



Tecnicatura Universitaria en Programación

## **BASES DE DATOS I**

Unidad Temática N°3: Programación en SQL Server

Teórico

1° Año – 2° Cuatrimestre







## Índice

Pro	ogramación en Bases de Datos	2
	Control de flujo	3
Pro	ocedimientos almacenados	6
	Ventajas:	7
	Creación y ejecución de Procedimientos Almacenados	8
	Modificar procedimientos almacenados	9
	Eliminar procedimientos almacenados	9
	Procedimientos almacenados con parámetros de entrada	9
	Procedimientos almacenados con parámetros de salida	11
	Procedimientos almacenados: return	12
	Procedimientos almacenados: Insert	14
	Sintaxis completa para la creación de procedimientos almacenados	15
Fu	nciones definidas por el usuario	20
	Funciones Escalares	20
	Funciones de tabla de varias instrucciones	22
	Funciones con valores de tabla en línea	24
	Modificar Funciones	25
	Eliminar Funciones	26
	Sintaxis completa de la creación de funciones definidas por el usuario	27
Triç	ggers	30
	Disparador de inserción	32
	Disparador de borrado	34
	Disparadores de actualización	35
	Disparador Instead Off y After	36
Ма	nejo de errores	38
BIE	BLIOGRAFÍA	42





## Programación en Bases de Datos

Todas las aplicaciones que se comunican con SQL Server lo hacen enviando instrucciones Transact-SQL al servidor, independientemente de la interfaz de usuario de la aplicación. Es el lenguaje de programación que proporciona SQL Server para ampliar el lenguaje SQL con los elementos característicos de los lenguajes de programación:

- variables.
- sentencias de control de flujo,
- bucles

Cuando se desea realizar una aplicación completa para el manejo de una base de datos relacional, resulta necesario utilizar alguna herramienta que soporte la capacidad de consulta del SQL y la versatilidad de los lenguajes de programación tradicionales.

#### Variables

Una variable local de es un objeto que contiene un valor individual de datos de un tipo específico. 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. Del mismo modo como acumulador
- Para contener un valor de datos que desea probar mediante una instrucción de control 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 de una función.

Tenga en cuenta que no se pueden usar variables en una vista.

La instrucción DECLARE inicializa una variable. El nombre debe ser único y tener un único @ como primer carácter. Se debe asignar siempre el tipo de dato de y su longitud si fuera necesaria.

Por ejemplo:

```
declare @codigo int;
```

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.





Para declarar más de una variable local, use una coma después de la primera variable local definida y, a continuación, especifique el nombre y tipo de datos de la siguiente variable local. Por ejemplo:

```
declare @codigo int, @nombre varchar(50), @fecha datetime;
```

Cuando una variable se declara por primera vez, su valor se establece en NULL. Para asignar un valor a una variable, use la instrucción SET. También se puede asignar un valor a una variable si se hace referencia a ella en la lista de selección de una instrucción SELECT. Ejemplo:

```
set @codigo = 1;
```

También se puede asignar un valor a una variable si se hace referencia a la misma en una lista de selección. Cuando se hace referencia a una variable en una lista de selección, se le debe asignar un valor escalar o la instrucción SELECT solo debe devolver una fila. Por ejemplo:

```
declare @ultimo_nro int;
select @ultimo_nro=max(nro_factura)
  from facturas;
```

Si una instrucción SELECT devuelve más de una fila y la variable hace referencia a una expresión no escalar, la variable se establece en el valor devuelto para la expresión en la última fila del conjunto de resultados.

#### **Operadores**

- Aritméticos: +, ,\* ,/ ,% (módulo).
- Lógicos: And, Or, Not
- Comparación: =, <>, >, >=, <, <=.</li>
- Concatenación: +

#### Comentarios

- Para comentario de línea: --.
- Para comentario de bloque: /\* \*/

#### Control de flujo

 BEGIN / END: se usan para agrupar varias instrucciones Transact-SQL en un bloque lógico. Use las instrucciones BEGIN y END en cualquier parte cuando





una instrucción de control de flujo deba ejecutar un bloque con dos o más instrucciones Transact-SQL.

 IF / ELSE: se usa para validar una condición y realizar una serie de acciones solo si la condición es verdadera o realizar unas acciones si la condición es verdadera y otras si es falsa.

Por ejemplo si se quisieran aumentar los precios de los artículos menores a \$30 y si no existen, dé un mensaje informando esta situación.

```
if exists (select * from articulos where pre_unitario<30)
begin
  update articulos
    set pre_unitario=pre_unitario*1.1
    where pre_unitario<30
    select 'Precios actualizados'
end
else
  select 'no hay artículos con esos precios'</pre>
```

 RETURN: termina incondicionalmente una serie de acciones y sale del procedimiento almacenado o lote. Ninguna de las instrucciones del lote que siga a la instrucción RETURN se ejecutará.

Como ejemplo se crea un procedimiento que muestre todos los articulos de una descripción determinada que se ingresa como parámetro:

```
create procedure pa_articulos_descrip
  @descripcion varchar(100) = null
as
  if @descripcion is null
begin
  select 'Debe indicar una descripción'
  return
end;
  select descripcion, pre_unitario, observaciones
  from articulos
  where descripcion like @descripcion
```

Si al ejecutar el procedimiento enviamos el valor "null" o no pasamos valor, con lo cual toma el valor por defecto "null", se muestra un mensaje y sale; en caso contrario, ejecuta la consulta luego del "else".

 WHILE / BREAK o CONTINUE: repite una instrucción o bloque de instrucciones mientras la condición especificada siga siendo verdadera.. La





instrucción BREAK sale del bucle WHILE más profundo, y la instrucción CONTINUE reinicia un bucle WHILE.

Por ejemplo si se quiere crear un SP llamado pa\_factorial que retorne el factorial de un número,

```
create procedure pa factorial
 @numero int
 as
  if @numero>=0 and @numero<=12</pre>
  begin
   declare @resultado int
   declare @num int
   set @resultado=1
   set @num=@numero
   while (@num>1)
   begin
     set @resultado=@resultado*@num
     set @num=@num-1
   end
   select rtrim(convert(char,@numero))+'!='+convert(char,@resultado)
  end
  else
   select 'Debe ingresar un número entre 0 y 12';
go
```

Ejecutar el procedimiento que nos devuelve el factorial de un número:

```
exec pa factorial 10;
```

 CASE: permite mostrar un valor alternativo dependiendo del valor de una columna o variable.

```
case VALORACOMPARAR
  when VALOR_1 then Rta_1
  when VALOR_2 then Rta_2
  ...
  else Rta_N
end
```

Por cada valor hay un "when" y un "then"; si encuentra un valor coincidente en algún "when" ejecuta el "then" correspondiente a ese "when", 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.

Por ejemplo: se quiere mostrar un mensaje según la cantidad vendida





```
when cantidad < 10 then 'Minorista'
when pre_unitario >= 10 and pre_unitario < 50 then '5% dto'
else '15% dto.'
end
from facturas f join detalle_facturas d
on f.nro_factura = d.nro_factura</pre>
```

La sentencia CASE se puede utilizar dentro de SELECT, UPDATE y DELETE; el resto de las sentencias de control de flujo no pueden utilizase dentro de las primeras tres (SELECT, UPDATE y DELETE)

A partir del SQL Server 2012 se tiene IIF que es una función incorporada del sistema que tiene tres parámetros; el 1°, es la condición que se va a evaluar, el 2° es el valor a devolver en caso de que la condición sea verdadera y el 3° es el valor que devuelve en caso de que la condición sea falsa.

Como ejemplo del uso de la función iif se podría querer listar los artículos con su precio y una leyenda que diga "Caro" si el precio unitario es mayor a \$100 y "Barato" en caso contrario.

```
SELECT descripcion, pre_unitario, iif(pre_unitario>100,'Caro','Barato')
FROM articulos
```

#### Procedimientos almacenados

Un procedimiento almacenado es una colección guardada de instrucciones Transact-SQL o una referencia a un método Common Language Runtime (CLR) de Microsoft .NET Framework que puede recopilar y devolver parámetros proporcionados por el usuario. Permiten encapsular tareas repetitivas

Los procedimientos se pueden crear para uso permanente o para uso temporal en una sesión, un procedimiento local temporal, o para su uso temporal en todas las sesiones, un procedimiento temporal global.

Los procedimientos almacenados también se pueden crear de modo que se ejecuten de forma automática al iniciarse una instancia de SQL Server.

SQL Server permite los siguientes tipos de procedimientos almacenados:

- Del sistema: están almacenados en la base de datos "master" y llevan el prefijo "sp\_".
- Locales: los crea el usuario





- Temporales: pueden ser locales, cuyos nombres comienzan con un signo numeral (#), o globales, cuyos nombres comienzan con 2 signos numeral (##). Los procedimientos almacenados temporales locales están disponibles en la sesión de un solo usuario y se eliminan automáticamente al finalizar la sesión; los globales están disponibles en las sesiones de todos los usuarios.
- Remotos son una característica anterior de SQL Server.
- Extendidos: se implementan como bibliotecas de vínculos dinámicos (DLL, Dynamic-Link Libraries), se ejecutan fuera del entorno de SQL Server.
   Generalmente llevan el prefijo "xp\_".

Un procedimiento almacenado puede hacer referencia a objetos que no existen al momento de crearlo. Los objetos deben existir cuando se ejecute el procedimiento almacenado.

Los procedimientos almacenados en SQL Server pueden:

- Contener instrucciones que realizan operaciones en la base de datos y llamar a otros procedimientos almacenados.
- Aceptar parámetros de entrada.
- Devolver un valor de estado a un procedimiento.
- Devolver varios parámetros de salida.

#### Ventajas:

- Comparten la lógica de la aplicación con las otras aplicaciones, con lo cual el acceso y las modificaciones de los datos se hacen en un solo sitio.
- Permiten realizar todas las operaciones que los usuarios necesitan evitando que tengan acceso directo a las tablas.
- Reducen el tráfico de red; en vez de enviar muchas instrucciones, los usuarios realizan operaciones enviando una única instrucción.
- Pueden encapsular la funcionalidad del negocio. Las reglas o directivas empresariales encapsuladas en los procedimientos almacenados se pueden cambiar en una sola ubicación.
- Apartar a los usuarios de la exposición de los detalles de las tablas de la base de datos.
- Proporcionar mecanismos de seguridad: los usuarios pueden obtener permiso para ejecutar un procedimiento almacenado incluso si no tienen permiso de acceso a las tablas o vistas a las que hace referencia.





Los procedimientos almacenados pueden hacer referencia a tablas, vistas, a funciones definidas por el usuario, a otros procedimientos almacenados y a tablas temporales.

Un procedimiento almacenado puede incluir cualquier cantidad y tipo de instrucciones, excepto: *create default, create procedure, create rule, create trigger y create view.* 

Si un procedimiento almacenado crea una tabla temporal, dicha tabla sólo existe dentro del procedimiento y desaparece al finalizar el mismo. Lo mismo sucede con las variables.

#### Creación y ejecución de Procedimientos Almacenados

Para crear un procedimiento almacenado se emplea la instrucción "create procedure". La sintaxis básica parcial es:

# CREATE PROCEDURE nombreprocedimiento AS instrucciones;

Con las siguientes instrucciones se crea un procedimiento almacenado llamado "pa\_articulos\_precios\_mas100" que muestra todos los artículos de la librería con precios mayores a 100:

```
CREATE PROC pa_ articulos_precios_mas100
AS
SELECT *
FROM articulos
WHERE pre unitario >100;
```

El procedimiento se ejecuta colocando "execute" (o "exec") seguido del nombre del procedimiento almacenado creado anteriormente como se muestra en la siguiente sentencia:

```
EXEC pa_articulos_precios_mas100;
```

Cuando un procedimiento se ejecuta por primera vez, se compila para determinar que dispone de un plan de acceso óptimo para recuperar los datos. En las siguientes ejecuciones del procedimiento almacenado se puede volver a utilizar el plan ya generado si aún permanece en la caché de planes del Database Engine (Motor de base de datos).





#### **Modificar procedimientos almacenados**

Los procedimientos almacenados pueden modificarse, por necesidad de los usuarios o por cambios en la estructura de las tablas que referencia. Sintaxis:

## ALTER PROCEDURE nombreprocedimiento @parametro TIPO = valorpredeterminado AS sentencias;

Modificar el procedimiento almacenado anterior para que muestre determinadas columnas:

```
ALTER PROC pa_articulos_precios_mas100
AS
SELECT descripcion, pre_unitario,observaciones,stock
FROM articulos
WHERE pre unitario >100;
```

#### Eliminar procedimientos almacenados

Los procedimientos almacenados se eliminan con "DROP PROCEDURE". Sintaxis:

#### **DROP PROCEDURE nombreprocedimiento;**

Eliminar el procedimiento almacenado llamado "pa\_articulosA":

```
DROP PROCEDURE pa articulosA;
```

#### Procedimientos almacenados con parámetros de entrada

Los parámetros de entrada posibilitan pasar información a un procedimiento. Para que un procedimiento almacenado admita parámetros de entrada se deben declarar variables como parámetros al crearlo. La sintaxis es:

```
CREATE PROC nombreprocedimiento
@nombreparametro tipo =valorpordefecto
AS sentencias;
```

Los parámetros se definen luego del nombre del procedimiento, comenzando con un signo arroba (@). Los parámetros son locales al procedimiento, es decir, existen solamente dentro del mismo. Pueden declararse varios parámetros por procedimiento, se separan por comas.





Cuando el procedimiento es ejecutado, deben explicitarse valores para cada uno de los parámetros (en el orden que fueron definidos), a menos que se haya definido un valor por defecto, en tal caso, pueden omitirse.

Luego de definir un parámetro y su tipo, se puede especificar un valor por defecto; tal valor es el que asume el procedimiento al ser ejecutado si no recibe parámetros. El valor por defecto puede ser "null" o una constante, también puede incluir comodines si el procedimiento emplea "like".

Como ejemplo se crea un procedimiento que recibe dos precios unitarios para mostrar todos aquellos que se encuentren entre dichos valores:

```
create procedure pa_articulos_precios
  @precio1 money,
  @precio2 money,
AS
  select descripcion,pre_unitario,observaciones
  from articulos
  where pre unitario between @precio1 and @precio2;
```

El procedimiento se ejecuta colocando "exec" seguido del nombre del procedimiento y los valores para los parámetros separados por comas:

```
exec pa_articulos_precios 100,200;
```

El ejemplo anterior ejecuta el procedimiento pasando valores a los parámetros por posición. También podemos emplear la otra sintaxis en la cual pasamos valores a los parámetros por su nombre:

```
exec pa_articulos_precios @precio1=100, @precio2=200;
```

En este caso, cuando los valores son pasados con el nombre del parámetro, el orden en que se colocan puede alterarse.

```
exec pa articulos precios @precio2=200, @precio1=100;
```

Si se quiere ejecutar un procedimiento que permita omitir los valores para los parámetros se debe, al crear el procedimiento, definir valores por defecto para cada parámetro:

```
create procedure pa_articulos_precios
  @precio1 money=100,
  @precio2 money=200,
AS
  select descripcion,pre unitario,observaciones
```





```
from articulos
where pre_unitario between @precio1 and @precio2;
```

```
exec pa articulos precios;
```

Si se envía un solo parámetro a un procedimiento que tiene definido más de un parámetro sin especificar a qué parámetro corresponde (valor por posición), asume que es el primero.

Se puede emplear patrones de búsqueda en la consulta que define el procedimiento almacenado y utilizar comodines como valores por defecto:

```
create procedure pa_articulos_descripcion
  @descripcion varchar(100)= '%'
AS
  select descripcion,pre_unitario,observaciones
  from articulos
  where descripción like @descripcion;
exec pa_articulos_descripcion;
```

#### Procedimientos almacenados con parámetros de salida

Dijimos que los procedimientos almacenados pueden devolver información; para ello se emplean parámetros de salida. El valor se retorna a quien realizó la llamada con parámetros de salida.

Para que un procedimiento almacenado devuelva un valor se debe declarar una variable con la palabra clave "output" al crear el procedimiento:

```
CREATE PROCEDURE nombreprocedimiento
@parametroentrada tipo =valorpordefecto,
@parametrosalida tipo=valorpordefecto output
AS
sentencias
select @parametrosalida=sentencias;
```

Los parámetros de salida pueden ser de cualquier tipo de datos, excepto text, ntext e image. Se crea un procedimiento almacenado que muestre los descripción, precio y observaciones de los articulos que comiencen con una (o más) letra/s determinada/s (enviada como parámetro de entrada) y retorne la suma y el promedio de los precios de todos los artículos que cumplan con la condición de búsqueda:





```
create procedure pa_articulos_sumaypromedio
    @descripcion varchar(100)='%',
    @suma decimal(10,2) output,
    @promedio decimal(8,2) output
    as
    select descripcion,pre_unitario,observaciones
    from articulos
    where descripcion like @descripcion
    select @suma=sum(pre_unitario)
    from articulos
    where descripcion like @descripcion
    select @promedio=avg(pre_unitario)
    from articulos
    where descripcion like @descripcion;
```

Al ejecutar el procedimiento se puede ver el contenido de las variables en las que se almacenan los parámetros de salida del procedimiento (Al ejecutarlo también debe emplearse "output"):

```
declare @s decimal(10,2), @p decimal(8,2)
  execute pa_autor_sumaypromedio 'lápiz%', @s output, @p output
  select @s as total, @p as promedio;
```

#### Procedimientos almacenados: return

La instrucción "return" sale de una consulta o procedimiento y todas las instrucciones posteriores no son ejecutadas.

Como ejemplo se crea un procedimiento que muestre todos los articulos de una descripción determinada que se ingresa como parámetro:

```
create procedure pa_articulos_descrip
  @descripcion varchar(100)= null
as
  if @descripcion is null
begin
    select 'Debe indicar una descripción'
    return
end;
select descripcion, pre_unitario, observaciones
    from articulos
    where descripción like @descripcion
```

Si al ejecutar el procedimiento enviamos el valor "null" o no pasamos valor, con lo cual toma el valor por defecto "null", se muestra un mensaje y sale; en caso contrario, ejecuta la consulta luego del "else"; "return" puede retornar un valor entero.





Se crea un procedimiento almacenado que ingresa registros en la tabla "articulos". Los parámetros correspondientes a la descripción y precio DEBEN ingresarse con un valor distinto de "null", los demás son opcionales. El procedimiento retorna "1" si la inserción se realiza, es decir, si se ingresan valores para descripción y precio y "0", en caso que descripción y precio sean nulos:

```
create procedure pa_articulos_alta
    @descrip varchar(100)=null,
    @precio money=null,
    @observ varchar(100)=null

as
    if (@descrip is null) or (@precio is null)
        return 0
    else
    begin
    insert into articulos(descripcion,pre_unitario,observaciones)
    values (@descrip,@precio,@observ)
    return 1
    end;
```

Para ver el resultado, debemos declarar una variable en la cual se almacene el valor devuelto por el procedimiento; luego, ejecutar el procedimiento asignándole el valor devuelto a la variable, finalmente mostramos el contenido de la variable:

```
declare @retorno int
exec @retorno=pa_articulos_alta 'Lapicera Big azul',45, 'trazo fino'
select 'Ingreso realizado=1' = @retorno
exec @retorno=pa_articulos_alta
select 'Ingreso realizado=1' = @retorno;
```

También podríamos emplear un "if" para controlar el valor de la variable de retorno:

```
declare @retorno int;
exec @retorno= pa_articulos_alta 'Cuaderno 24 hojas',60
if @retorno=1 print 'Registro ingresado'
else select 'Registro no ingresado porque faltan datos'
```

#### Procedimientos almacenados encriptados

Si no quiere que los usuarios puedan leer el contenido del procedimiento podemos indicarle a SQL Server que codifique la entrada a la tabla "syscomments" que contiene el texto. Para ello, debemos colocar la opción "with encryption" al crear el procedimiento:

# CREATE PROCEDURE nombreprocedimiento parámetros





# WITH ENCRYPTION AS instrucciones;

Si se ejecuta el procedimiento almacenado del sistema "sp\_helptext" para ver su contenido, no aparece.

#### Procedimientos almacenados: Insert

Podemos ingresar datos en una tabla con el resultado devuelto por un procedimiento almacenado. La instrucción siguiente crea el procedimiento "pa\_ofertas", que ingresa libros en la tabla "ofertas":

```
create proc pa_articulos_ofertas
  as
select descripcion,pre_unitario,observaciones
  from articulos
  where pre unitario<100;</pre>
```

La siguiente instrucción ingresa en la tabla "ofertas" el resultado del procedimiento "pa\_ofertas":

```
insert into ofertas exec pa_articulos_ofertas;
```

Las tablas deben existir y los tipos de datos deben coincidir.

#### Anidamiento de procedimientos almacenados

Los procedimientos almacenados pueden anidarse, es decir, un procedimiento almacenado puede llamar a otro. Las características del anidamiento de procedimientos almacenados son las siguientes:

- Los procedimientos almacenados se pueden anidar hasta 32 niveles.
- El nivel actual de anidamiento se almacena en la función del sistema
   @ @ nestlevel.
- Si un procedimiento almacenado llama a otro, éste puede obtener acceso a todos los objetos que cree el primero, incluidas las tablas temporales.
- Los procedimientos almacenados anidados pueden ser recursivos. Por ejemplo, el procedimiento almacenado A puede llamar al procedimiento almacenado B y éste puede llamar al procedimiento almacenado A.

Se creamos un procedimiento que retorne el factorial de un número:





```
create procedure pa_factorial
  @numero int
as

declare @resultado int
declare @num int
set @resultado=1
set @num=@numero
while (@num>1)
begin
  exec pa_multiplicar @resultado,@num, @resultado output
  set @num=@num-1
end
select rtrim(convert(char,@numero))+'!='+convert(char,@resultado);
```

Cuando un procedimiento (A) llama a otro (B), el segundo (B) tiene acceso a todos los objetos que cree el primero (A).

#### Sintaxis completa para la creación de procedimientos almacenados

#### **Argumentos**

**schema\_name** Es el nombre del esquema al que pertenece el procedimiento.





**procedure\_name** Es el nombre del nuevo procedimiento almacenado. Los nombres de los procedimientos deben cumplir las reglas de los identificadores y deben ser exclusivos en el esquema Se recomienda no utilizar el prefijo sp\_ en el nombre del procedimiento. SQL Server utiliza este prefijo para designar a los procedimientos almacenados del sistema.

Los procedimientos temporales locales o globales se pueden crear anteponiendo el signo de número (#) a procedure\_name (#procedure\_name) para los procedimientos temporales locales y dos signos de número para los procedimientos temporales globales (##procedure\_name). No es posible especificar nombres temporales para los procedimientos almacenados CLR.

El nombre completo de un procedimiento almacenado o un procedimiento almacenado temporal global, incluidas ##, no puede superar los 128 caracteres. El nombre completo de un procedimiento almacenado temporal local, incluidas #, no puede superar los 116 caracteres.

; number Es un entero opcional que se utiliza para agrupar procedimientos que tengan el mismo nombre.

Estos procedimientos agrupados se pueden quitar juntos mediante una instrucción DROP PROCEDURE. Por ejemplo, una aplicación denominada orders puede utilizar procedimientos denominados orderproc;1, orderproc;2, etc. La instrucción DROP PROCEDURE orderproc quita el grupo completo.

Si el nombre contiene identificadores delimitados, el número no debe incluirse como parte del identificador; sólo utilice el delimitador adecuado alrededor de procedure\_name. Los procedimientos almacenados numerados tienen las siguientes restricciones:

- No permiten el uso de tipos de datos xml o tipos definidos por el usuario CLR.
- No permiten crear una guía de plan en un procedimiento almacenado numerado.

**@parameter** Es un parámetro del procedimiento con parámetros de entrada y/o salida. En una instrucción CREATE PROCEDURE se pueden declarar uno o más parámetros. El usuario debe proporcionar el valor de cada parámetro declarado cuando se llama al procedimiento, a menos que se haya definido un valor predeterminado para el parámetro o se haya establecido en el mismo valor que otro parámetro.

Un procedimiento almacenado puede tener un máximo de 2.100 parámetros. Se debe especicar un nombre de parámetro con un signo (@) como primer carácter. El nombre del parámetro se debe ajustar a las reglas de los identificadores.





Los parámetros son locales para el procedimiento; los mismos nombres de parámetro se pueden utilizar en otros procedimientos.

De manera predeterminada, los parámetros sólo pueden ocupar el lugar de expresiones constantes; no se pueden utilizar en lugar de nombres de tabla, nombres de columna o nombres de otros objetos de base de datos.

No es posible declarar parámetros si se ha especificado FOR REPLICATION.

[type\_schema\_name.]data\_type Es el tipo de datos del parámetro y el esquema al que pertenece.

Se pueden utilizar todos los tipos de datos, excepto table, como un parámetro de un procedimiento almacenado Transact-SQL. Sin embargo, el tipo de datos cursor sólo se puede utilizar en parámetros OUTPUT. Cuando se especifica un tipo de datos cursor, deben especificarse también las palabras clave VARYING y OUTPUT. Con el tipo de datos cursor, es posible especificar varios parámetros de salida. En los procedimientos almacenados CLR, no es posible especificar como parámetros los tipos char, varchar, text, ntext, image, cursor y table.

Si el tipo de datos del parámetro es un tipo definido por el usuario CLR, es necesario disponer de permiso EXECUTE en el tipo. Si no se especifica el parámetro type\_schema\_name, el SQL Server Database Engine (Motor de base de datos de SQL Server) hace referencia a type\_name en el siguiente orden:

- Los tipos de datos del sistema de SQL Server.
- El esquema predeterminado del usuario actual en la base de datos actual.
- El esquema dbo en la base de datos actual.

En los procedimientos almacenados numerados, el tipo de datos no puede ser un tipo xml o definido por el usuario CLR.

**VARYING** Especifica el conjunto de resultados admitido como un parámetro de salida. Este parámetro es creado de forma dinámica por el procedimiento almacenado y su contenido puede variar. Sólo se aplica a los parámetros de tipo cursor.

**default** Es un valor predeterminado para el parámetro. Si se define un valor default, el procedimiento se puede ejecutar sin especificar un valor para ese parámetro. El valor predeterminado debe ser una constante o puede ser NULL.

Si el procedimiento utiliza el parámetro con la palabra clave LIKE, puede incluir los siguientes caracteres comodín: % \_ [] y [^].





**OUTPUT** Indica que se trata de un parámetro de salida. El valor de esta opción puede devolverse a la instrucción EXECUTE que llama. Utilice los parámetros OUTPUT para devolver valores al autor de la llamada del procedimiento. Los parámetros text, ntext e image no se pueden utilizar como parámetros OUTPUT, a menos que se trate de un procedimiento CLR. Un parámetro de salida que utilice la palabra clave OUTPUT puede ser un marcador de posición de cursor, a menos que el procedimiento sea un procedimiento CLR.

**RECOMPILE** Indica que el Database Engine (Motor de base de datos) no almacena en caché un plan para este procedimiento y que éste se compila en tiempo de ejecución. Esta opción no se puede utilizar cuando se especifica FOR REPLICATION. No es posible especificar RECOMPILE en los procedimientos almacenados CLR. Para indicar al Motor de base de datos que descarte planes para las consultas individuales de un procedimiento almacenado, utilice la sugerencia de consulta RECOMPILE.

Utilice la sugerencia de consulta RECOMPILE cuando se utilicen valores atípicos o temporales en sólo un subconjunto de consultas que pertenece al procedimiento almacenado.

**ENCRYPTION** Indica que SQL Server convertirá el texto original de la instrucción CREATE PROCEDURE en un formato ofuscado. La salida de la ofuscación no es directamente visible en ninguna de las vistas de catálogo de SQL Server.

Los usuarios que no disponen de acceso a las tablas del sistema o a los archivos de base de datos no pueden recuperar el texto ofuscado. Sin embargo, estará disponible para los usuarios con privilegios que puedan obtener acceso a las tablas del sistema a través del puerto DAC o directamente a los archivos de base de datos. Además, los usuarios que pueden adjuntar un depurador al proceso del servidor pueden recuperar el procedimiento descifrado de la memoria en tiempo de ejecución.

Los procedimientos creados mediante esta opción no se pueden publicar como parte de la réplica de SQL Server.

**EXECUTE** AS Especifica el contexto de seguridad en el que se ejecuta el procedimiento almacenado.

**FOR REPLICATION** Especifica que los procedimientos almacenados creados para la réplica no se pueden ejecutar en el suscriptor.





Se utiliza un procedimiento almacenado creado con la opción FOR REPLICATION como filtro de procedimiento almacenado y sólo se ejecuta durante la réplica. No es posible declarar parámetros si se ha especificado FOR REPLICATION. No es posible especificar FOR REPLICATION en los procedimientos almacenados CLR. La opción RECOMPILE se pasa por alto en los procedimientos creados con FOR REPLICATION. Un procedimiento FOR REPLICATION tendrá un tipo de objeto RF en sys.objects y sys.procedures.

<sql\_statement> Una o más instrucciones Transact-SQL que se van a
incluir en el procedimiento.

#### EXTERNAL NAME assembly name.class name.method name

Especifica el método de un ensamblado de .NET Framework para que un procedimiento almacenado CLR haga referencia a él. class\_name debe ser un identificador válido de SQL Server que debe existir como clase en el ensamblado. Si la clase tiene un nombre completo de espacio de nombres que utiliza un punto (.) para separar las partes del espacio de nombres, el nombre de la clase debe delimitarse mediante paréntesis ([]) o comillas (" "). El método especificado debe ser un método estático de la clase.

#### Otras características de los procedimientos almacenados

No hay un tamaño máximo predefinido para un procedimiento almacenado. Un procedimiento almacenado definido por el usuario sólo se puede crear en la base de datos actual.

Los procedimientos temporales constituyen una excepción a esta regla, puesto que siempre se crean en tempolo.

Si no se especifica el nombre de esquema, se utiliza el esquema predeterminado del usuario que está creando el procedimiento.

La instrucción CREATE PROCEDURE no se puede combinar con otras instrucciones Transact-SQL en un único lote.

De manera predeterminada, los parámetros admiten valores NULL. Si se pasa un valor de parámetro NULL y ese parámetro se utiliza en una instrucción CREATE TABLE o ALTER TABLE en la que la columna a la que se hace referencia no admite valores NULL, el Motor de base de datos genera un error. Para impedir que se pase un valor NULL a una columna que no admite valores NULL, agregue lógica de





programación al procedimiento o utilice un valor predeterminado para la columna mediante la palabra clave DEFAULT de CREATE TABLE o ALTER TABLE.

Se recomienda especificar de forma explícita NULL o NOT NULL para cada columna de una tabla temporal.

Las opciones ANSI\_DFLT\_ON y ANSI\_DFLT\_OFF controlan la forma en la que el Motor de base de datos asigna los atributos NULL o NOT NULL a las columnas si no se especifican dichos atributos en una instrucción CREATE TABLE o ALTER TABLE.

Si una conexión ejecuta un procedimiento almacenado con valores distintos para estas opciones a los que utilizó la conexión que creó el procedimiento, las columnas de la tabla creada para la segunda conexión pueden ser distintas al admitir valores NULL y exhibir distintos comportamientos.

Si se especifica NULL o NOT NULL explícitamente para cada columna, las tablas temporales se crean con la misma capacidad de admitir valores NULL para todas las conexiones que ejecutan el procedimiento almacenado.

## Funciones definidas por el usuario

Con Microsoft SQL Server, puede diseñar sus propias funciones para complementar y ampliar las funciones (integradas) suministradas por el sistema. SQL Server admite tres tipos de funciones definidas por el usuario:

- Funciones escalares: es similar a una función integrada.
- Funciones con valores de tabla de varias instrucciones: devuelve una tabla creada por una o varias instrucciones Transact-SQL y es similar a un procedimiento almacenado.
- Funciones con valores de tabla en línea: devuelve una tabla que es el resultado de una sola instrucción SELECT.

#### **Funciones Escalares**

Una función escalar retorna un único valor. Como todas las funciones, se crean con la instrucción "create function". La sintaxis básica es:

CREATE FUNCTION nombre (@parametro tipo=valorpordefecto) RETURNS tipo BEGIN





instrucciones RETURN valor END:

Luego del nombre se colocan (opcionalmente) los parámetros de entrada con su tipo. La cláusula "returns" indica el tipo de dato retornado. El cuerpo de la función, se define en un bloque "begin…end" que contiene las instrucciones que retornan el valor. Creamos una función denominada "f\_promedio" que recibe 2 valores y retorna el promedio:

```
create function f_promedio
  (@valor1 decimal(4,2),
    @valor2 decimal(4,2)
)
  returns decimal (6,2)
  as
  begin
    declare @resultado decimal(6,2)
    set @resultado=(@valor1+@valor2)/2
    return @resultado
end;
```

En el ejemplo anterior se declara una variable local a la función (desaparece al salir de la función) que calcula el resultado que se retornará. Al hacer referencia a una función escalar, se debe especificar el propietario y el nombre de la función:

```
select dbo.f_promedio(5.5,8.5);
```

Cuando se llama a funciones que tienen definidos parámetros de entrada SE DEBEN suministrar SIEMPRE un valor para él.

Crear una función a la cual se le envía una fecha y retorna el nombre del mes en español:

```
create function f_nombreMes
  (@fecha datetime='2007/01/01')
  returns varchar(10)
  as
  begin
    declare @nombre varchar(10)
    set @nombre=
    case datename(month,@fecha)
      when 'January' then 'Enero'
      when 'February' then 'Febrero'
      when 'March' then 'Marzo'
      when 'April' then 'Abril'
      when 'May' then 'Mayo'
```





```
when 'June' then 'Junio'
when 'July' then 'Julio'
when 'August' then 'Agosto'
when 'September' then 'Setiembre'
when 'October' then 'Octubre'
when 'November' then 'Noviembre'
when 'December' then 'Diciembre'
end--case
return @nombre
end;
```

Las funciones que retornan un valor escalar pueden emplearse en cualquier consulta donde se coloca un campo.

```
select nombre,
  dbo.f_nombreMes(fechaingreso) as 'mes de ingreso'
  from empleados;
```

Se puede colocar un valor por defecto al parámetro, pero al invocar la función, para que tome el valor por defecto SE DEBE especificar "default".

```
select dbo.f_nombreMes(default);
```

La instrucción "create function" debe ser la primera sentencia de un lote.

Las funciones definidas por el usuario no se pueden utilizar para realizar acciones que modifican el estado de la base de datos.

#### Funciones de tabla de varias instrucciones

Las funciones que retornan una tabla pueden emplearse en lugar de un "from" de una consulta. Sintaxis:

```
CREATE FUNCTION nombrefuncion
(@parametro tipo)
RETURNS @nombretablaretorno TABLE-- nombre de la tabla
--formato de la tabla
(campo1 tipo,
 campo2 tipo,
 campo3 tipo
)
AS
BEGIN
```





```
INSERT @nombretablaretorno
SELECT campos
FROM tabla
WHERE campo operador @parametro
RETURN
END
```

La cláusula "returns" define un nombre de variable local para la tabla que retornará, el tipo de datos a retornar (que es "table") y el formato de la misma (campos y tipos).

El siguiente ejemplo crea una función denominada "f\_ofertas" que recibe un parámetro. La función retorna una tabla con el codigo, descripcion pre\_unitario de todos los articulos cuyo precio sea inferior al parámetro:

```
create function f ofertas
(@minimo decimal(8,2))
returns @ofertas table-- nombre de la tabla
--formato de la tabla
(cod articulo int,
descripcion varchar(100),
pre unitario money,
observaciones varchar(100)
)
as
begin
  insert @ofertas
   select cod articulo, descripcion, pre unitario, observaciones
   from articulos
  where pre unitario < @minimo
  return
end;
```

La siguiente consulta realiza un join entre la tabla "articulos" y la tabla retornada por la función "f\_ofertas":

```
from articulos as a
  join dbo.f_ofertas(25) as o
  on a.cod_articulo=o.articulo;
```

Se puede llamar a la función como si fuese una tabla o vista listando algunos campos:

```
select descripcion,pre unitario from dbo.f ofertas(40);
```





#### Funciones con valores de tabla en línea

Una función con valores de tabla en línea retorna una tabla que es el resultado de una única instrucción "select". Es similar a una vista, pero con la posibilidad de utilizar parámetros. Sintaxis:

```
CREATE FUNCTION nombrefuncion
(@parametro tipo=valorpordefecto)
RETURNS TABLE
AS
RETURN (
SELECT campos
FROM tabla
WHERE condición
);
```

El valor por defecto es opcional.

"returns" especifica "table" como el tipo de datos a retornar. No se define el formato de la tabla a retornar porque queda establecido en el "select".

El cuerpo de la función no contiene un bloque "begin...end" como las otras funciones.

La cláusula "return" contiene una sola instrucción "select" entre paréntesis. El resultado del "select" es la tabla que se retorna. El "select" está sujeto a las mismas reglas que los "select" de las vistas.

Se crea una función con valores de tabla en línea que recibe un valor de autor como parámetro:

```
create function f_articulos
(@descrip varchar(100)='Lápiz')
returns table
as
return (
  select descripcion,pre_unitario
  from articulos
  where descripcion like '%'+@descrip+'%'
);
```

Estas funciones retornan una tabla y se hace referencia a ellas en la cláusula "from", como una vista:

```
select * from f_articulos('Lápiz');
```





#### Funciones encriptadas

Las funciones definidas por el usuario pueden encriptarse, para evitar que sean leídas con "sp\_helptext". Para ello debemos agregar al crearlas la opción "with encryption" antes de "as". En funciones escalares. En funciones de tabla de varias sentencias se coloca luego del formato de la tabla a retornar.

#### **Modificar Funciones**

Las funciones definidas por el usuario pueden modificarse con la instrucción "alter function".

Sintaxis para modificar funciones escalares:

```
ALTER FUNTION propietario.nombrefuncion
(@parametro tipo=valorpordefecto)
RETURNS tipo
AS
BEGIN
cuerpo
RETURN expresionescalar
END
```

Sintaxis para modificar una función de varias instrucciones que retorna una tabla:

```
ALTER FUNCTION NOMBREFUNCION
(@parametro tipo=valorpordefecto)
RETURNS @variable TABLE
(definicion de la tabla a retornar)
AS
BEGIN
cuerpo de la funcion
RETURN
END
```

Sintaxis para modificar una función con valores de tabla en línea





```
ALTER FUNCTION nombrefuncion
(@parametro tipo)
RETURNS TABLE
AS
RETURN (sentencias select)
```

En un ejemplo: se crea una función que retorna una tabla en línea:

```
create function f articulos
(@descrip varchar(100)='Lápiz')
returns table
as
return (
 select descripcion, pre unitario
 from articulos
where descripcion like '%'+@descrip+'%'
);
    Para modificar la función agregando otro campo en el "select":
alter function f articulos
(@descrip varchar(100)='Lápiz')
returns table
as
return (
 select descripcion, pre unitario, observaciones
 from articulos
 where descripcion like '%'+@descrip+'%'
);
```

#### **Eliminar Funciones**

Las funciones definidas por el usuario se eliminan con la instrucción "drop function": Sintaxis:

#### **DROP FUNCTION nombreppropietario.nombrefuncion**;

Se coloca el nombre del propietario seguido del nombre de la función.

Si la función que se intenta eliminar no existe, aparece un mensaje indicándolo, para evitarlo, podemos verificar su existencia antes de solicitar su eliminación

Se elimina, si existe, la función denominada "f\_fechacadena":





```
if object_id('dbo.f_fechacadena') is not null
  drop function dbo.f_fechacadena;
```

#### Sintaxis completa de la creación de funciones definidas por el usuario

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

#### Argumentos

**schema\_name** El nombre del esquema al que pertenece la función definida por el usuario.

**function\_name** Nombre de la función definida por el usuario. Los nombres de funciones deben seguir las reglas de los identificadores y deben ser únicos en la base de datos y para su esquema.

@parameter\_name Es un parámetro de la función definida por el usuario. Es posible declarar uno o varios parámetros. Una función puede tener un máximo de 1.024 parámetros. El usuario debe proporcionar el valor de cada parámetro declarado cuando se ejecuta la función, a menos que se defina un valor predeterminado para el parámetro. Especifique un nombre de parámetro con una arroba (@) como primer carácter. El nombre del parámetro debe cumplir las mismas reglas que los identificadores. Los parámetros son locales para la función; los mismos nombres de parámetro se pueden utilizar en otras funciones. Los parámetros sólo pueden ocupar el lugar de constantes; no se pueden utilizar en lugar de nombres de tablas, nombres de columnas o nombres de otros objetos de base de datos.

[type\_schema\_name.] parameter\_data\_type Es el tipo de datos del parámetro y, de forma opcional, el esquema al que pertenece.





Para las funciones Transact-SQL, se permiten todos los tipos de datos, incluidos los tipos definidos por el usuario CLR, a excepción del tipo de datos timestamp. Para las funciones CLR, se permiten todos los tipos de datos, incluidos los tipos definidos por el usuario CLR, a excepción de los tipos de datos text, ntext, image y timestamp.

Los tipos de datos no escalares cursor y table no se pueden especificar como tipos de datos de parámetro en funciones Transact-SQL o CLR. Si no se especifica type\_schema\_name, el SQL Server Database Engine (Motor de base de datos de SQL Server) busca scalar\_parameter\_data\_type en el siguiente orden:

- El esquema que contiene los nombres de los tipos de datos del sistema de SQL Server.
- El esquema predeterminado del usuario actual en la base de datos actual.
- El esquema dbo en la base de datos actual.

[=default] Es un valor predeterminado para el parámetro. Si se define un valor default, la función se puede ejecutar sin especificar un valor para ese parámetro. Cuando un parámetro de la función tiene un valor predeterminado, se debe especificar la palabra clave DEFAULT al llamar a la función para recuperar el valor predeterminado. Este comportamiento es distinto del uso de parámetros con valores predeterminados en los procedimientos almacenados, donde la omisión del parámetro implica especificar el valor predeterminado.

return\_data\_type Es el valor devuelto de una función escalar definida por el usuario. Para las funciones Transact-SQL, se permiten todos los tipos de datos, incluidos los tipos definidos por el usuario CLR, a excepción del tipo de datos timestamp. Para las funciones CLR, se permiten todos los tipos de datos, incluidos los tipos definidos por el usuario CLR, a excepción de los tipos de datos text, ntext, image y timestamp. Los tipos de datos no escalares cursor y table no se pueden especificar como tipos de datos de retorno en funciones Transact-SQL o CLR.

**function\_body** Especifica que una serie de instrucciones Transact-SQL, que juntas no producen ningún efecto secundario (como, por ejemplo, modificar una tabla), definen el valor de la función. function\_body sólo se utiliza en funciones escalares y funciones con valores de tabla de múltiples instrucciones. En las funciones escalares, function\_body es una serie de instrucciones Transact-SQL que juntas se evalúan como un valor escalar. En las funciones con valores de tabla de múltiples instrucciones, function\_body es una serie de instrucciones Transact-SQL que rellenan una variable de retorno de TABLE.





scalar\_expression Especifica el valor escalar que devuelve la función escalar.

**TABLE** Especifica que el valor devuelto de la función con valores de tabla es una tabla. Sólo se pueden pasar constantes y @local\_variables a las funciones con valores de tabla. En las funciones con valores de tabla en línea, el valor devuelto de TABLE se define mediante una única instrucción SELECT.

Las funciones en línea no tienen variables de retorno asociadas. En las funciones con valores de tabla de múltiples instrucciones, @return\_variable es una variable de TABLE, que se utiliza para almacenar y acumular las filas que se deben devolver como valor de la función. @return\_variable sólo se puede especificar para funciones Transact-SQL, no para funciones CLR

select\_stmt Es la instrucción SELECT individual que define el valor devuelto de una función con valores de tabla en línea.

EXTERNAL NAME , assembly\_name.class\_name.method\_name Especifica el método de ensamblado para enlazar con la función. assembly\_name debe coincidir con un ensamblado existente en SQL Server, en la base de datos actual con la visibilidad activada. class\_name debe ser un identificador SQL Server válido y debe existir como clase en el ensamblado.

Si la clase tiene un nombre calificado como espacio de nombres con un punto (.) para separar las partes del espacio de nombres, se debe delimitar el nombre de la clase con corchetes ([]) o comillas (" "). method\_name debe ser un identificador SQL Server válido y debe existir como método estático en la clase especificada.

Define el tipo de datos de tabla para una función Transact-SQL. La declaración de tabla incluye definiciones de columna y restricciones de columna o de tabla. La tabla se coloca siempre en el grupo de archivos principal.

Existen más parámetros que no hacen al contenido del curso, pero es recomendado que se consulte la documentación de Microsoft para estar al tanto de todo lo que se puede realizar por medio de funciones y sus respectivos parámetros.





Notas Las funciones definidas por el usuario son con valores escalares o con valores de tabla. Son funciones con valores escalares si la cláusula RETURNS especificó uno de los tipos de datos escalares. Las funciones con valores escalares se pueden definir utilizando varias instrucciones Transact-SQL.

Son funciones con valores de tabla si la cláusula RETURNS especificó TABLE. Según cómo se haya definido el cuerpo de la función, las funciones con valores de tabla se pueden clasificar en funciones en línea o de múltiples instrucciones.

Las siguientes instrucciones son válidas en una función:

- Instrucciones de asignación.
- Instrucciones de control de flujo, excepto las instrucciones TRY...CATCH.
- Instrucciones DECLARE que definen variables de datos locales y cursores locales.
- Instrucciones SELECT que contienen listas de selección con expresiones que asignan valores a variables locales.
- Operaciones de cursor que hacen referencia a cursores locales que se declaran, abren, cierran y cuya asignación se cancela en la función. Sólo se permiten las instrucciones FETCH que asignan valores a las variables locales mediante la cláusula INTO; no se permiten las instrucciones FETCH que devuelven los datos al cliente.
- Instrucciones INSERT, UPDATE y DELETE que modifican variables table locales.
- Instrucciones EXECUTE que llaman a procedimientos almacenados extendidos.

Las funciones definidas por el usuario se modifican con ALTER FUNCTION y se quitan con DROP FUNCTION.

## **Triggers**

Un "trigger" (disparador o desencadenador) es un tipo de procedimiento almacenado que se ejecuta cuando se intenta modificar los datos de una tabla (o vista). Se definen para una tabla (o vista) específica.

Se crean para conservar la integridad referencial y la coherencia entre los datos entre distintas tablas.





Si se intenta modificar (agregar, actualizar o eliminar) datos de una tabla en la que se definió un disparador para alguna de estas acciones (inserción, actualización y eliminación), el disparador se ejecuta (se dispara) en forma automática.

Un trigger se asocia a un evento (inserción, actualización o borrado) sobre una tabla.

La diferencia con los procedimientos almacenados del sistema es que los triggers:

- no pueden ser invocados directamente; al intentar modificar los datos de una tabla para la que se ha definido un disparador, el disparador se ejecuta automáticamente.
- no reciben y retornan parámetros.
- son apropiados para mantener la integridad de los datos, no para obtener resultados de consultas.

Los disparadores, a diferencia de las restricciones "check", pueden hacer referencia a campos de otras tablas. Por ejemplo, puede crearse un trigger de inserción en la tabla "ventas" que compruebe el campo "stock" de un artículo en la tabla "articulos"; el disparador controlaría que, cuando el valor de "stock" sea menor a la cantidad que se intenta vender, la inserción del nuevo registro en "ventas" no se realice.

Los disparadores se ejecutan DESPUES de la ejecución de una instrucción "insert", "update" o "delete" en la tabla en la que fueron definidos. Las restricciones se comprueban ANTES de la ejecución de una instrucción "insert", "update" o "delete". Por lo tanto, las restricciones se comprueban primero, si se infringe alguna restricción, el desencadenador no llega a ejecutarse.

Se crea solamente en la base de datos actual pero puede hacer referencia a objetos de otra base de datos

Las siguientes instrucciones no están permitidas en un desencadenador: create database, alter database, drop database, load database, restore database, load log, reconfigure, restore log, disk init, disk resize.

Se pueden crear varios triggers para cada evento, es decir, para cada tipo de modificación (inserción, actualización o borrado) para una misma tabla. Por ejemplo, se puede crear un "insert trigger" para una tabla que ya tiene otro "insert trigger".

Los triggers se crean con la instrucción "create trigger". Esta instrucción especifica la tabla en la que se define el disparador, los eventos para los que se





ejecuta y las instrucciones que contiene Debe ser la primera sentencia de un bloque y sólo se puede aplicar a una tabla

Sintaxis básica:

```
create trigger NOMBRE_DISPARADOR
  on NOMBRETABLA
  for EVENTO -- insert, update o delete
as
  SENTENCIAS
```

#### **Argumentos:**

create trigger junto al nombre del disparador.

**on** seguido del nombre de la tabla o vista para la cual se establece el trigger.

**for**, se indica la acción (evento, el tipo de modificación) sobre la tabla o vista que activará el trigger. Puede ser "insert", "update" o "delete". Debe colocarse al menos UNA acción, si se coloca más de una, deben separarse con comas.

**As** viene el cuerpo del trigger, se especifican las condiciones y acciones del disparador; es decir, las condiciones que determinan cuando un intento de inserción, actualización o borrado provoca las acciones que el trigger realizará.

#### Disparador de inserción

Se puede crear un disparador para que se ejecute siempre que una instrucción "insert" ingrese datos en una tabla.

Sintaxis básica:

```
create trigger NOMBRE_DISPARADOR
on NOMBRETABLA
for insert
as
SENTENCIAS
```

#### Argumentos:

create trigger junto al nombre del disparador;

on seguido del nombre de la tabla para la cual se establece el trigger.





**for** se coloca el evento (en este caso "insert"), lo que indica que las inserciones sobre la tabla activarán el trigger.

**as** se especifican las condiciones y acciones, es decir, las condiciones que determinan cuando un intento de inserción provoca las acciones que el trigger realizará.

Por ejemplo, si se quiere controlar que en una venta la cantidad vendida sea menor o igual al stock disponible y a la vez actualizar el stock, se puede crear un trigger sobre la tabla detalles\_facturas para el evento de inserción:

```
create trigger dis ventas insertar
 on detalle facturas
 for insert
  declare @stock int
  select @stock= stock from articulos
join inserted
on inserted.cod articulo=articulos.cod articulo
 if (@stock>=(select cantidad from inserted))
   update articulos set stock=stock-inserted.cantidad
    from articulos
    join inserted
    on inserted.cod articulo=articulos.cod articulo
 else
 begin
   raiserror ('El stock en articulos es menor que la cantidad
              solicitada', 16, 1)
   rollback transaction
 end
```

Cuando se activa un disparador "insert", los registros se agregan a la tabla del disparador y a una tabla denominada "inserted". La tabla "inserted" es una tabla virtual que contiene una copia de los registros insertados; tiene una estructura similar a la tabla en que se define el disparador, es decir, la tabla en que se intenta la acción. La tabla "inserted" guarda los valores nuevos de los registros.

Dentro del trigger se puede acceder a esta tabla virtual "inserted" que contiene todos los registros insertados, es lo que hicimos en el disparador creado anteriormente, lo que solicitamos es que se le reste al "stock" de "articulos", la





cantidad ingresada en el nuevo registro de "detalles\_facturas", valor que recuperamos de la tabla "inserted".

"rollback transaction" es la sentencia que deshace la transacción, es decir, borra todas las modificaciones que se produjeron en la última transacción restableciendo todo al estado anterior.

"raiserror" muestra un mensaje de error personalizado.

Para identificar fácilmente los disparadores de otros objetos se recomienda usar un prefijo y darles el nombre de la tabla para la cual se crean junto al tipo de acción.

#### Disparador de borrado

Se puede crear un disparador para que se ejecute siempre que una instrucción "delete" elimine datos en una tabla.

Sintaxis básica:

```
create trigger NOMBREDISPARADOR
on NOMBRETABLA
for delete
as
SENTENCIAS
```

#### Argumentos:

**create trigger** junto al nombre del disparador; "on" seguido del nombre de la tabla para la cual se establece el trigger.

**for** se coloca el evento (en este caso "delete"), lo que indica que las eliminaciones sobre la tabla activarán el trigger.

**as** se especifican las condiciones que determinan cuando un intento de eliminación causa las acciones que el trigger realizará.

En el siguiente ejemplo, el disparador se crea para que cada vez que se elimine un registro de detalles\_facturas, se actualice el campo "stock" de la tabla artículos:

```
create trigger dis_ventas_borrar
  on detalle_facturas
  for delete
as
  update articulos set stock = a.stock + deleted.cantidad
```





```
from articulos a
join deleted on deleted.cod_articulo = a.cod_articulo;
```

Cuando se activa un disparador "delete", los registros eliminados en la tabla del disparador se agregan a una tabla llamada "deleted". La tabla "deleted" es una tabla virtual que conserva una copia de los registros eliminados; tiene una estructura similar a la tabla en que se define el disparador, es decir, la tabla en que se intenta la acción.

Dentro del trigger se puede acceder a esta tabla virtual "deleted".

La sentencia "truncate table" no puede incluirse en un disparador de borrado (delete trigger).

#### Disparadores de actualización

Cuando se ejecute una instrucción update se podría crear un disparador para controlar las actualizaciones de los datos de una tabla.

Sintaxis básica:

```
create trigger NOMBREDISPARADOR
  on NOMBRETABLA
  for update
  as
    SENTENCIAS
```

#### Argumentos:

create trigger junto al nombre del disparador; "on" seguido del nombre de la tabla para la cual se establece el trigger.

**for** se coloca el evento (en este caso "update"), lo que indica que las actualizaciones sobre la tabla activarán el trigger.

**as** se especifican las condiciones y acciones, es decir, las condiciones que determinan cuando un intento de modificación provoca las acciones que el trigger realizará.

Por ejemplo, se quiere evitar que se modifiquen los datos de la tabla artículos y para ello se crea el siguiente disparador:

```
create trigger dis_articulos_actualizar
on articulos
for update
as
```





Cuando se ejecuta una instrucción "update" en una tabla que tiene definido un disparador, los registros originales (antes de ser actualizados) se mueven a la tabla virtual "deleted" y los registros actualizados (con los nuevos valores) se copian a la tabla virtual "inserted". Dentro del trigger se puede acceder a estas tablas.

En el cuerpo de un trigger se puede emplear la función "update(campo)" que recibe un campo y retorna verdadero si el evento involucra actualizaciones (o inserciones) en ese campo; en caso contrario retorna "false".

Si se quisiera evitar que se actualice el campo "pre\_unitario" de la tabla articulos:

#### **Disparador Instead Off y After**

Hasta el momento se ha mostrado triggers que se crean sobre una tabla específica para un evento de inserción, eliminación o actualización; pero, también se puede especificar el momento en que se ejecutan. Este momento de disparo indica que las acciones (sentencias) del trigger se ejecuten después (after) de la acción (insert, delete o update) que dispara el trigger o en lugar (instead of) de la acción.

La sintaxis básica es:

```
create trigger NOMBREDISPARADOR
on NOMBRETABLA o VISTA
MOMENTODEDISPARO-- after o instead of
ACCION-- insert, update o delete
as
SENTENCIAS
```





Si no se especifica el momento de disparo en la creación del trigger, por defecto se establece como "after", es decir, las acciones que el disparador realiza se ejecutan luego del suceso disparador.

Los disparadores "instead of" se ejecutan en lugar de la acción desencadenante, es decir, cancelan la acción desencadenante (suceso que disparó el trigger) reemplazándola por otras acciones.

Por ejemplo. Se crea un trigger que inserte un registro en la tabla detalles\_facturas si la cantidad vendida es menor al stock del artículo sino da un mensaje informando la situación:

```
create trigger dis clie actualizar
 on clientes
 instead of update
  if update(cod cliente)
  begin
     raiserror('Los códigos no pueden modificarse', 10, 1)
     rollback transaction
   end
   else
  begin
     update clientes
        set clientes.nom cliente=inserted.nom cliente,
            clientes.ape cliente=inserted.ape cliente
       from clientes join inserted
         on clientes.cod cliente =inserted.cod cliente
   end
```

#### Consideraciones:

- Se pueden crear disparadores "instead of" en vistas y tablas.
- No se puede crear un disparador "instead of" en vistas definidas "with check option".
- No se puede crear un disparador "instead of delete" y "instead of update" sobre tablas que tengan una "foreign key" que especifique una acción "on delete cascade" y "on update cascade" respectivamente.
- Los disparadores "after" no pueden definirse sobre vistas.





- No pueden crearse disparadores "after" en vistas ni en tablas temporales; pero pueden referenciar vistas y tablas temporales.
- Si existen restricciones en la tabla del disparador, se comprueban DESPUES de la ejecución del disparador "instead of" y ANTES del disparador "after". Si se infringen las restricciones, se revierten las acciones del disparador "instead of"; en el caso del disparador "after", no se ejecuta.

## Manejo de errores

Hay dos formas de obtener información de errores en Transact-SQL:

- En el ámbito del bloque CATCH de una construcción TRY...CATCH se pueden utilizar las siguientes funciones del sistema:
  - o **ERROR** LINE () devuelve el número de línea en que se produjo el error.
  - ERROR\_MESSAGE () devuelve el texto del mensaje que se devolvería a la aplicación. Este texto incluye los valores suministrados para los parámetros reemplazables, como longitudes, nombres de objetos u horas.
  - ERROR\_NUMBER() devuelve el número de error.
  - ERROR\_PROCEDURE() devuelve el nombre del procedimiento almacenado o desencadenador en que se produjo el error. Esta función devuelve NULL si el error no se ha producido en un procedimiento almacenado o un desencadenador.
  - ERROR\_SEVERITY() devuelve la gravedad.
  - ERROR\_STATE() devuelve el estado.
- Inmediatamente después de ejecutar una instrucción de Transact-SQL, puede hacer una prueba para detectar errores y recuperar el número de error mediante la función @@ERROR.

Las funciones ERROR\_LINE, ERROR\_MESSAGE, ERROR\_NUMBER, ERROR\_PROCEDURE, ERROR\_SEVERITY y ERROR\_STATE sólo devuelven información de errores cuando se utilizan en el ámbito del bloque CATCH de una construcción TRY...CATCH. Fuera del ámbito de un bloque CATCH devuelven NULL.

Estas funciones devuelven información acerca del error que provocó la invocación del bloque CATCH. Las funciones devuelven la misma información de





error en cualquier parte en que se ejecuten dentro del ámbito de un bloque CATCH, aunque se haga referencia a ellas varias veces.

Además, estas funciones proporcionan a las instrucciones de Transact-SQL los mismos datos que se devuelven a la aplicación. En bloques CATCH anidados, las funciones ERROR\_LINE, ERROR\_MESSAGE, ERROR\_NUMBER, ERROR\_PROCEDURE, ERROR\_SEVERITY y ERROR\_STATE devuelven la información de error específica para el bloque CATCH en el que se haga referencia a ellas. Por ejemplo, el bloque CATCH de una construcción TRY...CATCH externa podría tener una construcción TRY...CATCH anidada. Dentro del bloque CATCH anidado, estas funciones devuelven información acerca del error que invocó el bloque CATCH interno. Las mismas funciones en el bloque CATCH externo devolverían información acerca del error que invocó dicho bloque CATCH.

En el ejemplo siguiente se ilustra esto mostrando que cuando se hace referencia a ERROR\_MESSAGE en el bloque CATCH externo, la función devuelve el texto del mensaje generado en el bloque TRY externo. Cuando se hace referencia en el bloque CATCH interno, ERROR\_MESSAGE devuelve el texto de mensaje generado en el bloque TRY interno.

Este ejemplo también muestra que en el bloque CATCH externo, ERROR\_MESSAGE siempre devuelve el mensaje generado en el bloque TRY externo, incluso después de ejecutar la construcción TRY...CATCH interna.

```
IF EXISTS (SELECT message_id FROM sys.messages
WHERE message_id = 50010)
EXECUTE sp_dropmessage 50010;
GO
EXECUTE sp_addmessage @msgnum = 50010,
  @severity = 16,
  @msgtext = N'Message text is from the %s TRY block.';
GO
BEGIN TRY -- Outer TRY block.
  --Raise an error in the outer TRY block.
RAISERROR (50010, -- Message id.
  16, -- Severity,
  1, -- State,
  N'outer'); -- Indicate TRY block.
END TRY -- Outer TRY block.
BEGIN CATCH -- Outer CATCH block.
  -- Print the error message recieved for this
```





```
-- CATCH block.
 PRINT N'OUTER CATCH1: ' + ERROR MESSAGE();
 BEGIN TRY -- Inner TRY block.
   -- Start a nested TRY...CATCH and generate
   -- a new error.
   RAISERROR (50010, -- Message id.
      16, -- Severity,
     2, -- State,
     N'inner'); -- Indicate TRY block.
 END TRY -- Inner TRY block.
 BEGIN CATCH -- Inner CATCH block.
     -- Print the error message recieved for this
     -- CATCH block.
   PRINT N'INNER CATCH: ' + ERROR MESSAGE();
 END CATCH; -- Inner CATCH block.
 -- Show that ERROR MESSAGE in the outer CATCH
 -- block still returns the message from the
 -- error generated in the outer TRY block.
 PRINT N'OUTER CATCH2: ' + ERROR MESSAGE();
END CATCH; -- Outer CATCH block.
GΟ
```

La función @@ERROR se puede utilizar para capturar el número de un error generado con la instrucción anterior de Transact-SQL. @@ERROR sólo devuelve información de error inmediatamente después de la instrucción de Transact-SQL que genera el error.

- Si la instrucción que genera el error se encuentra en un bloque TRY, el valor de @@ERROR debe probarse y recuperarse en la primera instrucción del bloque CATCH asociado.
- Si la instrucción que genera el error no se encuentra en un bloque TRY, el valor de @@ERROR debe probarse y recuperarse en la instrucción inmediatamente después de aquélla que genera el error.

Fuera del ámbito de un bloque CATCH, el número de error de @ @ ERROR es la única información disponible acerca de un error dentro del código de Transact-SQL. Si el error utilizó un mensaje de error definido en sys.messages, se puede recuperar la gravedad definida y el texto del mensaje de error de sys.messages, tal como se muestra en este ejemplo.





```
IF EXISTS (SELECT message id FROM sys.messages
            WHERE message id = 50010)
  EXECUTE sp dropmessage 50010;
GO
-- Define a message with text that accepts
-- a substitution string.
EXECUTE sp addmessage @msgnum = 50010,
        @severity = 16,
        @msgtext = N'Substitution string = %s.';
GO
DECLARE @ErrorVariable INT;
-- RAISERROR uses a different severity and
-- supplies a substitution argument.
RAISERROR (50010, -- Message id.
   15, -- Severity,
   1, -- State,
   N'ABC'); -- Substitution Value.
-- Save @@ERROR.
SET @ErrorVariable = @@ERROR;
-- The results of this select illustrate that
-- outside a CATCH block only the original
-- information from sys.messages is available to
-- Transact-SQL statements. The actual message
-- string returned to the application is not
-- available to Transact-SQL statements outside
-- of a CATCH block.
SELECT @ErrorVariable AS ErrorID, text
  FROM sys.messages
 WHERE message id = @ErrorVariable;
GO
```





## **BIBLIOGRAFÍA**

Gorman K., Hirt A., Noderer D., Rowland-Jones J., Sirpal A., Ryan D. & Woody B (2019) Introducing Microsoft SQL Server 2019. Reliability, scalability, and security both on premises and in the cloud. Packt Publishing Ltd. Birmingham UK

Microsoft (2021) SQL Server technical documentation. Disponible en: https://docs.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver15

Opel, A. & Sheldon, R. (2010). Fundamentos de SQL. Madrid. Editorial Mc Graw Hill

Varga S., Cherry D., D'Antoni J. (2016). Introducing Microsoft SQL Server 2016 Mission-Critical Applications, Deeper Insights, Hyperscale Cloud. Washington. Microsoft Press

#### Atribución-No Comercial-Sin Derivadas

Se permite descargar esta obra y compartirla, siempre y cuando no sea modificado y/o alterado su contenido, ni se comercialice. Referenciarlo de la siguiente manera: Universidad Tecnológica Nacional Facultad Regional Córdoba (S/D). Material para la Tecnicatura Universitaria en Programación, modalidad virtual, Córdoba, Argentina.