremos ver el valor devuelto en la pestaña Messages

DECLARE @valor INT
EXECUTE @valor= p1
PRINT @valor
```

Veamos unos ejemplos de **procedimientos almacenados con un parámetro de entrada** que se encuentran en "Ej2 pa con parametro de entrada.sql". El archivo pertenece a: EJEMPLOS_PROCEDIMIENTOS_ALMACENADOS.

```
/*Creación de un procedimiento almacenado con un parámetro de
entrada.*/

/*Deseamos saber cuáles son los empleados de un departamento que se
pasa como parámetro*/
USE empresa;

CREATE PROCEDURE p2 @NOMBREDEPARTAMENTO VARCHAR(50)
AS
BEGIN

SELECT numem,nomem
FROM tdepto d JOIN  temple e ON (d.numde=e.numde)
WHERE  d.nomde = @NOMBREDEPARTAMENTO

END;

--EJECUCIÓN
p2 'NOMINAS';

EXECUTE p2 'SECTOR SERVICIOS';

EXEC p2 @NOMBREDEPARTAMENTO='PERSONAL';

-- Así nos daría error, pues espera un parámetro de entrada
p2;

DROP PROCEDURE p2;
```

```
/*Creación de un procedimiento almacenado con un parámetro de entrada
al que se le asigna un valor por defecto.*/
CREATE OR ALTER PROCEDURE p2 @NOMBREDEPARTAMENTO VARCHAR(50)='SECTOR
SERVICIOS'
AS
BEGIN

SELECT  numem,nomem
FROM tdepto d JOIN temple e ON ( d.numde=e.numde)
WHERE d.nomde = @NOMBREDEPARTAMENTO

END;

--EJECUCIÓN.
p2 'NOMINAS';

p2 @NOMBREDEPARTAMENTO='PERSONAL';

p2;

p2 @NOMBREDEPARTAMENTO=DEFAULT;

EXEC p2 DEFAULT;

--Podríamos consultar el valor de estado.
DECLARE @valor INT;
EXECUTE @valor= p2;
PRINT @valor;
```

Veamos unos ejemplos de **procedimientos almacenados con parámetros de entrada y salida** que se encuentran en “Ej3 pa con parametros de entrada y salida.sql”. El archivo pertenece a: EJEMPLOS_PROCEDIMIENTOS_ALMACENADOS.

```
/*Creación de un procedimiento almacenado con un parámetro de entrada y
un parámetro de salida.*/

/*Deseamos saber cuántos empleados tiene un departamento que se pasa
como parámetro.*/
USE empresa;

CREATE PROCEDURE p3 @NOMBREDEPARTAMENTO VARCHAR(50), @NUMERO INT OUTPUT
AS
BEGIN

SELECT @NUMERO = COUNT(*)
FROM temple e JOIN tdepto D ON (e.numde=d.numde)
WHERE d.nomde LIKE @NOMBREDEPARTAMENTO

END;

--EJECUCIÓN.
DECLARE @NUM INT;
EXEC p3 'SECTOR SERVICIOS', @NUM OUTPUT;
PRINT @NUM;

DECLARE @NUM INT;
EXEC p3 'NOMINAS',@NUM OUTPUT;
PRINT @NUM;
```

```
/*Creación de un procedimiento almacenado con un parámetro de entrada y
dos parámetros de salida.*/

/*Deseamos saber cuántos empleados tiene un departamento que se
pasa como parámetro y la edad de su empleado más joven.*/
CREATE OR ALTER PROCEDURE p3v1 @NOMBREDEPARTAMENTO VARCHAR(50),
@NUM_EMP INT OUTPUT,@MIN_EDAD  INT  OUTPUT
AS
BEGIN

SELECT @NUM_EMP = COUNT(*),
       @MIN_EDAD=MIN(DATEDIFF(DAY,fecna,GETDATE())/365)
FROM temple e JOIN tdepto d ON (e.numde=d.numde)
WHERE d.nomde LIKE @NOMBREDEPARTAMENTO

END;

--EJECUCIÓN.
DECLARE @NUM INT, @EDAD INT;
EXEC p3v1 'SECTOR SERVICIOS', @NUM OUTPUT,@EDAD OUTPUT;
PRINT @NUM;
PRINT @EDAD;

--Podríamos consultar el valor de estado.
DECLARE @return_status INT;
DECLARE @NUM INT;
EXECUTE @return_status = p3 'NOMINAS',@NUM OUTPUT;
PRINT @return_status;
PRINT @NUM;
```

Veamos unos ejemplos de **procedimientos almacenados que devuelven un valor con la cláusula RETURN** que se encuentra en “Ej4 pa con return.sql”. El archivo pertenece a: EJEMPLOS_PROCEDIMIENTOS_ALMACENADOS.

```
/*Creación de un procedimiento almacenado que devuelve un valor con la
cláusula  RETURN.*/

--RETURN devuelve un valor entero.

/*Este procedimiento almacenado devuelve un 1 si la media de los
salarios de un departamento pasado por parámetro supera los 1800 euros,
en caso contrario devuelve un 2.*/
USE empresa;

CREATE PROCEDURE p4 @NOMBREDEPARTAMENTO VARCHAR(50)
AS
BEGIN

IF  (SELECT AVG(salar) FROM tdepto d JOIN temple e ON (d.numde=e.numde)
     WHERE nomde LIKE @NOMBREDEPARTAMENTO) > 1800
BEGIN
  PRINT 'MAYOR QUE 1800'
  RETURN 1
END
ELSE
BEGIN
  RETURN 2
END

END;

--EJECUCIÓN
DECLARE @VALOR INT;
EXEC @VALOR=p4 'NOMINAS';
PRINT @VALOR;

DECLARE @VALOR INT;
EXEC @VALOR=p4 'ORGANIZACION';
PRINT @VALOR;
```

```
/*Este procedimiento no da error en la creación, aunque devolvamos un
valor que no sea entero, sin embargo, observa que el valor devuelto lo
convierte a entero.*/
CREATE OR ALTER PROCEDURE p4v1 @NOMBREDEPARTAMENTO VARCHAR(50)
AS
BEGIN

DECLARE @MEDIA FLOAT

SELECT @MEDIA=AVG(salar) FROM tdepto d JOIN temple e ON
(d.numde=e.numde)
     WHERE nomde LIKE @NOMBREDEPARTAMENTO

PRINT 'LA MEDIA ES'

RETURN @MEDIA

END;

--EJECUCIÓN.
DECLARE @VALOR FLOAT;
EXEC @VALOR=P4V1 'ORGANIZACION';
PRINT @VALOR;

--Comprobación:
SELECT AVG(salar) FROM tdepto d JOIN temple e ON (d.numde=e.numde)
WHERE nomde LIKE 'ORGANIZACION';
```

Veamos un ejemplo de **procedimiento almacenado que llama a otro procedimiento almacenado** que se encuentra en "EJ5 pa que llama al pa del EJ4.sql". El archivo pertenece a: EJEMPLOS_PROCEDIMIENTOS_ALMACENADOS.

```
/*Este procedimiento almacenado llama al procedimiento almacenado p4, y
dependiendo del valor devuelto, muestra  un determinado mensaje.*/

USE empresa;

CREATE PROCEDURE p5 @NOMBREDEPARTAMENTO VARCHAR(50)
AS
BEGIN

DECLARE @VALOR_RETORNO INT
EXEC @VALOR_RETORNO = p4 @NOMBREDEPARTAMENTO

IF (@VALOR_RETORNO = 1)
BEGIN
      PRINT 'LA MEDIA DE LOS SALARIOS DEL DEPARTAMENTO ' +
@NOMBREDEPARTAMENTO + ' SUPERA LOS 1800 EUROS'
END

ELSE
BEGIN
      PRINT 'NO LA SUPERA'
END

END;

--Comprobaciones:
SELECT d.nomde, AVG(salar)
FROM tdepto d JOIN temple e ON (d.numde=e.numde)
GROUP BY d.nomde;

--EJECUCIÓN.
EXEC p5 'NOMINAS';
EXEC p5 'ORGANIZACION';
```

```
--Borrado de Procedimientos almacenados.
DROP PROCEDURE p1;
DROP PROCEDURE p2, p3, p4, p5;
DROP PROCEDURE p3v1, p4v1;
```

CURSORES EN Microsoft SQL Server

CONCEPTO

Un cursor es una estructura que permite recorrer fila a fila un conjunto de resultados en SQL Server. Se usa cuando se necesita aplicar una lógica personalizada a cada fila individualmente.

USO

Se usan:

- Cuando no es posible usar operaciones en bloque como UPDATE o JOIN.
- Cuando se necesita realizar validaciones o transformaciones personalizadas por fila.

TIPOS

CURSOR ESTÁTICO:
Rellena el conjunto de resultados durante la creación del cursor y el resultado de la consulta se almacena en caché durante la vida útil del cursor. Un cursor estático puede moverse hacia adelante y hacia atrás.

FAST_FORWARD (tipo de cursor predeterminado)
Es idéntico al estático excepto que solo puede desplazarse hacia adelante.

Las adiciones y eliminaciones DINÁMICAS
son visibles para otros usuarios de la fuente de datos mientras el cursor está abierto. A diferencia de los cursores estáticos, todos los cambios realizados en el cursor dinámico reflejarán los datos originales. Puede usar este cursor para realizar operaciones de INSERTAR, ELIMINAR y ACTUALIZAR.

CONJUNTO DE LLAVES:
Es como un cursor dinámico, pero no podemos ver los registros que otros usuarios añaden. Si otro usuario elimina un registro, no se podrá acceder a él desde nuestro conjunto de registros.

ESTRUCTURA

Estructura básica de un cursor:

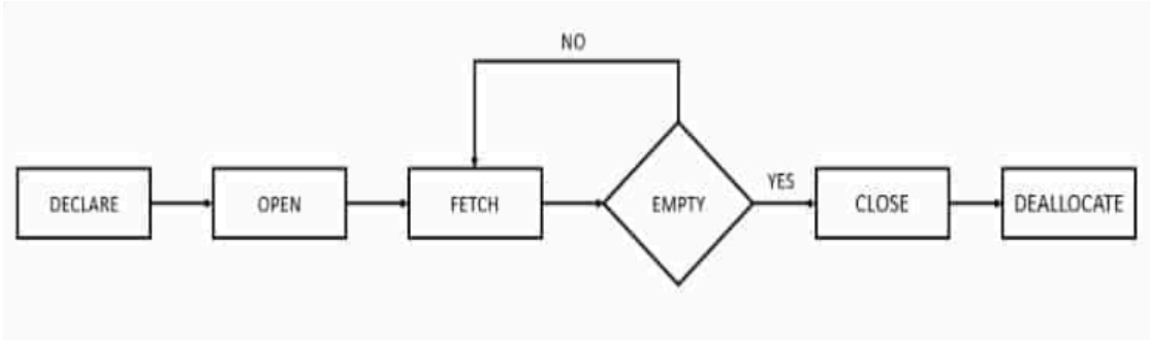
```
DECLARE nombre_cursor CURSOR FOR
SELECT columnas FROM tabla;

OPEN nombre_cursor;
FETCH NEXT FROM nombre_cursor INTO @variable;

WHILE @@FETCH_STATUS = 0
BEGIN
    -- Acciones por fila
    FETCH NEXT FROM nombre_cursor INTO @variable;
END;

CLOSE nombre_cursor;
DEALLOCATE nombre_cursor;
```

CICLO DE VIDA DE UN CURSOR SQL Server



VENTAJAS

Permiten un control detallado fila a fila.

INCONVENIENTES

- Son menos eficientes que las operaciones en bloque.
- Mayor consumo de recursos.

ALTERNATIVAS MÁS EFICIENTES

- ROW_NUMBER()
- Operaciones con JOIN, UPDATE, etc.
- Tablas temporales.

Veamos un ejemplo que se encuentra en “Ejemplo_Cursor.sql”. En primer lugar, deberás ejecutar el script llamado “Script BD cursores.sql”. Los archivos pertenecen a: EJEMPLOS_PROCEDIMIENTOS_ALMACENADOS.

Ejemplo: Supongamos que estás desarrollando una nueva versión de una aplicación que usa una base de datos de albaranes. Antes de hacer cambios peligrosos (como alterar estructuras o cargar datos nuevos), quieres guardar una copia exacta de las tablas originales.

Vamos a hacer lo que nos pide de dos maneras: con cursores y sin cursores.

```
 --No olvides ejecutar "Script BD cursores.sql"

USE albaran_ej_cursores;

--Ejemplo: Con cursores
/*El siguiente ejemplo cambia el nombre de todas las tablas en la
base de datos, agregando '_Backup' al nombre de cada tabla y al mismo
tiempo garantizando que las tablas que contienen '_Backup' en su
nombre no se renombrarán nuevamente ejecutando el siguiente código:*/

/*Puesto que vamos a renombrar las tablas, desactivamos restricciones
de clave externa*/
EXEC sp_MSforeachtable 'ALTER TABLE ? NOCHECK CONSTRAINT ALL';
```

```
CREATE OR ALTER PROCEDURE proc1
AS

BEGIN

      DECLARE @NombreTabla VARCHAR(128),
              @NombreTablaNueva VARCHAR(128);
--Declaramos el cursor
      DECLARE cursor1 CURSOR FOR
      SELECT T.TABLE_NAME
      FROM INFORMATION_SCHEMA.TABLES T
      WHERE TABLE_CATALOG = 'albaran_ej_cursores';

/*INFORMATION_SCHEMA.TABLES es una vista del sistema. Aparece en la
base de datos/Views/System Views*/
/*Abrimos el cursor, con ello se prepara el conjunto de resultados para
ser recorrido fila a fila.
Es como "preparar el puntero al principio del conjunto".
El cursor ahora apunta justo antes de la primera fila.*/
      OPEN cursor1;

--El valor del campo TABLE_NAME de esa fila se copia en @NombreTabla.
      FETCH NEXT FROM cursor1 INTO @NombreTabla;

--@@FETCH_STATUS = 0 Indica que FETCH fue exitoso y hay más filas.
      WHILE @@FETCH_STATUS = 0
      BEGIN
--RIGHT Extrae los 6 últimos carácteres a la derecha de la cadena
           IF RIGHT(@NombreTabla, 6) <> 'Backup'
            BEGIN
                 SET @NombreTablaNueva = @NombreTabla + '_Backup'
                 EXEC sp_rename @NombreTabla, @NombreTablaNueva;
                 PRINT 'Tabla renombrada: ' + @NombreTabla + ' a ' +
@NombreTablaNueva;
           END
           ELSE
           BEGIN
                 PRINT 'Ya existe la tabla:  No se renombró: ' +
@NombreTabla;
           END
           FETCH NEXT FROM cursor1 INTO @NombreTabla;
      END
```

```
--Cerramos el cursor
     CLOSE cursor1;
--Liberamos memoria
     DEALLOCATE cursor1;
END;

EXEC proc1;

--Ejemplo: Procedimiento optimizado sin cursores
/*En lugar de usar CURSOR, usaremos una tabla temporal y un bucle WHILE
con TOP  1, que suele ser más eficiente.
*/
CREATE OR ALTER PROCEDURE proc1
AS
BEGIN

   DECLARE @NombreTabla VARCHAR(128),
           @NombreTablaNueva VARCHAR(128);

/* Creamos tabla temporal con nombres de todas las tablas.
Al poner # delante de la tabla indicamos que es una tabla real, si no
al acabar el procedimiento la tabla temporal desaparece y da error.
Además, ten encuenta que SQL Server no muestra las tablas temporales en
INFORMATION_SCHEMA.TABLES.
*/

-- Eliminamos la tabla temporal si ya existe
    IF OBJECT_ID('tempdb..#tabla_temporal') IS NOT NULL
        DROP TABLE #tabla_temporal;

/* Obtenemos todos los nombres de las tablas, sin excluir las que ya
terminan en 'Backup'*/
    SELECT TABLE_NAME AS NombreTabla
    INTO #tabla_temporal
    FROM INFORMATION_SCHEMA.TABLES
    WHERE TABLE_TYPE = 'BASE TABLE';
```

```
-- Procesamos el nombre de cada tabla sin usar cursores
    WHILE EXISTS (SELECT 1 FROM #tabla_temporal)
    BEGIN
        SELECT TOP (1) @NombreTabla = NombreTabla FROM #tabla_temporal;

-- Si ya termina en 'Backup', no renombramos, solo informamos
        IF RIGHT(@NombreTabla, 6) = 'Backup'
        BEGIN
            PRINT 'Ya existe la tabla con sufijo _Backup: ' +
@NombreTabla + '. No se renombró.';
        END
        ELSE
        BEGIN
            SET @NombreTablaNueva = @NombreTabla + '_Backup';

-- Comprobamos si ya existe la tabla con el nuevo nombre
            IF NOT EXISTS (
                SELECT 1 FROM INFORMATION_SCHEMA.TABLES
                WHERE TABLE_NAME = @NombreTablaNueva
            )
            BEGIN
                EXEC sp_rename @NombreTabla, @NombreTablaNueva;
                PRINT 'Tabla renombrada: ' + @NombreTabla + ' a ' +
@NombreTablaNueva;
            END

        END

        DELETE FROM #tabla_temporal WHERE NombreTabla = @NombreTabla;
    END
END;

EXEC proc1;
```

CONCLUSIÓN

El procedimiento proc1 hecho con **tabla temporal** (sin cursores) tiene **mayor rendimiento** que el hecho con **cursores**. Veamos por que:

Con cursores

- Recorre las tablas **una a una**, usando FETCH NEXT.
- SQL Server **no optimiza bien operaciones secuenciales** con cursores.
- Es más **lento** y **consume más recursos** (memoria y CPU) en bases de datos con muchas tablas.
- No aprovecha el **procesamiento en conjunto** que SQL Server está diseñado para hacer eficientemente.

Sin cursores (con tabla temporal)

- Usa una **tabla temporal con todos los nombres desde el principio**.
- Opera con un WHILE que borra fila a fila, pero la obtención y filtrado de datos es **por conjunto**.
- SQL Server puede optimizar mejor las consultas con TOP, DELETE, EXISTS.
- Es **más rápido y escalable**, incluso con cientos de tablas.
- Requiere **menos recursos por iteración**.

Por tanto este procedimiento almacenado desarrollado utilizando tabla temporal tiene un rendimiento más alto, la codificación es más clara y el consumo de recursos es más eficiente.

Por último, antes de concluir esta parte del tema, veamos la función ROW_NUMBER(). Para ello, veamos un ejemplo de su funcionamiento. El ejemplo que se encuentra en "Ejemplo_ROW_NUMBER.sql".El archivos pertenecen a:
EJEMPLOS_PROCEDIMIENTOS_ALMACENADOS.

```
--Función ROW_NUMBER()

/*La función ROW_NUMBER() asigna un número de fila
secuencial (1, 2, 3, ...) a cada fila de un conjunto de resultados.
Se utiliza junto con la cláusula OVER(ORDER BY ...) para indicar
en qué orden se numeran las filas. Es decir, define el criterio
del orden.*/
--Sintaxis:
ROW_NUMBER() OVER (ORDER BY columna)

--Ejemplo:
USE albaran_ej_cursores;

CREATE TABLE empleados (
    Id INT,
    Nombre VARCHAR(50),
    Sueldo INT
);

INSERT INTO empleados VALUES
(1, 'Ana', 2000),
(2, 'Luis', 2500),
(3, 'Eva', 3000),
(4, 'Carlos', 2500);

/*Obtendremos los empleados números del 1 a 4, de modo
que le asigna el 1 al empleado con mayor sueldo*/
SELECT
    Nombre,
    Sueldo,
    ROW_NUMBER() OVER (ORDER BY Sueldo DESC) AS Posicion
FROM empleados;

DROP TABLE empleados;
```