UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
Secretaría Académica

Tecnicatura Superior en Programación

Laboratorio de Computación II

Unidad Nº 4: *Creación de Objetos*

Año 2017

Contenido de la Unidad N° 4: Creación de objetos

- Vistas. Procedimientos almacenados. Funciones. Utilidad, definición y sentencias.

Introducción a las Vistas

Una vista ofrece la posibilidad de almacenar una consulta predefinida como un objeto en una base de datos para usarse posteriormente. Las tablas consultadas en una vista se denominan tablas base. Algunos ejemplos habituales de vistas son los siguientes:

- Un subconjunto de las filas o columnas de una tabla base.
- Una unión de dos o más tablas base.
- Una combinación de dos o más tablas base.
- Un resumen estadístico de una tabla base.
- Un subconjunto de otra vista o alguna combinación de vistas y tablas base.

Ventajas de las vistas

Las vistas ofrecen diversas ventajas, como:

- Centrar el interés en los datos de los usuarios: Las vistas crean un entorno controlado que permite el acceso a datos específicos mientras se oculta el resto. Los usuarios pueden tratar la presentación de los datos en una vista de forma similar a como lo hacen en una tabla.
- Enmascarar la complejidad de la base de datos: Las vistas ocultan al usuario la complejidad del diseño de la base de datos. De este modo, los programadores pueden cambiar el diseño sin afectar a la interacción entre el usuario y la base de datos.
- Simplificar la administración de los permisos de usuario: En lugar de conceder a los usuarios permisos para consultar columnas específicas de las tablas base, los propietarios de las bases de datos pueden conceder permisos para que el usuario sólo pueda consultar los datos a través de vistas.
- Mejorar el rendimiento: Las vistas le permiten almacenar los resultados de consultas complejas. Otras consultas pueden utilizar estos resultados resumidos.
- Organizar los datos para exportarse a otras aplicaciones.

Creación de Vistas

Puede crear vistas con el Asistente para creación de vistas, con el Administrador corporativo de SQL Server o con Transact-SQL. Las vistas sólo se pueden crear en la base de datos actual. La sintaxis básica (parcial) para crear una vista es la siguiente:

```
create view NOMBREVISTA as
  SENTENCIASSELECT
   from TABLA;
```

En el siguiente ejemplo creamos la vista "vista_empleados", que es resultado de una combinación en la cual se muestran 4 campos:

```
 create view vista_empleados as
  select (apellido+' '+e.nombre) as nombre,sexo,
   s.nombre as seccion, cantidadhijos
   from empleados as e
   join secciones as s
   on codigo=seccion
```

Para ver la información contenida en la vista creada anteriormente tipeamos:

```
select * from vista_empleados;
```

Podemos realizar consultas a una vista como si se tratara de una tabla:

```
select seccion,count(*) as cantidad
from vista_empleados;
```

Los campos y expresiones de la consulta que define una vista DEBEN tener un nombre. Se debe colocar nombre de campo cuando es un campo calculado o si hay 2 campos con el mismo nombre.

Los nombres de los campos y expresiones de la consulta que define una vista DEBEN ser únicos (no puede haber dos campos o encabezados con igual nombre).

Creamos otra vista de "empleados" denominada "vista_empleados_ingreso" que almacena la cantidad de empleados por año:

```
create view vista_empleados_ingreso (fecha,cantidad)
 as
  select datepart(year,fechaingreso),count(*)
  from empleados
  group by datepart(year,fechaingreso)
```

La diferencia es que se colocan entre paréntesis los encabezados de las columnas que aparecerán en la vista. Si no los colocamos y empleamos la sintaxis vista anteriormente, se emplean los nombres de los campos o alias colocados en el "select" que define la vista. Los nombres que se colocan entre paréntesis deben ser tantos como los campos o expresiones que se definen en la vista.

Existen algunas restricciones para el uso de "create view", a saber:

- no se pueden crear vistas temporales ni crear vistas sobre tablas temporales.
- no se pueden asociar reglas ni valores por defecto a las vistas.
- no puede combinarse con otras instrucciones en un mismo lote.
- Las vistas no pueden hacer referencia a más de 1.024 columnas.

Vistas encriptadas

Podemos ver el texto que define una vista ejecutando el procedimiento almacenado del sistema "**sp_helptext**" seguido del nombre de la vista:

```
sp_helptext NOMBREVISTA;
```

Podemos ocultar el texto que define una vista empleando "with encryption". Creamos una vista con su definición oculta:

```
create view vista_empleados
 with encryption
as
 select (apellido+' '+e.nombre) nombre, sexo, s.nombre,
  seccion, cantidadhijos
 from empleados e
  join secciones as s
  on codigo=seccion
```

Eliminar Vistas.

Para quitar una vista se emplea: `drop view NOMBREVISTA`

Si se elimina una tabla a la que hace referencia una vista, la vista no se elimina, hay que eliminarla explícitamente. Eliminamos la vista denominada "vista_empleados":

```
drop view vista_empleados;
```

Modificar datos de una tabla a través de vistas

Si se modifican los datos de una vista, se modifica la tabla base. Se puede insertar, actualizar o eliminar datos de una tabla a través de una vista, teniendo en cuenta lo siguiente, las modificaciones que se realizan a las vistas:

- no pueden afectar a más de una tabla consultada.
- no se pueden cambiar los campos resultado de un cálculo.
- pueden generar errores si afectan a campos a las que la vista no hace referencia.

Modificar Vistas (alter view)

Para modificar una vista puede hacerlo con "alter view". En el ejemplo siguiente se altera vista_empleados para agregar el campo "domicilio":

```
 alter view vista_empleados
as
  select (apellido+' '+e.nombre) as nombre,sexo,
   s.nombre as seccion, cantidadhijos,domicilio
   from empleados as e
   join secciones as s
   on codigo=seccion
```

Si crea una vista con "select *" y luego agrega campos a la estructura de las tablas involucradas, los nuevos campos no aparecerán en la vista; esto es porque los campos se seleccionan al ejecutar "create view"; debe alterar la vista.

Introducción a los Procedimientos Almacenados

Un procedimiento almacenado es un conjunto de instrucciones a las que se les da un nombre, que se almacena en el servidor. Permiten encapsular tareas repetitivas. 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 pueden 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 empleamos la instrucción "create procedure". La sintaxis básica parcial es:

```
CREATE PROCEDURE nombreprocedimiento
 AS instrucciones;
```

Con las siguientes instrucciones creamos un procedimiento almacenado llamado "pa_libros_limite_stock" que muestra todos los libros de los cuales hay menos de 10 disponibles:

```
CREATE PROC pa_libros_limite_stock
 AS
  SELECT * FROM libros
  WHERE cantidad <=10;
```

Para ejecutar el procedimiento almacenado creado anteriormente tipeamos:

```
EXEC pa_libros_limite_stock;
```

Eliminar procedimientos almacenados

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

```
DROP PROCEDURE nombreprocedimiento;
```

Eliminamos el procedimiento almacenado llamado "pa_libros_autor":

```
DROP PROCEDURE pa_libros_autor;
```

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".

Creamos un procedimiento que recibe el nombre de un autor como parámetro para mostrar todos los libros del autor solicitado:

```
create procedure pa_libros_autor
 @autor varchar(30)
as
 select titulo, editorial,precio
  from libros
  where autor= @autor;
```

El procedimiento se ejecuta colocando "execute" (o "exec") seguido del nombre del procedimiento y un valor para el parámetro (si son más parámetros se separan con comas):

```
exec pa_libros_autor 'Borges';
```

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_libros_autor_editorial @editorial='Planeta', @autor='Richard Bach';
```

Cuando pasamos valores con el nombre del parámetro, el orden en que se colocan puede alterarse.

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

```
create procedure pa_libros_autor_editorial2
 @autor varchar(30)='Richard Bach',
 @editorial varchar(20)='Planeta'
as
 select titulo, autor,editorial,precio
  from libros
  where autor= @autor and
  editorial=@editorial;
```

Si enviamos 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.

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

```
create proc pa_libros_autor_editorial3
 @autor varchar(30) = '%',
 @editorial varchar(30) = '%'
as
```

```
  select titulo,autor,editorial,precio
   from libros
   where autor like @autor and
   editorial like @editorial;
exec pa_libros_autor_editorial3 'P%';
```

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. Creamos un procedimiento almacenado que muestre los títulos, editorial y precio de los libros de un determinado autor (enviado como parámetro de entrada) y nos retorne la suma y el promedio de los precios de todos los libros del autor enviado:

```
create procedure pa_autor_sumaypromedio
 @autor varchar(30)='%',
 @suma decimal(6,2) output,
 @promedio decimal(6,2) output
 as
  select titulo,editorial,precio
  from libros
  where autor like @autor
 select @suma=sum(precio)
  from libros
  where autor like @autor
 select @promedio=avg(precio)
  from libros
  where autor like @autor;
```

Ejecutamos el procedimiento y vemos el contenido de las variables en las que almacenamos los parámetros de salida del procedimiento (Al ejecutarlo también debe emplearse "output"):

```
declare @s decimal(6,2), @p decimal(6,2)
execute pa_autor_sumaypromedio 'Richard Bach', @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. Creamos un procedimiento que muestre todos los libros de un autor determinado que se ingresa como parámetro:

```
create procedure pa_libros_autor
 @autor varchar(30)=null
as
if @autor is null
begin
 select 'Debe indicar un autor'
 return
```

```
end;
select titulo from  libros where autor = @autor;
```

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 se sale; en caso contrario, ejecuta la consulta luego del "else"; "return" puede retornar un valor entero.

Creamos un procedimiento almacenado que ingresa registros en la tabla "libros". Los parámetros correspondientes al título y autor 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 título y autor y "0", en caso que título o autor sean nulos:

```
create procedure pa_libros_ingreso
 @titulo varchar(40)=null,
 @autor varchar(30)=null,
 @editorial varchar(20)=null,
 @precio decimal(5,2)=null
as
if (@titulo is null) or (@autor is null)
 return 0
else
begin
 insert into libros values (@titulo,@autor,@editorial,@precio)
 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_libros_ingreso 'Alicia en el pais...','Lewis Carroll'
select 'Ingreso realizado=1' = @retorno
exec @retorno=pa_libros_ingreso
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_libros_ingreso 'El gato con botas','Anónimo'
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
 PARAMETROS
 with encryption
 as INSTRUCCIONES;
```

Si ejecutamos el procedimiento almacenado del sistema "sp_helptext" para ver su contenido, no aparece.

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;
```

Modificamos el procedimiento almacenado "pa_libros_autor" para que muestre, además, la editorial y precio:

```
alter procedure pa_libros_autor
 @autor varchar(30)=null
as
if @autor is null
begin
 select 'Debe indicar un autor'
 return
end
else
 select titulo,editorial,precio
  from  libros
  where autor = @autor;
```

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_ofertas
as
 select titulo,autor,editorial,precio
 from libros
 where precio<50;
```

La siguiente instrucción ingresa en la tabla "ofertas" el resultado del procedimiento "pa_ofertas":

```
insert into ofertas exec pa_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.

Creamos un procedimiento que nos 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).

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 llamamos a funciones que tienen definidos parámetros de entrada DEBEMOS suministrar SIEMPRE un valor para él. Creamos una función a la cual le enviamos una fecha y nos 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;
```

Podemos colocar un valor por defecto al parámetro, pero al invocar la función, para que tome el valor por defecto DEBEMOS especificar "default".

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

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

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, título, autor y precio de todos los libros cuyo precio sea inferior al parámetro:

```
create function f_ofertas
(@minimo decimal(6,2))
returns @ofertas table-- nombre de la tabla
--formato de la tabla
(codigo int,
 titulo varchar(40),
 autor varchar(30),
 precio decimal(6,2)
)
as
begin
  insert @ofertas
   select codigo,titulo,autor,precio
   from libros
   where precio < @minimo
  return
end;
```

La siguiente consulta realiza un join entre la tabla "libros" y la tabla retornada por la función "f_ofertas":

```
select *from libros as l
 join dbo.f_ofertas(25) as o
 on l.codigo=o.codigo;
```

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

```
select titulo,precio 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 CONDICION
);
```

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.

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

```
create function f_libros
(@autor varchar(30)='Borges')
returns table
as
return (
```

```
  select titulo,editorial
  from libros
  where autor like '%'+@autor+'%'
 );
```

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

```
select *from f_libros('Bach');
```

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)
```

Veamos un ejemplo. Creamos una función que retorna una tabla en línea:

```
create function f_libros
(@autor varchar(30)='Borges')
returns table
as
return (
 select titulo,editorial
 from libros
 where autor like '%'+@autor+'%'
);
```

La modificamos agregando otro campo en el "select":

```
alter table f_libros
(@autor varchar(30)='Borges')
returns table
as
return (
 select codigo,titulo,editorial
```

```
  from libros
  where autor like '%'+@autor+'%'
);
```

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. En funciones con valores de tabla en línea:

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

Eliminamos, si existe, la función denominada "f_fechacadena":

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