126 - Procedimientos almacenados (encriptado)

Dijimos que SQL Server guarda el nombre del procedimiento almacenado en la tabla del sistema "sysobjects" y su contenido en la tabla "syscomments".

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;
```

Esta opción es opcional.

Creamos el procedimiento almacenado "pa_libros_autor" con la opción de encriptado:

```
 create procedure pa_libros_autor
  @autor varchar(30)=null
  with encryption
  as
   select *from libros
    where autor=@autor;
```

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

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('libros') is not null
  drop table libros;

create table libros(
  codigo int identity,
  titulo varchar(40),
  autor varchar(30),
  editorial varchar(20),
```

```
  precio decimal(5,2),
  primary key(codigo)
);

go

-- Eliminamos el procedimiento llamado
"pa_libros_autor", si existe:
if object_id('pa_libros_autor') is not null
  drop procedure pa_libros_autor;

go

-- Creamos el procedimiento almacenado "pa_libros_autor"
con la opción de encriptado:
create procedure pa_libros_autor
  @autor varchar(30)=null
  with encryption
  as
   select *from libros
    where autor=@autor;

go

-- Ejecutamos el procedimiento almacenado del sistema
"sp_helptext" para ver su contenido
-- (no aparece):
exec sp_helptext pa_libros_autor;
```

127 - Procedimientos almacenados (modificar)

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

Un procedimiento almacenado existente puede modificarse con "alter procedure". Sintaxis:

```
 alter procedure NOMBREPROCEDIMIENTO
  @PARAMETRO TIPO = VALORPREDETERMINADO
  as SENTENCIAS;
```

Modificamos el procedimiento almacenado "pa_libros_autor" para que muestre, además del título, 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;
```

Si quiere modificar un procedimiento que se creó con la opción "with encryption" y quiere conservarla, debe incluirla al alterarlo.

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('libros') is not null
  drop table libros;

create table libros(
  codigo int identity,
  titulo varchar(40),
  autor varchar(30),
  editorial varchar(20),
  precio decimal(5,2),
  primary key(codigo)
);

go

insert into libros values ('Uno','Richard
Bach','Planeta',15);
insert into libros values ('Ilusiones','Richard
Bach','Planeta',12);
insert into libros values ('El
aleph','Borges','Emece',25);
insert into libros values ('Aprenda PHP','Mario
Molina','Nuevo siglo',50);
```

```
insert into libros values ('Matematica estas
ahi','Paenza','Nuevo siglo',18);
insert into libros values ('Puente al infinito','Richard
Bach','Sudamericana',14);
insert into libros values ('Antología','J. L.
Borges','Paidos',24);
insert into libros values ('Java en 10 minutos','Mario
Molina','Siglo XXI',45);
insert into libros values
('Antología','Borges','Planeta',34);

if object_id('pa_libros_autor') is not null
  drop procedure pa_libros_autor;

go

-- Creamos el procedimiento almacenado "pa_libros_autor"
con la opción de encriptado
-- para que muestre todos los títulos de los libros cuyo
autor se envía como argumento:
create procedure pa_libros_autor
  @autor varchar(30)=null
  with encryption
  as
   select titulo from libros
    where autor like @autor;

-- Ejecutamos el procedimiento:
exec pa_libros_autor 'Richard Bach';

-- Intentamos ver el contenido del procedimiento (No se
puede porque está encriptado):
exec sp_helptext pa_libros_autor;

go

-- Modificamos el procedimiento almacenado
"pa_libros_autor" para que muestre,
-- además del título, la editorial y precio, quitándole
la encriptación:
alter procedure pa_libros_autor
  @autor varchar(30)=null
  as
   select titulo, editorial, precio from libros
```

```
    where autor like @autor;

go

-- Ejecutamos el procedimiento:
exec pa_libros_autor 'Borges';

-- Veamos el contenido del procedimiento (es posible
porque ya no está encriptado):
exec sp_helptext pa_libros_autor;

go

-- Modificamos el procedimiento almacenado
"pa_libros_autor" para que,
--  en caso de no enviarle un valor, muestre todos los
registros:
alter procedure pa_libros_autor
  @autor varchar(30)='%'
  as
   select titulo, editorial, precio from libros
    where autor like @autor;

go

-- Ejecutamos el procedimiento:
exec pa_libros_autor;
```

128 - Procedimientos almacenados (insertar)

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.

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('libros') is not null
  drop table libros;

create table libros(
  codigo int identity,
  titulo varchar(40),
  autor varchar(30),
  editorial varchar(20),
  precio decimal(5,2),
  primary key(codigo)
);

go

insert into libros values ('Uno','Richard
Bach','Planeta',15);
insert into libros values ('Ilusiones','Richard
Bach','Planeta',12);
insert into libros values ('El
aleph','Borges','Emece',25);
insert into libros values ('Aprenda PHP','Mario
Molina','Nuevo siglo',50);
insert into libros values ('Matematica estas
ahi','Paenza','Nuevo siglo',18);
insert into libros values ('Puente al infinito','Richard
Bach','Sudamericana',14);
insert into libros values ('Antología','J. L.
Borges','Paidos',24);
insert into libros values ('Java en 10 minutos','Mario
Molina','Siglo XXI',45);
insert into libros values
('Antología','Borges','Planeta',34);
```

```
-- Eliminamos la tabla "ofertas" si existe y la creamos
con los mismos
-- campos de la tabla "libros":
if object_id('ofertas') is not null
  drop table ofertas;
create table ofertas(
  titulo varchar(40),
  autor varchar(30),
  editorial varchar(20),
  precio decimal(5,2)
);

go

-- Eliminamos el procedimiento llamado "pa_ofertas", si
existe:
if object_id('pa_ofertas') is not null
  drop procedure pa_ofertas;

go

-- Creamos el procedimiento para que seleccione los
libros
-- cuyo precio no supera los 30 pesos:
create proc pa_ofertas
  as
  select titulo,autor,editorial,precio
   from libros
   where precio<=30;

go

-- Vamos a ingresar en la tabla "ofertas" el resultado
devuelto
-- por el procedimiento almacenado "pa_ofertas":
insert into ofertas exec pa_ofertas;

-- Veamos el contenido de "ofertas":
select * from ofertas;

-- Eliminamos la tabla "libros_por_editorial" si existe
y
```

```
--luego creamos la tabla con dos campos: nombre de
editorial y cantidad:
if object_id('libros_por_editorial') is not null
  drop table libros_por_editorial;
create table libros_por_editorial(
  editorial varchar(20),
  cantidad int
);

go

-- Eliminamos el procedimiento llamado
"pa_libros_por_editorial", si existe:
if object_id('pa_libros_por_editorial') is not null
  drop procedure pa_libros_por_editorial;

go

-- Creamos el procedimiento para que cuente la cantidad
de libros de cada editorial:
create proc pa_libros_por_editorial
  as
  select editorial,count(*)
   from libros
   group by editorial;

go

-- Vamos a ingresar en la tabla "libros_por_editorial"
el resultado devuelto
-- por el procedimiento almacenado
"pa_libros_por_editorial":
insert into libros_por_editorial exec
pa_libros_por_editorial;

-- Veamos el contenido de la tabla
"libros_por_editorial":
select * from libros_por_editorial;
```

129 - Procedimientos almacenados (anidados)

Un procedimiento almacenado puede llamar a otro procedimiento almacenado. El procedimiento que es invocado por otro debe existir cuando

creamos el procedimiento que lo llama. Es decir, si un procedimiento A llama a otro procedimiento B, B debe existir al crear A.

Los procedimientos almacenados pueden anidarse hasta 32 niveles.

Creamos un procedimiento almacenado que reciba 2 números enteros y nos retorne el producto de los mismos:

```
 create procedure pa_multiplicar
  @numero1 int,
  @numero2 int,
  @producto int output
  as
   select @producto=@numero1*@numero2;
```

Creamos otro procedimiento que nos retorne el factorial de un número, tal procedimiento llamará al procedimiento "pa_multiplicar":

```
 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,@resultad
o);
```

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

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
-- Eliminamos, si existen, los procedimientos
almacenados siguientes:
if object_id('pa_multiplicar') is not null
  drop proc pa_multiplicar;
if object_id('pa_factorial') is not null
  drop proc pa_factorial;

go

-- Creamos un procedimiento almacenado que reciba 2
números enteros
-- y nos retorne el producto de los mismos:
create procedure pa_multiplicar
  @numero1 int,
  @numero2 int,
  @producto int output
  as
   select @producto=@numero1*@numero2;

go

-- Probamos el procedimiento anterior:
declare @x int
exec pa_multiplicar 3,9, @x output
select @x as '3*9'
exec pa_multiplicar 50,8, @x output
select @x as '50*8';

go

-- Creamos un procedimiento que nos retorne el factorial
de un número,
-- tal procedimiento llamará al procedimiento
"pa_multiplicar":
create procedure pa_factorial
  @numero int
 as
  if @numero>=0 and @numero<=12
  begin
   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,@resultad
o)
  end
  else select 'Debe ingresar un número entre 0 y 12';

go

-- Ejecutamos el procedimiento que nos retorna el
factorial de un número:
exec pa_factorial 5;
exec pa_factorial 10;

-- Veamos las dependencias del procedimiento
"pa_multiplicar":
exec sp_depends pa_multiplicar;

-- Veamos las dependencias del procedimiento
"pa_factorial":
exec sp_depends pa_factorial;
```

130 - Procedimientos Almacenados (recompilar)

La compilación es un proceso que consiste en analizar el procedimiento almacenado y crear un plan de ejecución. Se realiza la primera vez que se ejecuta un procedimiento almacenado o si el procedimiento almacenado se debe volver a compilar (recompilación).
SQL Server recompila automáticamente un procedimiento almacenado si se realiza algún cambio en la estructura de una tabla (o vista) referenciada en el procedimiento (alter table y alter view) y cuando se modifican las claves (insert o delete) de una tabla referenciada.

Un procedimiento almacenado puede recompilarse explícitamente. En general se recomienda no hacerlo excepto si se agrega un índice a una tabla referenciada por el procedimiento o si los datos han variado mucho desde la última compilación.

SQL Server ofrece tres métodos para recompilar explícitamente un procedimiento almacenado:

1) Se puede indicar, al crear el procedimiento, que SQL Server no guarde en la caché un plan de ejecución para el procedimiento sino que lo compile cada vez que se ejecute.
En este caso la sintaxis es la siguiente:

```
create procedure NOMBREPROCEDIMIENTO
 PARAMETROS
 with recompile
 as
  SENTENCIAS;
```

2) Podemos especificar "with recompile" al momento de ejecutarlo:

```
exec NOMBREPROCEDIMIENTO with recompile;
```

3) Podemos ejecutar el procedimiento almacenado del sistema "sp_recompile". Este procedimiento vuelve a compilar el procedimiento almacenado (o desencadenador) que se especifica. La sintaxis es:

```
exec sp_recompile NOMBREOBJETO;
```

El parámetro enviado debe ser el nombre de un procedimiento, de un desencadenador, de una tabla o de una vista. Si es el nombre de una tabla o vista, todos los procedimientos almacenados que usan tal tabla (o vista) se vuelven a compilar.

131 - Procedimientos Almacenados (con join)

Hasta ahora, hemos creado procedimientos que incluyen una sola tabla o pocas instrucciones para aprender la sintaxis, pero la funcionalidad de un procedimiento consiste básicamente en que contengan muchas instrucciones o instrucciones complejas y así evitar tipear repetidamente dichas instrucciones; además si no queremos que el usuario conozca la estructura de las tablas involucradas, los procedimientos permiten el acceso a ellas.

Podemos crear procedimientos que incluyan combinaciones (join), subconsultas, varias instrucciones y llamadas a otros procedimientos.

Podemos crear todos los procedimientos que necesitemos para que realicen todas las operaciones y consultas.

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
/*
Vamos a crear procedimientos que incluyan combinaciones
(join), subconsultas, varias instrucciones
y llamadas a otros procedimientos.
Un club dicta clases de distintos deportes. Almacena la
información en varias tablas:

- deportes: codigo y nombre,
- cursos: numero de curso, codigo de deporte, documento
del profesor que lo dicta
  y dia de la semana,
- profesores: documento, nombre y domicilio,
- socios: documento, nombre y domicilio,
- inscriptos: documento del socio, número del curso y si
la matricula está paga o no.

Una vez por semana se dicta cada curso.
Puede haber varios cursos de un mismo deporte que se
dicten distintos días y/o por
distintos profesores. Por ejemplo: curso 1 de natación
los lunes por Carlos Caseres,
curso 2 de natación los martes por Carlos Caseres y
curso 3 de natación los miércoles por Ana Acosta.
Un profesor puede estar a cargo de distintos cursos,
incluso de distintos deportes.
Por ejemplo: curso 1 de natación los lunes por Carlos
Caseres y curso 4 de tenis los miércoles por Carlos
Caseres.
Quien se inscriba debe ser socio, es decir, debe estar
en la tabla "socios".
Un socio no puede inscribirse en un mismo curso.
*/

-- Eliminamos las tablas si existen y las creamos:
if (object_id('inscriptos')) is not null
  drop table inscriptos;
if (object_id('deportes')) is not null
  drop table deportes;
if (object_id('cursos')) is not null
```

```
  drop table cursos;
if (object_id('profesores')) is not null
  drop table profesores;
if (object_id('socios')) is not null
  drop table socios;

create table deportes(
  codigo tinyint identity,
  nombre varchar(30),
  primary key (codigo)
);

create table profesores(
  documento char(8),
  nombre varchar(30),
  domicilio varchar(30),
  primary key (documento)
);

create table socios(
  documento char(8),
  nombre varchar(30),
  domicilio varchar(30),
  primary key (documento)
);

create table cursos(
  numero tinyint identity,
  codigodeporte tinyint not null,
  documentoprofesor char(8) not null,
  dia varchar(15),
  constraint PK_cursos_numero
   primary key clustered (numero),
  constraint FK_cursos_documentoprofesor
   foreign key (documentoprofesor)
   references profesores(documento)
   on update cascade,
  constraint FK_cursos_codigodeporte
   foreign key (codigodeporte)
   references deportes(codigo)
);

create table inscriptos(
  documentosocio char(8) not null,
```

```
  numero tinyint not null,
  matricula char(1) --'s'=paga 'n'=impaga,
  constraint PK_inscriptos_documentosocio_numero
   primary key(documentosocio,numero),
  constraint FK_inscriptos_documentosocio
   foreign key (documentosocio)
   references socios(documento),
  constraint FK_inscriptos_numero
   foreign key (numero)
   references cursos(numero)
);

go

-- Ingresamos algunos registros para todas las tablas:
 insert into deportes values('tenis');
 insert into deportes values('natacion');
 insert into deportes values('basquet');
 insert into deportes values('futbol');

 insert into profesores values('22222222','Ana
Acosta','Colon 123');
 insert into profesores values('23333333','Carlos
Caseres','Sarmiento 847');
 insert into profesores values('24444444','Daniel
Duarte','Avellaneda 284');
 insert into profesores values('25555555','Fabiola
Fuentes','Caseros 456');
 insert into profesores values('26666666','Gaston
Garcia','Bulnes 345');

 insert into cursos values(1,'22222222','jueves');
 insert into cursos values(1,'22222222','viernes');
 insert into cursos values(1,'23333333','miercoles');
 insert into cursos values(2,'22222222','miercoles');
 insert into cursos values(2,'23333333','lunes');
 insert into cursos values(2,'23333333','martes');
 insert into cursos values(3,'24444444','lunes');
 insert into cursos values(3,'24444444','jueves');
 insert into cursos values(3,'25555555','martes');
 insert into cursos values(3,'25555555','viernes');
 insert into cursos values(4,'24444444','martes');
 insert into cursos values(4,'24444444','miercoles');
 insert into cursos values(4,'24444444','viernes');
```

```
 insert into socios values('31111111','Luis
Lopez','Colon 464');
 insert into socios values('30000000','Nora
Nores','Bulnes 234');
 insert into socios values('33333333','Mariano
Morales','Sucre 464');
 insert into socios values('32222222','Patricia
Perez','Peru 1234');
 insert into socios values('34444444','Susana
Suarez','Salta 765');

 insert into inscriptos values('30000000',1,'s');
 insert into inscriptos values('30000000',4,'n');
 insert into inscriptos values('31111111',1,'s');
 insert into inscriptos values('31111111',4,'s');
 insert into inscriptos values('31111111',7,'s');
 insert into inscriptos values('31111111',13,'s');
 insert into inscriptos values('32222222',1,'s');
 insert into inscriptos values('32222222',4,'s');

 -- Eliminamos el procedimiento "pa_inscriptos", si
existe:
 if (object_id('pa_inscriptos')) is not null
  drop proc pa_inscriptos;

 go

-- Creamos un procedimiento que muestre el nombre del
socio, el nombre del deporte,
-- el día, el profesor y la matrícula.
-- Si necesitamos esta información frecuentemente, este
procedimiento nos evita tipear
-- este join repetidamente; además si no queremos que el
usuario conozca la estructura
-- de las tablas involucradas, éste y otros
procedimientos permiten el acceso a ellas.
 create procedure pa_inscriptos
  as
   select s.nombre, d.nombre, dia, p.nombre, matricula
   from socios as s
   join inscriptos as i
   on s.documento=i.documentosocio
   join cursos as c
```

```
   on c.numero=i.numero
   join deportes as d
   on c.codigodeporte=d.codigo
   join profesores as p
   on c.documentoprofesor=p.documento;

 go

-- Ejecutamos el procedimiento:
 exec pa_inscriptos;

-- Eliminamos el procedimiento "pa_documentovalido", si
existe:
 if (object_id('pa_documentovalido')) is not null
  drop proc pa_documentovalido;

  go

-- Creamos un procedimiento que reciba un documento y
nos retorne distintos valores según:
-- sea nulo (1), no sea válido (2), no esté en la tabla
"socios" (3), sea un socio deudor
--(4) o sea un socio sin deuda (0)
-- Este procedimiento recibe parámetro, emplea "return"
e incluye subconsultas.
 create procedure pa_documentovalido
  @documento char(8)=null
  as
   if @documento is null return 1
   else
    if len(@documento)<8 return 2
    else
     if not exists (select *from socios where
documento=@documento) return 3
     else
     begin
      if exists (select *from inscriptos
                 where documentosocio=@documento and
                 matricula='n') return 4
      else return 0
     end;

 go
```

```
-- Eliminamos el procedimiento "pa_deportediavalido", si
existe:
 if (object_id('pa_deportediavalido')) is not null
  drop proc pa_deportediavalido;

  go

-- Creamos un procedimiento al cual le enviamos el
nombre de un deporte y el día y
-- nos retorna un valor diferente según: el nombre del
deporte o día sean nulos (1),
-- el día sea inválido (2), deporte no se dicte (3), el
deporte se dicte pero no el
-- día ingresado (4) o el deporte se dicte el día
ingresado (0):
 create procedure pa_deportediavalido
  @deporte varchar(30)=null,
  @dia varchar (15)=null
  as
   if @deporte is null or @dia is null return 1
   else
    if @dia not in
('lunes','martes','miercoles','jueves','viernes','sabado
') return 2
    else
    begin
     declare @coddep tinyint
     select @coddep= codigo from deportes where
nombre=@deporte
     if @coddep is null return 3
     else
      if not exists(select *from cursos where
codigodeporte=@coddep and dia=@dia) return 4
      else return 0
    end;

 go

-- Eliminamos el procedimiento "pa_ingreso", si existe:
 if (object_id('pa_ingreso')) is not null
  drop proc pa_ingreso;

 go
```

```
-- Creamos un procedimiento que nos permita ingresar una
inscripción con los siguientes datos:
-- documento del socio, nombre del deporte, dia y
matrícula.
-- El procedimiento llamará a los procedimientos
"pa_documentovalido" y "pa_deportediavalido"
-- y mostrará diferentes mensajes. Un socio que deba
alguna matrícula NO debe poder
-- inscribirse en ningún curso:
 create procedure pa_ingreso
  @documento char(8)=null,
  @deporte varchar(20)=null,
  @dia varchar(20)=null,
  @matricula char(1)=null
  as
   --verificamos el documento
   declare @doc int
   exec @doc=pa_documentovalido @documento
   if @doc=1 select 'Ingrese un documento'
   else
    if @doc=2 select 'Documento debe tener 8 digitos'
    else
     if @doc=3 select @documento+' no es socio'
     else
      if @doc=4 select 'Socio '+ @documento+' debe
matriculas'
  --verificamos el deporte y el dia
   declare @depdia int
   exec @depdia=pa_deportediavalido @deporte, @dia
   if @depdia=1 select 'Ingrese deporte y dia'
   else
    if @depdia=2 select 'Ingrese día válido'
    else
     if @depdia=3 select @deporte+' no se dicta'
     else
      if @depdia=4 select @deporte+' no se dicta el '+
@dia

   --verificamos que el socio no esté inscripto ya en el
deporte el día solicitado
   if @doc=0 and @depdia=0
   begin
     declare @codcurs int
     select @codcurs=c.numero from cursos as c
```

```
                    join deportes as d
                    on c.codigodeporte=d.codigo
                  where @deporte=d.nombre and
                    @dia=c.dia
     if exists (select *from inscriptos as i
                join cursos as c
                on i.numero=c.numero
                where @codcurs=i.numero and
                i.documentosocio=@documento)
            select 'Ya está inscripto en '+@deporte+' el
'+ @dia
     else
       if @matricula is null or @matricula='s' or
@matricula='n'
       begin
        insert into inscriptos
values(@documento,@codcurs,@matricula)
        print 'Inscripción del socio '+@documento+' para
'+@deporte+' el '+@dia+' realizada'
       end
       else select 'Matricula debe ser s, n o null'
   end;

   go

-- Podemos ejecutar el procedimiento "pa_ingreso" con
distintos valores para ver el resultado.
-- Enviamos un documento que no está en "socios":
 exec pa_ingreso '22222222';

-- Enviamos un documento de un socio que tiene deudas:
 exec pa_ingreso '30000000';

-- Enviamos un documento de un socio que no tiene
deudas, pero falta el deporte y el día:
 exec pa_ingreso '31111111';

-- Enviamos valor de día inválido:
 exec pa_ingreso '31111111','tenis','sabado';

-- Enviamos datos que ya están en la tabla "inscriptos":
 exec pa_ingreso '31111111','tenis','jueves';
```

```
-- Enviamos el documento de un socio y un deporte y día
en el cual no está inscripto:
 exec pa_ingreso '33333333','tenis','jueves';

-- Podemos verificar este ingreso consultando
"pa_inscriptos":
 exec pa_inscriptos;

-- Eliminamos el procedimiento "pa_profesor", si existe:
 if (object_id('pa_profesor')) is not null
  drop proc pa_profesor;

 go

-- Creamos un procedimiento que recibe el documento de
un profesor y nos muestra los
-- distintos deportes de los cuales está a cargo y los
días en que se dictan:
 create proc pa_profesor
  @documento char(8)=null
 as
  if @documento is null or len(@documento)<8
    select 'Ingrese un documento válido'
  else
  begin
   declare @nombre varchar(30)
   select @nombre=nombre from profesores where
documento=@documento
   if @nombre is null select 'No es profesor'
   else
     if not exists(select *from cursos where
documentoprofesor=@documento)
       select 'El profesor '+@nombre+' no tiene cursos
asignados'
     else
       select d.nombre,c.dia
        from cursos as c
        join deportes as d
        on c.codigodeporte=d.codigo
        where c.documentoprofesor=@documento
  end;

 go
```

```
-- Ejecutamos el procedimiento creado anteriormente
enviando un documento que
-- no está en la tabla "profesores":
 exec pa_profesor '34343434';

-- Nuevamente ejecutamos el procedimiento creado
anteriormente, esta vez con
-- un documento existente en "profesores":
 exec pa_profesor '22222222';

-- Eliminamos el procedimiento
"pa_inscriptos_por_curso", si existe:
 if (object_id('pa_inscriptos_por_curso')) is not null
  drop proc pa_inscriptos_por_curso;

 go

-- Creamos un procedimiento que recibe un parámetro
correspondiente al nombre de un deporte
-- y muestra los distintos cursos (número, día y
profesor) y la cantidad de inscriptos;
-- en caso que el parámetro sea "null", muestra la
información de todos los cursos:
 create procedure pa_inscriptos_por_curso
  @deporte varchar(20)=null
 as
  if @deporte is null
    select c.numero,d.nombre,dia,p.nombre,
     (select count(*)
     from inscriptos as i
     where i.numero=c.numero) as cantidad
     from cursos as c
     join deportes as d
     on c.codigodeporte=d.codigo
     join profesores as p
     on p.documento=c.documentoprofesor
  else
    select c.numero,dia,p.nombre,
     (select count(*)
     from inscriptos as i
     where i.numero=c.numero) as cantidad
     from cursos as c
     join deportes as d
     on c.codigodeporte=d.codigo
```

```
     join profesores as p
     on p.documento=c.documentoprofesor
     where d.nombre=@deporte;

 go

-- Ejecutamos el procedimiento sin enviar valor para el
parámetro:
 exec pa_inscriptos_por_curso;

 -- Ejecutamos el procedimiento enviando un valor:
 exec pa_inscriptos_por_curso 'tenis';

-- Ejecutamos el procedimiento enviando otro valor:
 exec pa_inscriptos_por_curso 'voley';

-- Veamos las dependencias. Ejecutamos "sp_depends" con
distintos objetos:
 exec sp_depends socios;

 exec sp_depends profesores;

 exec sp_depends cursos;

 exec sp_depends deportes;

 exec sp_depends inscriptos;

-- Vemos las dependencias de los distintos
procedimientos:
 exec sp_depends pa_documentovalido;

 exec sp_depends pa_inscriptos;

 exec sp_depends pa_deportediavalido;

 exec sp_depends pa_ingreso;

 exec sp_depends pa_profesor;

 exec sp_depends pa_inscriptos_por_curso;
```

132 - Tablas temporales

Las tablas temporales son visibles solamente en la sesión actual.

Las tablas temporales se eliminan automáticamente al acabar la sesión o la función o procedimiento almacenado en el cual fueron definidas. Se pueden eliminar con "drop table".

Pueden ser locales (son visibles sólo en la sesión actual) o globales (visibles por todas las sesiones).

Para crear tablas temporales locales se emplea la misma sintaxis que para crear cualquier tabla, excepto que se coloca un signo numeral (#) precediendo el nombre.

```
create table #NOMBRE(
 CAMPO DEFINICION,
 ...
);
```

Para referenciarla en otras consultas, se debe incluir el numeral(#), que es parte del nombre. Por ejemplo:

```
insert into #libros default values;
select *from #libros;
```

Una tabla temporal no puede tener una restricción "foreign key" ni ser indexada, tampoco puede ser referenciada por una vista.

Para crear tablas temporales globales se emplea la misma sintaxis que para crear cualquier tabla, excepto que se coloca un signo numeral doble (##) precediendo el nombre.

```
create table ##NOMBRE(
 CAMPO DEFINICION,
 ...
);
```

El (o los) numerales son parte del nombre. Así que puede crearse una tabla permanente llamada "libros", otra tabla temporal local llamada "#libros" y una tercera tabla temporal global denominada "##libros".

No podemos consultar la tabla "sysobjects" para ver las tablas temporales, debemos tipear:

```
 select *from tempdb..sysobjects;
```

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
 create table #usuarios(
  nombre varchar(30),
  clave varchar(10)
  primary key(nombre)
);

insert into #usuarios (nombre, clave) values
('Mariano','payaso');

select * from #usuarios;

insert into #usuarios (clave, nombre) values
('River','Juan');

select * from #usuarios;

insert into #usuarios (nombre,clave) values
('Boca','Luis');

select * from #usuarios;
```

133 - Funciones

SQL Server ofrece varios tipos de funciones para realizar distintas operaciones. Hemos visto y empleado varias de ellas.

Se pueden emplear las funciones del sistema en cualquier lugar en el que se permita una expresión en una sentencia "select".

Las funciones pueden clasificarse en:

- deterministicas: siempre retornan el mismo resultado si se las invoca enviando el mismo valor de entrada. Todas las funciones de agregado y string son deterministicas, excepto "charindex" y "patindex".

- no deterministicas: pueden retornar distintos resultados cada vez que se invocan con el mismo valor de entrada. Las siguientes son algunas de las

funciones no deterministicas: getdate, datename, textptr, textvalid, rand. Todas las funciones de configuración, cursor, meta data, seguridad y estadísticas del sistema son no deterministicas.

SQL Server provee muchas funciones y además permite que el usuario pueda definir sus propias funciones.

Sabemos que una función es un conjunto de sentencias que operan como una unidad lógica, una rutina que retorna un valor. Una función tiene un nombre, acepta parámetros de entrada y retorna un valor escalar o una tabla.

Los parámetros de entrada pueden ser de cualquier tipo, excepto timestamp, cursor y table.

Las funciones definidas por el usuario no permiten parámetros de salida.

No todas las sentencias SQL son válidas dentro de una función. NO es posible emplear en ellas funciones no deterministicas (como getdate()) ni sentencias de modificación o actualización de tablas o vistas. Si podemos emplear sentencias de asignación, de control de flujo (if), de modificación y eliminación de variables locales.

SQL Server admite 3 tipos de funciones definidas por el usuario clasificadas según el valor retornado:

1) escalares: retornan un valor escalar;

2) de tabla de varias instrucciones (retornan una tabla) y

3) de tabla en línea (retornan una tabla).

Las funciones definidas por el usuario se crean con la instrucción "create function" y se eliminan con "drop function".

134 - Funciones (drop)

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 (como con cualquier otro objeto):

```
 if object_id('NOMBREPROPIETARIO.NOMBREFUNCION') is not
null
  drop function NOMBREPROPIETARIO.NOMBREFUNCION;
```

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

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

135 - Funciones escalares (crear y llamar)

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. El tipo del valor retornado puede ser de cualquier tipo, excepto text, ntext, image, cursor o timestamp.

Creamos una simple 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;
```

Entonces, luego de "create function" y el nombre de la función, se deben especificar los parámetros de entrada con sus tipos de datos (entre paréntesis), el tipo de dato que retorna luego de "returns", luego de "as" comienza el bloque "begin...end" dentro del cual se encuentran las instrucciones de procesamiento y el valor retornado luego de "return".

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.

Si llamamos a la función anterior sin enviarle los valores para los parámetros:

```
 select dbo.f_promedio();
```

SQL Server muestra un mensaje de error indicando que necesita argumentos.

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;
```

Analicemos: luego de "create function" y el nombre de la función, especificamos los parámetros de entrada con sus tipos de datos (entre paréntesis). El parámetro de entrada tiene definido un valor por defecto.

Luego de los parámetros de entrada se indica el tipo de dato que retorna luego de "returns"; luego de "as" comienza el bloque "begin...end" dentro del cual se encuentran las instrucciones de procesamiento y el valor retornado luego de "return".

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

Recuerde que al invocar una función escalar, se debe especificar el propietario y el nombre de la función:

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

No olvide que cuando invocamos funciones que tienen definidos parámetros de entrada DEBEMOS suministrar SIEMPRE un valor para él.

Podemos colocar un valor por defecto al parámetro, pero al invocar la función, para que tome el valor por defecto DEBEMOS especificar "default". Por ejemplo, si llamamos a la función anterior sin enviarle un valor:

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

SQL Server muestra un mensaje de error indicando que necesita argumento.

Para que tome el valor por defecto debemos enviar "default" como argumento:

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

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

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
-- Especificamos el entorno de idioma para la sesión.
-- El idioma de la sesión determina los formatos de
fecha y hora
--  y los mensajes del sistema.
set language us_english;

-- Una empresa tiene almacenados los datos de sus
empleados en una tabla denominada "empleados".
--Eliminamos la tabla, si existe y la creamos con la
siguiente estructura:
if object_id('empleados') is not null
  drop table empleados;

create table empleados(
  documento char(8) not null,
  nombre varchar(30),
  fechaingreso datetime,
  mail varchar(50),
  telefono varchar(12)
);

go

-- Fijamos el formato de la fecha
set dateformat ymd;

insert into empleados values('22222222', 'Ana
Acosta','1985/10/10','anaacosta@gmail.com','4556677');
```

```
insert into empleados values('23333333', 'Bernardo
Bustos', '1986/02/15',null,'4558877');
insert into empleados values('24444444', 'Carlos
Caseros','1999/12/02',null,null);
insert into empleados values('25555555', 'Diana
Dominguez',null,null,'4252525');

-- Eliminamos, si existe, la función "f_fechaCadena":
if object_id('dbo.f_fechaCadena') is not null
  drop function dbo.f_fechaCadena;

go

-- Creamos una función a la cual le enviamos una fecha
(de tipo varchar),
-- en el cuerpo de la función se analiza si el dato
enviado corresponde a una fecha,
-- si es así, se almacena en una variable el mes (en
español) y se le concatenan el día
-- y el año y se retorna esa cadena; en caso que el
valor enviado no corresponda a una fecha,
-- la función retorna la cadena 'Fecha inválida':
create function f_fechaCadena
 (@fecha varchar(25))
  returns varchar(25)
  as
  begin
    declare @nombre varchar(25)
    set @nombre='Fecha inválida'
    if (isdate(@fecha)=1)
    begin
      set @fecha=cast(@fecha as datetime)
      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
      set @nombre=rtrim(cast(datepart(day,@fecha) as
char(2)))+' de '+@nombre
      set @nombre=@nombre+' de '+
rtrim(cast(datepart(year,@fecha)as char(4)))
    end--si es una fecha válida
    return @nombre
 end;

go

-- Recuperamos los registros de "empleados", mostrando
el nombre y la fecha
-- de ingreso empleando la función creada anteriormente:
select nombre, dbo.f_fechaCadena(fechaingreso) as
ingreso from empleados;

-- Empleamos la función en otro contexto:
select dbo.f_fechaCadena(getdate());
```

136 - Funciones de tabla de varias instrucciones

Hemos visto el primer tipo de funciones definidas por el usuario, que retornan un valor escalar. Ahora veremos las funciones con varias instrucciones que retornan una tabla.

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

Este tipo de función es similar a un procedimiento almacenado; la diferencia es que la tabla retornada por la función puede ser referenciada en el "from" de una consulta, pero el resultado de un procedimiento almacenado no.

También es similar a una vista; pero en las vistas solamente podemos emplear "select", mientras que en funciones definidas por el usuario podemos incluir sentencias como "if", llamadas a funciones, procedimientos, etc.

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
```

Como cualquier otra función, se crea con "create function" seguida del nombre de la función; luego (opcionalmente) los parámetros de entrada con su tipo de dato.

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 cuerpo de la función se define también en un bloque "begin... end", el cual contiene las instrucciones que insertan filas en la variable (tabla que será retornada) definida en "returns". "return" indica que las filas insertadas en la variable son retornadas; no puede ser un argumento.

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;
```

Las funciones que retornan una tabla pueden llamarse sin especificar propietario:

```
 select *from f_ofertas(30);
 select *from dbo.f_ofertas(30);
```

Dijimos que este tipo de función puede ser referenciada en el "from" de una consulta; 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);
```

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('libros') is not null
  drop table libros;

create table libros(
  codigo int identity,
  titulo varchar(40),
  autor varchar(30),
  editorial varchar(20),
  precio decimal(6,2)
);

go
```

```
insert into libros values('Uno','Richard
Bach','Planeta',15);
insert into libros values('Ilusiones','Richard
Bach','Planeta',10);
insert into libros values('El
aleph','Borges','Emece',25);
insert into libros values('Aprenda PHP','Mario
Molina','Siglo XXI',55);
insert into libros values('Alicia en el pais','Lewis
Carroll','Paidos',35);
insert into libros values('Matematica estas
ahi','Paenza','Nuevo siglo',25);

-- Eliminamos la función "f_ofertas" si existe":
if object_id('f_ofertas') is not null
  drop function f_ofertas;

go

-- Creamos la función "f_ofertas" que reciba un
parámetro correspondiente a un precio y
-- nos retorne una tabla con código, 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;

go
```

```
--Llamamos a la función como si fuera una tabla,
recuerde que podemos
-- omitir el nombre del propietario:
select * from f_ofertas(30);

-- Realizamos un join entre "libros" y la tabla
retornada por la función
-- "f_ofertas" y mostramos todos los campos de "libros".
-- Incluimos una condición para el autor:
select l.titulo,l.autor,l.editorial
  from libros as l
  join dbo.f_ofertas(25) as o
  on l.codigo=o.codigo
  where l.autor='Richard Bach';

-- La siguiente consulta nos retorna algunos campos de
la tabla
--retornada por "f_ofertas" y algunos registros que
cumplen
-- con la condición "where":
select titulo,precio from f_ofertas(40)
  where autor like '%B%';

-- Eliminamos la función "f_listadolibros" si existe":
if object_id('f_listadolibros') is not null
  drop function f_listadolibros;

go

-- Creamos otra función que retorna una tabla:
create function f_listadolibros
 (@opcion varchar(10)
 )
 returns @listado table
 (titulo varchar(40),
 detalles varchar(60)
 )
 as
 begin
  if @opcion not in ('autor','editorial')
    set @opcion='autor'
  if @opcion='editorial'
   insert @listado
```

```
    select titulo,
   (editorial+'-'+autor) from libros
   order by 2
  else
    if @opcion='autor'
     insert @listado
     select titulo,
     (autor+'-'+editorial) from libros
     order by 2
  return
end;

go

-- Llamamos a la función enviando el valor "autor":
select * from dbo.f_listadolibros('autor');

-- Llamamos a la función enviando el valor "editorial":
select * from dbo.f_listadolibros('editorial');

-- Llamamos a la función enviando un valor inválido:
select * from dbo.f_listadolibros('precio');
```

137 - 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 más flexible en el empleo de parámetros. En una vista no se puede incluir un parámetro, lo que hacemos es agregar una cláusula "where" al ejecutar la vista. Las funciones con valores de tabla en línea funcionan como una vista con parámetros.

Sintaxis:

```
 create function NOMBREFUNCION
 (@PARAMETRO TIPO=VALORPORDEFECTO)
 returns table
 as
 return (
  select CAMPOS
  from TABLA
  where CONDICION
```

```
 );
```

Como todas las funciones definidas por el usuario, se crea con "create function" seguido del nombre que le damos a la función; luego declaramos los parámetros de entrada con su tipo de dato entre paréntesis. 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');
```

Recuerde a que todas las funciones que tienen definidos parámetros se les DEBE suministrar valores para ellos al invocarse.

Recuerde que para que el parámetro tome el valor por defecto (si lo tiene) DEBE enviarse "default" al llamar a la función; si no le enviamos parámetros, SQL Server muestra un mensaje de error.

```
--incorrecto: select *from f_libros();
 select *from f_libros(default);--correcto
```

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('libros') is not null
  drop table libros;

create table libros(
  codigo int identity,
  titulo varchar(40),
  autor varchar(30),
  editorial varchar(20)
);

go

insert into libros values('Uno','Richard
Bach','Planeta');
insert into libros values('El aleph','Borges','Emece');
insert into libros values('Ilusiones','Richard
Bach','Planeta');
insert into libros values('Aprenda PHP','Mario
Molina','Nuevo siglo');
insert into libros values('Matematica estas
ahi','Paenza','Nuevo siglo');

-- Eliminamos, si existe la función "f_libros":
if object_id('f_libros') is not null
  drop function f_libros;

go

-- 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+'%'
```

```
 );

go

-- Llamamos a la función creada anteriormente enviando
un autor:
select * from f_libros('Bach');

-- Eliminamos, si existe la función
"f_libros_autoreditorial":
if object_id('f_libros_autoreditorial') is not null
  drop function f_libros_autoreditorial;

go

-- Creamos una función con valores de tabla en línea que
recibe dos parámetros:
create function f_libros_autoreditorial
 (@autor varchar(30)='Borges',
 @editorial varchar(20)='Emece')
 returns table
 as
 return (
  select titulo,autor,editorial
  from libros
  where autor like '%'+@autor+'%' and
  editorial like '%'+@editorial+'%'
 );

go

-- Llamamos a la función creada anteriormente:
select * from f_libros_autoreditorial('','Nuevo siglo');

-- Llamamos a la función creada anteriormente enviando
"default"
-- para que tome los valores por defecto:
select * from f_libros_autoreditorial(default,default);
```

138 - Funciones (modificar)

Las funciones de SQL Server no pueden ser modificadas, las funciones definidas por el usuario si.

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

Sintaxis general:

```
alter function PROPIETARIO.NOMBREFUNCION
NUEVADEFINICION;
```

Sintaxis para modificar funciones escalares:

```
alter function 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+'%'
 );
```

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('libros') is not null
  drop table libros;

create table libros(
  codigo int identity,
  titulo varchar(40),
  autor varchar(30),
  editorial varchar(20)
);

go

insert into libros values('Uno','Richard
Bach','Planeta');
insert into libros values('El aleph','Borges','Emece');
insert into libros values('Ilusiones','Richard
Bach','Planeta');
insert into libros values('Aprenda PHP','Mario
Molina','Nuevo siglo');
```

```
insert into libros values('Matematica estas
ahi','Paenza','Nuevo siglo');

if object_id('f_libros') is not null
  drop function f_libros;

go

-- 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+'%'
 );

go

-- Llamamos a la función creada anteriormente enviando
un autor:
select * from f_libros('Bach');

go

-- La modificamos agregando otro campo en el "select":
alter function f_libros
 (@autor varchar(30)='Borges')
 returns table
 as
 return (
  select codigo,titulo,editorial
  from libros
  where autor like '%'+@autor+'%'
 );

go

-- Probamos la función para ver si se ha modificado:
select * from f_libros('Bach');
```

139 - Funciones (encriptado)

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:

```
create function NOMBREFUNCION
(@PARAMETRO TIPO)
 returns TIPO
 with encryption
 as
 begin
  CUERPO
  return EXPRESION
 end
```

En funciones de tabla de varias sentencias se coloca luego del formato de la tabla a retornar:

```
create function NOMBREFUNCION
(@PARAMETRO TIPO)
returns @NOMBRETABLARETORNO table-- nombre de la tabla
--formato de la tabla
(CAMPO1 TIPO,
 CAMPO2 TIPO,
 CAMPO3 TIPO
)
with encryption
as
begin
  insert @NOMBRETABLARETORNO
   select CAMPOS from TABLA
    where campo OPERADOR @PARAMETRO
  RETURN
end
```

En funciones con valores de tabla en línea:

```
create function NOMBREFUNCION
(@PARAMETRO TIPO=VALORPORDEFECTO)
```

```
returns table
with encryption
as
return (SELECT);
```

Veamos un ejemplo:

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

140 - Funciones (información)

Las funciones son objetos, así que para obtener información de ellos pueden usarse los siguientes procedimientos almacenados del sistema y las siguientes tablas:

- "sp_help": sin parámetros nos muestra todos los objetos de la base de datos seleccionada, incluidas las funciones definidas por el usuario. En la columna "Object_type" aparece "scalar function" si es una función escalar, "table function" si es una función de tabla de varias sentencias y "inline function" si es una función de tabla en línea.

Si le enviamos como argumento el nombre de una función definida por el usuario, obtenemos el propietario, el tipo de función y la fecha de creación; si es una función de tabla, los campos de la tabla retornada.

- "sp_helptext": seguido del nombre de una función definida por el usuario nos muestra el texto que define la función, excepto si ha sido encriptado.

- "sp_stored_procedures": muestra todos los procedimientos almacenados y funciones definidas por el usuario.

- "sp_depends": seguido del nombre de un objeto, nos devuelve 2 resultados: 1) nombre, tipo, campos, etc. de los objetos de los cuales depende el objeto enviado (referenciados por el objeto) y 2) nombre y tipo

de los objetos que dependen del objeto nombrado (que lo referencian). Por ejemplo, ejecutamos "sp_depends" seguido del nombre de una función definida por el usuario:

```
exec sp_depends pa_libroslistado;
```

aparecen las tablas (y demás objetos) de las cuales depende el procedimiento, es decir, las tablas (y campos) referenciadas en la misma. No aparecen objetos que dependan de la función porque no existe ningún objeto que la referencie.

Podemos ejecutar el procedimiento seguido del nombre de una tabla:

```
exec sp_depends libros;
```

aparecen las funciones (y demás objetos) que dependen de ella (que la referencian). No aparecen objetos de los cuales depende porque la tabla no los tiene.

- La tabla del sistema "sysobjects": muestra nombre y varios datos de todos los objetos de la base de datos actual. La columna "xtype" indica el tipo de objeto. Si es una función definida por el usuario escalar, muestra "FN", si es una función de tabla de varias sentencias, muestra "TF" y si es una función de tabla en linea muestra "IF".

Si queremos ver el nombre, tipo y fecha de creación de todas las funciones definidas por el usuario, podemos tipear:

```
select name,xtype as tipo,crdate as fecha
 from sysobjects
 where xtype in ('FN','TF','IF');
```

141 - Disparadores (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.

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.

Sintaxis básica:

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

```
  SENTENCIAS
```

Analizamos la sintaxis:

- "create trigger" junto al nombre del disparador.

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

- luego de "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.

- luego de "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.

Consideraciones generales:

- "create trigger" debe ser la primera sentencia de un bloque y sólo se puede aplicar a una tabla.

- un disparador 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".

A continuación veremos la creación de un disparador para el suceso de inserción: "insert trigger".

142 - Disparador de inserción (insert trigger)

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

Sintaxis básica:

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

Analizamos la sintaxis:

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

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

Luego de "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.

Creamos un trigger sobre la tabla "ventas" para el evento de inserción. Cada vez que se realiza un "insert" sobre "ventas", el disparador se ejecuta. El disparador controla que la cantidad que se intenta vender sea menor o igual al stock del libro y actualiza el campo "stock" de "libros", restando al valor anterior la cantidad vendida:

```
create trigger DIS_ventas_insertar
 on ventas
 for insert
 as
  declare @stock int
  select @stock= stock from libros
            join inserted
            on inserted.codigolibro=libros.codigo
            where libros.codigo=inserted.codigolibro
 if (@stock>=(select cantidad from inserted))
   update libros set stock=stock-inserted.cantidad
    from libros
    join inserted
    on inserted.codigolibro=libros.codigo
    where codigo=inserted.codigolibro
 else
 begin
```

```
    raiserror ('Hay menos libros en stock de los
solicitados para la venta', 16, 1)
    rollback transaction
  end
```

Entonces, creamos el disparador ("create trigger") dándole un nombre ("DIS_ventas_insertar") sobre ("on") una tabla específica ("ventas") para ("for") el suceso de inserción ("insert"). Luego de "as" colocamos las sentencias, las acciones que el trigger realizará cuando se ingrese un registro en "ventas" (en este caso, controlar que haya stock y disminuir el stock de "libros").

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 "libros", la cantidad ingresada en el nuevo registro de "ventas", 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.

La instrucción "writetext" no activa un disparador.

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('ventas') is not null
  drop table ventas;
```

```
if object_id('libros') is not null
  drop table libros;

create table libros(
  codigo int identity,
  titulo varchar(40),
  autor varchar(30),
  precio decimal(6,2),
  stock int,
  constraint PK_libros primary key(codigo)
);

create table ventas(
  numero int identity,
  fecha datetime,
  codigolibro int not null,
  precio decimal (6,2),
  cantidad int,
  constraint PK_ventas primary key(numero),
  constraint FK_ventas_codigolibro
   foreign key (codigolibro) references libros(codigo)
);

go

insert into libros values('Uno','Richard Bach',15,100);
insert into libros values('Ilusiones','Richard
Bach',18,50);
insert into libros values('El aleph','Borges',25,200);
insert into libros values('Aprenda PHP','Mario
Molina',45,200);

go

-- Creamos un disparador para que se ejecute cada vez
que una instrucción "insert"
-- ingrese datos en "ventas"; el mismo controlará que
haya stock en "libros"
-- y actualizará el campo "stock":
create trigger DIS_ventas_insertar
  on ventas
  for insert
  as
   declare @stock int
```

```
   select @stock= stock from libros
             join inserted
             on inserted.codigolibro=libros.codigo
             where libros.codigo=inserted.codigolibro
  if (@stock>=(select cantidad from inserted))
    update libros set stock=stock-inserted.cantidad
     from libros
     join inserted
     on inserted.codigolibro=libros.codigo
     where codigo=inserted.codigolibro
  else
  begin
    raiserror ('Hay menos libros en stock de los
solicitados para la venta', 16, 1)
    rollback transaction
  end

go

set dateformat ymd;

-- Ingresamos un registro en "ventas":
insert into ventas values('2018/04/01',1,15,1);
-- Al ejecutar la sentencia de inserción anterior, se
disparó el trigger, el registro
-- se agregó a la tabla del disparador ("ventas") y
disminuyó el valor del campo "stock"
-- de "libros".

-- Verifiquemos que el disparador se ejecutó consultando
la tabla "ventas" y "libros":
select * from ventas;
select * from libros where codigo=1;

-- Ingresamos un registro en "ventas", solicitando una
cantidad superior al stock
-- (El disparador se ejecuta y muestra un mensaje, la
inserción no se realizó porque
-- la cantidad solicitada supera el stock.):
 insert into ventas values('2018/04/01',2,18,100);

 -- Finalmente probaremos ingresar una venta con un
código de libro inexistente
```

```
 -- (El trigger no llegó a ejecutarse, porque la
comprobación de restricciones
 -- (que se ejecuta antes que el disparador) detectó que
la infracción a la "foreign key"):
 insert into ventas values('2018/04/01',5,18,1);
```

143 - Disparador de borrado (delete trigger)

Podemos 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
```

Analizamos la sintaxis:

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

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

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

El disparador del siguiente ejemplo se crea para la tabla "ventas", para que cada vez que se elimine un registro de "ventas", se actualice el campo "stock" de la tabla "libros" (por ejemplo, si el comprador devuelve los libros comprados):

```
 create trigger DIS_ventas_borrar
  on ventas
  for delete
 as
   update libros set stock=
libros.stock+deleted.cantidad
   from libros
   join deleted
```

```
   on deleted.codigolibro=libros.codigo;
```

Entonces, creamos el disparador ("create trigger") dándole un nombre ("DIS_ventas_borrar") sobre ("on") una tabla específica ("ventas") para ("for") el evento de borrado ("delete"). Luego de "as" colocamos las sentencias, las acciones que el trigger realizará cuando se elimine un registro en "ventas" (en este caso, aumentar el stock de "libros").

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".

El siguiente disparador se crea para controlar que no se elimine más de un registro de la tabla "libros". El disparador se activa cada vez que se elimina un registro o varios, controlando la cantidad de registros que se están eliminando; si se está eliminando más de un registro, el disparador retorna un mensaje de error y deshace la transacción:

```
create trigger DIS_libros_borrar
 on libros
 for delete
 as
  if (select count(*) from deleted) > 1
  begin
   raiserror('No puede borrar más de un libro',16,1)
   rollback transaction
  end;
```

Si se ejecuta un "delete" sobre "libros" que afecte a varios registros, se activa el disparador y evita la transacción.

Si se ejecuta el siguiente "delete", que afecta a un solo registro, se activa el disparador y permite la transacción:

```
delete from libros where codigo=5;
```

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

`

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('ventas') is not null
  drop table ventas;
if object_id('libros') is not null
  drop table libros;

create table libros(
  codigo int identity,
  titulo varchar(40),
  autor varchar(30),
  editorial varchar(20),
  precio decimal(6,2),
  stock int,
  constraint PK_libros primary key(codigo)
);

create table ventas(
  numero int identity,
  fecha datetime,
  codigolibro int not null,
  precio decimal (6,2),
  cantidad int,
  constraint PK_ventas primary key(numero),
  constraint FK_ventas_codigolibro
   foreign key (codigolibro) references libros(codigo)
   on delete no action
 );

go

insert into libros values('Uno','Richard
Bach','Planeta',15,100);
insert into libros values('Ilusiones','Richard
Bach','Planeta',18,50);
insert into libros values('El
aleph','Borges','Emece',25,200);
insert into libros values('Aprenda PHP','Mario
Molina','Emece',45,200);

set dateformat ymd;
```

```
insert into ventas values('2018/01/01',1,15,1);
insert into ventas values('2018/01/01',2,18,2);

go

-- Creamos un disparador para actualizar el campo
"stock" de la tabla "libros"
-- cuando se elimina un registro de la tabla "ventas"
--(por ejemplo, si el comprador devuelve los libros
comprados):
create trigger DIS_ventas_borrar
  on ventas
  for delete
 as
   update libros set stock=
libros.stock+deleted.cantidad
   from libros
   join deleted
   on deleted.codigolibro=libros.codigo;

go

--Eliminamos un registro de "ventas":
delete from ventas where numero=2;
-- Al ejecutar la sentencia de eliminación anterior, se
disparó el trigger,
--el registro se eliminó de la tabla del disparador
("ventas") y
-- se actualizó el stock en "libros"

-- Verifiquemos que el disparador se ejecutó consultando
la tabla "libros"
-- y vemos si el stock aumentó:
select * from libros where codigo=2;

-- Verificamos que el registro se eliminó de "ventas":
select * from ventas;

go

-- Creamos un disparador para controlar que no se
elimine más de un registro
```

```
-- de la tabla "libros". El disparador se activa cada
vez que se ejecuta un "delete"
-- sobre "libros", controlando la cantidad de registros
que se están eliminando;
-- si se está eliminando más de un registro, el
disparador retorna un mensaje
--de error y deshace la transacción:
create trigger DIS_libros_borrar
  on libros
  for delete
  as
   if (select count(*) from deleted) > 1
   begin
    raiserror('No puede eliminar más de un libro',16,1)
    rollback transaction
   end;

go

-- Solicitamos la eliminación de varios registros de
"libros"
-- (Se activa el disparador y deshace la transacción):
delete from libros where editorial='Emece';

-- Solicitamos la eliminación de un solo libro
-- (Se activa el disparador y permite la transacción):
delete from libros where codigo=4;

-- Consultamos la tabla y vemos que el libro fue
eliminado:
select * from libros;
```

144 - Disparador de actualización (update trigger)

Podemos crear un disparador para que se ejecute siempre que una instrucción "update" actualice los datos de una tabla.

Sintaxis básica:

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

```
   SENTENCIAS
```

Analizamos la sintaxis:

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

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

Luego de "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.

El siguiente disparador de actualización se crea para evitar que se modifiquen los datos de la tabla "libros":

```
 create trigger DIS_libros_actualizar
  on libros
  for update
  as
    raiserror('Los datos de la tabla "libros" no pueden
modificarse', 10, 1)
    rollback transaction
```

Entonces, creamos el disparador ("create trigger") dándole un nombre ("DIS_libros_actualizar") sobre una tabla específica ("libros") para ("for") el suceso de actualización ("update"). Luego de "as" colocamos las sentencias, las acciones que el trigger realizará cuando se intente actualizar uno o varios registros en "libros" (en este caso, impedir las modificaciones).

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".

Creamos un disparador que evite que se actualice el campo "precio" de la tabla "libros":

```
 create trigger DIS_libros_actualizar_precio
  on libros
  for update
  as
   if update(precio)
   begin
    raiserror('El precio de un libro no puede
modificarse.', 10, 1)
    rollback transaction
   end;
```

Empleamos "if update()" para que el trigger controle la actualización del campo "precio"; así, cuando el disparador detecte una actualización en tal campo, realizará las acciones apropiadas (mostrar un mensaje y deshacer la actualización); en caso que se actualice otro campo, el disparador se activa, pero permite la transacción.

Creamos un disparador de actualización que muestra el valor anterior y nuevo valor de los registros actualizados:

```
 create trigger DIS_libros_actualizar2
  on libros
  for update
  as
   if (update(titulo) or update(autor) or
update(editorial)) and
    not (update(precio) or update(stock))
   begin
    select d.codigo,
    (d.titulo+'-'+ d.autor+'-'+d.editorial) as 'registro
anterior',
    (i.titulo+'-'+ i.autor+'-'+i.editorial) as 'registro
actualizado'
     from deleted as d
     join inserted as i
     on d.codigo=i.codigo
   end
   else
   begin
    raiserror('El precio y stock no pueden modificarse.
La actualización no se realizó.', 10, 1)
    rollback transaction
   end;
```

Empleamos "if update" para que el trigger controle si la actualización se realiza en ciertos campos permitidos (titulo, autor y editorial) y no en los campos prohibidos (precio y stock)); si se modifican los campos permitidos y ninguno de los no permitidos, mostrará los antiguos y nuevos valores consultando las tablas "deleted" e "inserted", en caso que se actualice un campo no permitido, el disparador muestra un mensaje y deshace la transacción.

Note que el disparador no controla los intentos de actualización sobre el campo "codigo", esto es porque tal campo, no puede modificarse porque está definido "identity", si intentamos modificarlo, SQL Server muestra un mensaje de error y el trigger no llega a dispararse.

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('libros') is not null
  drop table libros;

create table libros(
  codigo int identity,
  titulo varchar(40),
  autor varchar(30),
  editorial varchar(20),
  precio decimal(6,2),
  stock int,
  constraint PK_libros primary key(codigo)
);

go

insert into libros values('Uno','Richard
Bach','Planeta',15,100);
insert into libros values('Alicia en el pais...','Lewis
Carroll','Planeta',18,50);
insert into libros values('El
aleph','Borges','Emece',25,200);
insert into libros values('Aprenda PHP','Mario
Molina','Nuevo siglo',45,200);

go
```

```
-- Creamos un disparador para evitar que se modifiquen
los datos de la tabla "libros":
create trigger DIS_libros_actualizar
  on libros
  for update
  as
    raiserror('Los datos de la tabla "libros" no pueden
modificarse', 10, 1)
    rollback transaction;

go

-- Intentamos realizar alguna actualización en "libros":
update libros set titulo='Alicia en el pais de las
maravillas' where codigo=2;
-- El disparador se activó, mostró un mensaje y deshizo
la actualización.

-- Eliminamos el disparador creado anteriormente:
drop trigger DIS_libros_actualizar;

go

-- Creamos un disparador que evite que se actualice el
campo "precio" de la tabla "libros":
create trigger DIS_libros_actualizar_precio
  on libros
  for update
  as
   if update(precio)
   begin
    raiserror('El precio de un libro no puede
modificarse.', 10, 1)
    rollback transaction
   end;

go

-- Veamos qué sucede si intentamos actualizar el precio
de un libro:
update libros set precio=30 where codigo=2;
-- El disparador se activa, muestra un mensaje y deshace
la transacción.
```

```
-- Veamos qué sucede al actualizar el campo "titulo":
update libros set titulo='Alicia en el pais de las
maravillas' where codigo=2;
-- El disparador se activa y realiza la transacción

-- Lo verificamos consultando la tabla:
select * from libros;

-- Veamos qué sucede si intentamos actualizar el precio
y la editorial de un libro:
update libros set precio=30,editorial='Emece' where
codigo=1;
-- El disparador se activa, muestra un mensaje y deshace
la transacción;
--el registro no fue actualizado.

-- Lo verificamos consultando la tabla:
select * from libros;

-- Eliminamos el disparador creado anteriormente:
drop trigger DIS_libros_actualizar_precio;

go

-- Creamos un disparador de actualización que muestra el
valor anterior y nuevo valor de los
-- registros actualizados. El trigger debe controlar que
la actualización se realice en los
-- campos "titulo", "autor" y "editorial" y no en los
demás campos (precio y stock));
-- si se modifican los campos permitidos y ninguno de
los no permitidos, mostrará los antiguos
-- y nuevos valores consultando las tablas "deleted" e
"inserted", en caso que se actualice
-- un campo no permitido, el disparador muestra un
mensaje y deshace la transacción:
create trigger DIS_libros_actualizar2
  on libros
  for update
  as
   if (update(titulo) or update(autor) or
update(editorial)) and
    not (update(precio) or update(stock))
   begin
```

```
    select (d.titulo+'-'+ d.autor+'-'+d.editorial) as
'registro anterior',
    (i.titulo+'-'+ i.autor+'-'+i.editorial) as 'registro
actualizado'
     from deleted as d
     join inserted as i
     on d.codigo=i.codigo
   end
   else
   begin
    raiserror('El precio y stock no pueden modificarse.
La actualización no se realizó.', 10, 1)
    rollback transaction
   end;

go

-- Veamos qué sucede si modificamos campos permitidos:
update libros set editorial='Paidos',
autor='Desconocido' where codigo>3;
-- El trigger se dispara y muestra los registros
modificados, los valores antes
-- y después de la transacción.

-- Veamos qué sucede si en la sentencia "update"
intentamos modificar algún campo no permitido:
update libros set editorial='Paidos', precio=30 where
codigo>3;
-- El trigger se dispara y muestra el mensaje de error,
la transacción no se realizó.

-- Intentamos modificar el código de un libro:
 update libros set codigo=9 where codigo>=3;
-- El disparador no llega a dispararse porque SQL Server
muestra un mensaje de error ya que el
-- campo "codigo", por ser "identity", no puede
modificarse.
```

145 - Disparadores (varios eventos)

Hemos aprendido a crear disparadores para diferentes eventos (insert, update y delete).

Dijimos que un disparador puede definirse para más de una acción; en tal caso, deben separarse con comas.

Creamos un trigger para evitar que se inscriban socios que deben matrículas y no permitir que se eliminen las inscripciones de socios deudores. El trigger se define para ambos eventos en la misma sentencia de creación.

```
 create trigger dis_inscriptos_insert_delete
  on inscriptos
  for insert,delete
  as
   if exists (select *from inserted join morosos
              on morosos.documento=inserted.documento)
   begin
     raiserror('El socio es moroso, no puede inscribirse
en otro curso', 16, 1)
     rollback transaction
   end
   else
     if exists (select *from deleted join morosos
               on morosos.documento=deleted.documento)
     begin
       raiserror('El socio debe matriculas, no puede
borrarse su inscripcion', 16, 1)
       rollback transaction
     end
     else
      if (select matricula from inserted)='n'
       insert into morosos select documento from
inserted;
```

El trigger controla:

- si se intenta ingresar una inscripción de un socio moroso, se deshace la transacción;

- si se intenta eliminar una inscripción de un socio que está en "morosos", se deshace la transacción;

- si se ingresa una nueva inscripción y no se paga la matrícula, dicho socio se ingresa a la tabla "morosos".

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('inscriptos') is not null
  drop table inscriptos;
if object_id('socios') is not null
  drop table socios;
if object_id('morosos') is not null
  drop table morosos;

create table socios(
  documento char(8) not null,
  nombre varchar(30),
  domicilio varchar(30),
  constraint PK_socios primary key(documento)
);

create table inscriptos(
  numero int identity,
  documento char(8) not null,
  deporte varchar(20),
  matricula char(1),
  constraint FK_inscriptos_documento
   foreign key (documento)
   references socios(documento),
  constraint CK_inscriptos_matricula check (matricula in
('s','n')),
  constraint PK_inscriptos primary
key(documento,deporte)
);

create table morosos(
  documento char(8) not null
);

go

insert into socios values('22222222','Ana
Acosta','Avellaneda 800');
insert into socios values('23333333','Bernardo
Bustos','Bulnes 345');
```

```
insert into socios values('24444444','Carlos
Caseros','Colon 382');
insert into socios values('25555555','Mariana
Morales','Maipu 234');

insert into inscriptos values('22222222','tenis','s');
insert into inscriptos
values('22222222','natacion','n');
insert into inscriptos values('23333333','tenis','n');
insert into inscriptos values('24444444','futbol','s');
insert into inscriptos
values('24444444','natacion','s');

insert into morosos values('22222222');
insert into morosos values('23333333');

go

-- Creamos un trigger para evitar que se inscriban
socios que deben matrículas y
-- no permitir que se eliminen las inscripciones de
socios deudores.
-- El trigger se define para ambos eventos en la misma
sentencia de creación.
create trigger dis_inscriptos_insert_delete
  on inscriptos
  for insert,delete
  as
   if exists (select *from inserted join morosos
              on morosos.documento=inserted.documento)
   begin
     raiserror('El socio es moroso, no puede inscribirse
en otro curso', 16, 1)
     rollback transaction
   end
   else
     if exists (select *from deleted join morosos
               on morosos.documento=deleted.documento)
     begin
       raiserror('El socio debe matriculas, no puede
borrarse su inscripcion', 16, 1)
       rollback transaction
     end
     else
```

```
      if (select matricula from inserted)='n'
       insert into morosos select documento from
inserted;

go

-- Ingresamos una inscripción de un socio no deudor con
matrícula paga:
insert into inscriptos values('25555555','tenis','s');
-- El disparador se activa ante el "insert" y permite la
transacción.

-- Ingresamos una inscripción de un socio no deudor con
matrícula 'n':
insert into inscriptos
values('25555555','natacion','n');
-- El disparador se activa ante el "insert", permite la
transacción y agrega
-- al socio en la tabla "morosos".

--Verifiquémoslo consultando las tablas
correspondientes:
select * from inscriptos;
select * from morosos;

-- Ingresamos una inscripción de un socio deudor:
insert into inscriptos values('25555555','basquet','s');
-- El disparador se activa ante el "insert" y deshace la
transacción porque
-- encuentra su documento en la tabla "morosos".

-- Eliminamos una inscripción de un socio no deudor:
delete inscriptos where numero=4;
-- El disparador se activa ante la sentencia "delete" y
permite la transacción.

-- Verificamos que la inscripción nº 4 fue eliminada de
"inscriptos":
select * from inscriptos;

-- Intentamos eliminar una inscripción de un socio
deudor:
delete inscriptos where numero=6;
```

```
-- El disparador se activa ante el "delete" y deshace la
transacción porque
-- encuentra su documento en "morosos".
```

146 - Disparador (Instead Off y after)

Hasta el momento hemos aprendido que un trigger se crea sobre una tabla específica para un evento (inserción, eliminación o actualización).

También podemos especificar el momento de disparo del trigger. El momento de disparo indica que las acciones (sentencias) del trigger se ejecuten luego de la acción (insert, delete o update) que dispara el trigger o en lugar de la acción.

La sintaxis para ello es:

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

Entonces, el momento de disparo especifica cuando deben ejecutarse las acciones (sentencias) que realiza el trigger. Puede ser "después" (after) o "en lugar" (instead of) del evento que lo dispara.

Si no especificamos 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. Hasta el momento, todos los disparadores que creamos han sido "after".

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.

Veamos un ejemplo. Una empresa almacena los datos de sus empleados en una tabla "empleados" y en otra tabla "clientes" los datos de sus clientes. Se crea una vista que muestra los datos de ambas tablas:

```
 create view vista_empleados_clientes
 as
```

```
  select documento,nombre, domicilio, 'empleado' as
condicion from empleados
  union
   select documento,nombre, domicilio,'cliente' from
clientes;
```

Creamos un disparador sobre la vista "vista_empleados_clientes" para inserción, que redirija las inserciones a la tabla correspondiente:

```
 create trigger DIS_empleadosclientes_insertar
  on vista_empleados_clientes
  instead of insert
  as
    insert into empleados
     select documento,nombre,domicilio
     from inserted where condicion='empleado'

    insert into clientes
     select documento,nombre,domicilio
     from inserted where condicion='cliente';
```

El disparador anterior especifica que cada vez que se ingresen registros en la vista "vista_empleados_clientes", en vez de (instead of) realizar la acción (insertar en la vista), se ejecuten las sentencias del trigger, es decir, se ingresen los registros en las tablas correspondientes.

Entonces, las opciones de disparo pueden ser:

a) "after": el trigger se dispara cuando las acciones especificadas (insert, delete y/o update) son ejecutadas; todas las acciones en cascada de una restricción "foreign key" y las comprobaciones de restricciones "check" deben realizarse con éxito antes de ejecutarse el trigger. Es la opción por defecto si solamente colocamos "for" (equivalente a "after").

La sintaxis es:

```
 create trigger NOMBREDISPARADOR
  on NOMBRETABLA
  after | for-- son equivalentes
  ACCION-- insert, update o delete
 as
  SENTENCIAS
```

b) "instead of": sobreescribe la acción desencadenadora del trigger. Se puede definir solamente un disparador de este tipo para cada acción (insert, delete o update) sobre una tabla o vista.

Sintaxis:

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

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.

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('empleados') is not null
  drop table empleados;
if object_id('clientes') is not null
  drop table clientes;
```

```
create table empleados(
  documento char(8) not null,
  nombre varchar(30),
  domicilio varchar(30),
  constraint PK_empleados primary key(documento)
);

create table clientes(
  documento char(8) not null,
  nombre varchar(30),
  domicilio varchar(30),
  constraint PK_clientes primary key(documento)
);

-- Eliminamos la siguiente vista:
if object_id('vista_empleados_clientes') is not null
  drop view vista_empleados_clientes;

go

-- Creamos una vista que muestra los datos de ambas
tablas:
create view vista_empleados_clientes
 as
  select documento,nombre, domicilio, 'empleado' as
condicion from empleados
  union
   select documento,nombre, domicilio,'cliente' from
clientes;

go

-- Creamos un disparador sobre la vista
"vista_empleados_clientes" para inserción,
-- que redirija las inserciones a la tabla
correspondiente:
create trigger DIS_empl_clie_insertar
  on vista_empleados_clientes
  instead of insert
  as
    insert into empleados
     select documento,nombre,domicilio
     from inserted where condicion='empleado'
```

```
    insert into clientes
     select documento,nombre,domicilio
     from inserted where condicion='cliente';

go

-- Ingresamos un empleado y un cliente en la vista:
insert into vista_empleados_clientes
values('22222222','Ana Acosta', 'Avellaneda
345','empleado');
insert into vista_empleados_clientes
values('23333333','Bernardo Bustos', 'Bulnes
587','cliente');

-- Veamos si se almacenaron en la tabla correspondiente:
select * from empleados;
select * from clientes;

go

-- Creamos un disparador sobre la vista
"vista_empleados_clientes" para el evento "update",
-- que redirija las actualizaciones a la tabla
correspondiente:
create trigger DIS_empl_clie_actualizar
  on vista_empleados_clientes
  instead of update
  as
   declare @condicion varchar(10)
   set @condicion = (select condicion from inserted)
   if update(documento)
   begin
    raiserror('Los documentos no pueden modificarse',
10, 1)
    rollback transaction
   end
   else
   begin
    if @condicion ='empleado'
    begin
     update empleados set
empleados.nombre=inserted.nombre,
empleados.domicilio=inserted.domicilio
     from empleados
```

```
     join inserted
     on empleados.documento=inserted.documento
    end
    else
     if @condicion ='cliente'
     begin
      update clientes set
clientes.nombre=inserted.nombre,
clientes.domicilio=inserted.domicilio
      from clientes
      join inserted
      on clientes.documento=inserted.documento
     end
   end;

go

-- Realizamos una actualización sobre la vista, de un
empleado:
update vista_empleados_clientes set nombre= 'Ana Maria
Acosta' where documento='22222222';

-- Veamos si se actualizó la tabla correspondiente:
select * from empleados;

-- Realizamos una actualización sobre la vista, de un
cliente:
update vista_empleados_clientes set domicilio='Bulnes
1234' where documento='23333333';

-- Veamos si se actualizó la tabla correspondiente:
select * from clientes;
```

147 - Disparador (eliminar)

Los triggers se eliminan con la instrucción "drop trigger":

```
 drop trigger NOMBREDISPARADOR;
```

Si el disparador que se intenta eliminar no existe, aparece un mensaje indicándolo, para evitarlo, podemos verificar su existencia antes de solicitar su eliminación (como con cualquier otro objeto):

```
 if object_id('NOMBREDISPARADOR') is not null
  drop trigger NOMBREDISPARADOR;
```

Eliminamos, si existe, el trigger "dis_libros_insertar":

```
 if object_id('dis_libros_insertar') is not null
  drop trigger dis_libros_insertar;
```

Cuando se elimina una tabla o vista que tiene asociados triggers, todos los triggers asociados se eliminan automáticamente.

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('ventas') is not null
  drop table ventas;
if object_id('libros') is not null
  drop table libros;

create table libros(
  codigo int identity,
  titulo varchar(40),
  autor varchar(30),
  precio decimal(6,2),
  stock int,
  constraint PK_libros primary key(codigo)
);

create table ventas(
  numero int identity,
  fecha datetime,
  codigolibro int not null,
  precio decimal (6,2),
  cantidad int,
  constraint PK_ventas primary key(numero),
  constraint FK_ventas_codigolibro
   foreign key (codigolibro) references libros(codigo)
);

go

insert into libros values('Uno','Richard Bach',15,100);
```

```
insert into libros values('Ilusiones','Richard
Bach',18,50);
insert into libros values('El aleph','Borges',25,200);
insert into libros values('Aprenda PHP','Mario
Molina',45,200);

go

-- Creamos un disparador para que se ejecute cada vez
que una instrucción "insert"
-- ingrese datos en "ventas"; el mismo controlará que
haya stock en "libros"
-- y actualizará el campo "stock":
create trigger DIS_ventas_insertar
  on ventas
  for insert
  as
   declare @stock int
   select @stock= stock from libros
             join inserted
             on inserted.codigolibro=libros.codigo
             where libros.codigo=inserted.codigolibro
  if (@stock>=(select cantidad from inserted))
    update libros set stock=stock-inserted.cantidad
     from libros
     join inserted
     on inserted.codigolibro=libros.codigo
     where codigo=inserted.codigolibro
  else
  begin
    raiserror ('Hay menos libros en stock de los
solicitados para la venta', 16, 1)
    rollback transaction
  end

go

set dateformat ymd;

-- Ingresamos un registro en "ventas":
insert into ventas values('2018/04/01',1,15,1);
-- Al ejecutar la sentencia de inserción anterior, se
disparó el trigger, el registro
```

```
-- se agregó a la tabla del disparador ("ventas") y
disminuyó el valor del campo "stock"
-- de "libros".

-- Verifiquemos que el disparador se ejecutó consultando
la tabla "ventas" y "libros":
select * from ventas;
select * from libros where codigo=1;

-- Eliminamos trigger
drop trigger DIS_ventas_insertar;

insert into ventas values('2018/04/01',1,15,1);

-- Verifiquemos las tabla "ventas" y "libros":
select * from ventas;
select * from libros where codigo=1;
```

148 - Disparador (información)

Los triggers (disparadores) son objetos, así que para obtener información de ellos pueden usarse los siguientes procedimientos almacenados del sistema y las siguientes tablas:

- "sp_help": sin parámetros nos muestra todos los objetos de la base de datos seleccionada, incluidos los triggers. En la columna "Object_type" aparece "trigger" si es un disparador.

Si le enviamos como argumento el nombre de un disparador, obtenemos el propietario, el tipo de objeto y la fecha de creación.

- "sp_helptext": seguido del nombre de un disparador nos muestra el texto que define el trigger, excepto si ha sido encriptado.

- "sp_depends": retorna 2 resultados:

1) el nombre, tipo, campos, etc. de los objetos de los cuales depende el objeto enviado (referenciados por el objeto) y

2) nombre y tipo de los objetos que dependen del objeto nombrado (que lo referencian).

Por ejemplo, ejecutamos "sp_depends" seguido del nombre de un disparador:

```
 sp_depends dis_inscriptos_insertar;
```

Aparece una tabla similar a la siguiente:

```
name               type        updated     column
-------------------------------------------------
---------
dbo.condicionales user table  yes   codigocurso
dbo.condicionales user table  yes   fecha
dbo.inscriptos          user table  yes   numerocurso
dbo.inscriptos          user table  yes   fecha
dbo.condicionales user table  yes   documento
dbo.cursos        user table  no    numero
dbo.cursos        user table  no    cantidadmaxima
dbo.inscriptos          user table  yes   documento
```

En la columna "name" nos muestra las tablas (y demás objetos si hubiese) de las cuales depende el trigger, es decir, las tablas referenciadas en el mismo; el tipo de objeto en la columna "type" (en este caso, todas tablas); la columna "update" indica si el objeto es actualizado o no (note que la tabla "cursos" no se actualiza, solamente se consulta); la columna "column" muestra el nombre del campo que se referencia.

No aparecen objetos que dependen del trigger porque no existe ningún objeto que lo referencie.

También podemos ejecutar el mismo procedimiento seguido del nombre de una tabla:

```
 sp_depends inscriptos;
```

aparecen los objetos que dependen de ella (que la referencian). En este ejemplo: 1 solo objeto, su nombre y tipo (trigger). No aparecen objetos de los cuales depende porque la tabla no los tiene.

- Para conocer los disparadores que hay en una tabla específica y sus acciones respectivas, podemos ejecutar el procedimiento del sistema "sp_helptrigger" seguido del nombre de la tabla o vista. Por ejemplo:

```
 sp_helptrigger inscriptos;
```

Nos muestra la siguiente información:

```
trigger_name            trigger_owner     isupdate
      isdelete    isinsert    isafter     isinsteadof
-------------------------------------------------------
-------------------------------------------------
dis_inscriptos_insertar dbo         0           0
      1           0     1
```

El nombre del trigger, su propietario; en las 3 columnas siguientes indica para qué evento se ha definido (un valor 1 indica que está definido para tal evento); las 2 últimas columnas indican el momento de disparo (un valor 1 se interpreta como verdadero y un 0 como falso). En el ejemplo, el disparador "dis_inscriptos_insertar" está definido para el evento de inserción (valor 1 en "isinsert") y es "instead of" (valor 1 en "isinsteadof").

- La tabla del sistema "sysobjects": muestra nombre y varios datos de todos los objetos de la base de datos actual. La columna "xtype" indica el tipo de objeto. Si es un trigger muestra "TR".

Si queremos ver el nombre, tipo y fecha de creación de todos los disparadores, podemos tipear:

```
 select name,xtype as tipo,crdate as fecha
  from sysobjects
  where xtype = 'TR';
```

149 - Disparador (modificar)

Los triggers pueden modificarse y eliminarse.

Al modificar la definición de un disparador se reemplaza la definición existente del disparador por la nueva definición.

La sintaxis general es la siguiente:

```
 alter trigger NOMBREDISPARADOR
  NUEVADEFINICION;
```

Asumiendo que hemos creado un disparador llamado "dis_empleados_borrar" que no permitía eliminar más de 1 registro de la tabla empleados; alteramos el disparador, para que cambia la cantidad de eliminaciones permitidas de 1 a 3:

```
 alter trigger dis_empleados_borrar
  on empleados
  for delete
  as
   if (select count(*) from deleted)>3--antes era 1
   begin
    raiserror('No puede borrar mas de 3 empleados',16,
1)
    rollback transaction
   end;
```

Se puede cambiar el evento del disparador. Por ejemplo, si creó un disparador para "insert" y luego se modifica el evento por "update", el disparador se ejecutará cada vez que se actualice la tabla.

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('empleados') is not null
  drop table empleados;

create table empleados(
  documento char(8) not null,
  nombre varchar(30) not null,
  domicilio varchar(30),
  constraint PK_empleados primary key(documento),
);

go

insert into empleados values('22000000','Ana
Acosta','Avellaneda 56');
insert into empleados values('23000000','Bernardo
Bustos','Bulnes 188');
insert into empleados values('24000000','Carlos
Caseres','Caseros 364');
insert into empleados values('25555555','Diana
Duarte','Colon 1234');
insert into empleados values('26666666','Diana
Duarte','Colon 897');
insert into empleados values('27777777','Matilda
Morales','Colon 542');
```

```
go

-- Creamos un disparador para que no permita eliminar
más de un registro a
-- la vez de la tabla empleados:
create trigger dis_empleados_borrar
  on empleados
  for delete
 as
  if (select count(*) from deleted)>1
  begin
    raiserror('No puede eliminar más de un 1 empleado',
16, 1)
    rollback transaction
  end;

go

-- Eliminamos 1 empleado (El trigger se dispara y
realiza la eliminación):
delete from empleados where documento ='22000000';

-- Intentamos eliminar varios empleados
-- (El trigger se dispara, muestra un mensaje y deshace
la transacción.):
delete from empleados where documento like '2%';

select * from empleados;

go

-- Alteramos el disparador, para que cambia la cantidad
de eliminaciones
-- permitidas de 1 a 3:
alter trigger dis_empleados_borrar
  on empleados
  for delete
  as
   if (select count(*) from deleted)>3--antes era 1
   begin
    raiserror('No puede borrar más de 3 empleados',16,
1)
    rollback transaction
```

```
   end;

go

-- Intentamos eliminar 5 empleados (El trigger se
dispara,
-- muestra el nuevo mensaje y deshace la transacción.):
delete from empleados where documento like '2%';

-- Eliminamos 3 empleados (El trigger se dispara y
-- realiza las eliminaciones solicitadas):
delete from empleados where domicilio like 'Colon%';

select * from empleados;
```

150 - disparador (deshabilitar y habilitar)

Se puede deshabilitar o habilitar un disparador específico de una tabla o vista, o todos los disparadores que tenga definidos.

Si se deshabilita un disparador, éste sigue existiendo, pero al ejecutar una instrucción "insert", "update" o "delete" en la tabla, no se activa.

Sintaxis para deshabilitar o habilitar un disparador:

```
 alter table NOMBRETABLA
  ENABLE | DISABLE trigger NOMBREDISPARADOR;
```

El siguiente ejemplo deshabilita un trigger:

```
 alter table empleados
  disable trigger dis_empleados_borrar;
```

Se pueden deshabilitar (o habilitar) varios disparadores en una sola sentencia, separando sus nombres con comas. El siguiente ejemplo deshabilitamos dos triggers definidos sobre la tabla empleados:

```
 alter table empleados
  disable trigger dis_empleados_actualizar,
dis_empleados_insertar;
```

Sintaxis para habilitar (o deshabilitar) todos los disparadores de una tabla específica:

```
 alter table NOMBRETABLA
  ENABLE | DISABLE TRIGGER all;
```

La siguiente sentencia habilita todos los triggers de la tabla "empleados":

```
 alter table empleados
  enable trigger all;
```

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('empleados') is not null
  drop table empleados;

create table empleados(
  documento char(8) not null,
  nombre varchar(30) not null,
  domicilio varchar(30),
  seccion varchar(20),
  constraint PK_empleados primary key(documento),
);

go

insert into empleados values('22222222','Ana
Acosta','Bulnes 56','Secretaria');
insert into empleados values('23333333','Bernardo
Bustos','Bulnes 188','Contaduria');
insert into empleados values('24444444','Carlos
Caseres','Caseros 364','Sistemas');
insert into empleados values('25555555','Diana
Duarte','Colon 1234','Sistemas');
insert into empleados values('26666666','Diana
Duarte','Colon 897','Sistemas');
insert into empleados values('27777777','Matilda
Morales','Colon 542','Gerencia');

go

-- Creamos un disparador para que no permita eliminar
más de un registro a la vez
-- de la tabla empleados:
create trigger dis_empleados_borrar
```

```
  on empleados
  for delete
 as
  if (select count(*) from deleted)>1
  begin
    raiserror('No puede eliminar más de un 1 empleado',
16, 1)
    rollback transaction
  end;

go

-- Creamos un disparador para que no permita actualizar
el campo "documento"
-- de la tabla "empleados":
create trigger dis_empleados_actualizar
  on empleados
  for update
 as
  if update(documento)
  begin
    raiserror('No puede modificar el documento de los
empleados', 16, 1)
    rollback transaction
  end;

go

-- Creamos un disparador para que no permita ingresar
empleados
-- en la sección "Gerencia":
create trigger dis_empleados_insertar
  on empleados
  for insert
 as
  if (select seccion from inserted)='Gerencia'
  begin
    raiserror('No puede ingresar empleados en la sección
"Gerencia".', 16, 1)
    rollback transaction
  end;

go
```

```
-- Intentamos borrar varios empleados (El trigger se
dispara, muestra el
-- mensaje y deshace la transacción):
delete from empleados where domicilio like 'Bulnes%';

go
-- Deshabilitamos el trigger para el evento de
eliminación:
alter table empleados
  disable trigger dis_empleados_borrar;

go

-- Borramos varios empleados (El trigger no se disparó
porque está deshabilitado):
delete from empleados where domicilio like 'Bulnes%';

-- Podemos verificar que los registros de eliminaron
recuperando los datos de la tabla:
select * from empleados;

-- Intentamos modificar un documento (El trigger se
dispara, muestra el mensaje
-- y deshace la transacción):
update empleados set documento='23030303' where
documento='23333333';

-- Intentamos ingresar un nuevo empleado en "Gerencia"
(El trigger se dispara,
-- muestra el mensaje y deshace la transacción.):
insert into empleados values('28888888','Juan
Juarez','Jamaica 123','Gerencia');

go

-- Deshabilitamos los disparadores de inserción y
actualización definidos sobre "empleados":
alter table empleados
  disable trigger dis_empleados_actualizar,
dis_empleados_insertar;

go
```

```
-- Ejecutamos la sentencia de actualización del
documento (El trigger no se dispara porque
-- está deshabilitado, el documento se actualizó):
update empleados set documento='20000444' where
documento='24444444';

-- Verifiquémoslo:
select * from empleados;

-- Ingresar un nuevo empleado en "Gerencia" (El trigger
"dis_empleados_insertar"
-- no se dispara porque está deshabilitado):
insert into empleados values('28888888','Juan
Juarez','Jamaica 123','Gerencia');

-- Verifiquémoslo:
select * from empleados;

go

-- Habilitamos todos los triggers de la tabla
"empleados":
alter table empleados
  enable trigger all;

go
-- Ya no podemos eliminar más de un registro, actualizar
un documento
-- ni ingresar un empleado en la sección "Gerencia"; lo
intentamos
-- (El trigger se dispara (está habilitado), muestra el
mensaje y deshace la transacción):
update empleados set documento='30000000' where
documento='28888888';
```

151 - Disparador (with encryption)

Hasta el momento hemos aprendido que un trigger se crea sobre una tabla (o vista), especificando el momento de ejecución (after o instead of), para un evento (inserción, eliminación o actualización).

Podemos encriptar los triggers para evitar que sean leídos con "sp_helptext". Para ello debemos agregar al crearlos la opción "with encryption" luego del nombre de la tabla o vista:

```
 create triggre NOMBREDISPARADOR
  on NOMBRETABLAoVISTA
  with encryption
  MOMENTODEDISPARO--after o instead of
  ACCION-- insert, update, delete
  as
    SENTENCIAS
```

El siguiente disparador se crea encriptado:

```
 create trigger DIS_empleados_insertar
  on empleados
  with encryption
  after insert
 as
  if (select seccion from inserted)='Gerencia'
  begin
    raiserror('No puede ingresar empleados en la sección
"Gerencia".', 16, 1)
    rollback transaction
  end;
```

Si ejecutamos el procedimiento almacenado del sistema "sp_helptext" seguido del nombre del trigger creado anteriormente, SQL Server mostrará un mensaje indicando que tal disparador ha sido encriptado.

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('empleados') is not null
  drop table empleados;

create table empleados(
  documento char(8) not null,
  nombre varchar(30) not null,
  domicilio varchar(30),
  seccion varchar(20),
  constraint PK_empleados primary key(documento),
```

```
);

go

-- Creamos el siguiente disparador encriptado:
create trigger DIS_empleados_insertar
  on empleados
  with encryption
  after insert
 as
  if (select seccion from inserted)='Gerencia'
  begin
    raiserror('No puede ingresar empleados en la sección
"Gerencia".', 16, 1)
    rollback transaction
  end;

go

-- Ejecutamos el procedimiento almacenado del sistema
"sp_helptext"
-- seguido del nombre del trigger creado anteriormente
--(SQL Server muestra un mensaje indicando que tal
disparador ha sido encriptado):
exec sp_helptext dis_empleados_insertar;

go

-- Modificamos el disparador para quitar la
encriptación:
alter trigger dis_empleados_insertar
  on empleados
  after insert
 as
  if (select seccion from inserted)='Gerencia'
  begin
    raiserror('No puede ingresar empleados en la sección
"Gerencia".', 16, 1)
    rollback transaction
  end;

-- Ejecutamos el procedimiento almacenado del sistema
"sp_helptext"
```

```
-- seguido del nombre del trigger (SQL Server nos
permite ver la
-- definición del trigger porque ya no está
encriptado.):
exec sp_helptext dis_empleados_insertar;
```

152 - Disparador (condicionales)

Una instrucción "insert", "update" o "delete" que invoque a un disparador puede afectar a varios registros. En tales casos, un trigger rechaza o acepta cada transacción de modificación como una totalidad. Podemos optar por:

1) procesar todos los registros: todos los registros afectados deberán cumplir los criterios del disparador para que se produzca la acción, o

2) permitir acciones condicionales: puede definir un disparador que controle si cada registro afectado cumple con la condición; si algún registro no la cumple, la acción no se produce para tal registro pero si para los demás que si la cumplen.

Veamos un ejemplo. Tenemos la tabla "libros". Creamos un disparador de actualización sobre la tabla "libros". Se permite actualizar el stock de varios libros a la vez; pero ningún "stock" debe tener un valor negativo. Entonces, si algún "stock" queda con un valor negativo, no debe cambiar, los demás si:

```
 create trigger dis_libros_actualizar
 on libros
 after update
 as
  if exists (select *from inserted where stock<0)
  begin
   update libros set stock=deleted.stock
   from libros
   join deleted
   on deleted.codigo=libros.codigo
   join inserted
   on inserted.codigo=libros.codigo
   where inserted.stock<0;
  end;
```

No podemos revertir la transacción con "rollback transaction" porque en ese caso TODOS los registros modificados volverían a los valores anteriores, y

lo que necesitamos es que solamente aquellos que quedaron con valor negativo vuelvan a su valor original.

Tampoco podemos evitar que se actualicen todos los registros porque se actualizan antes que las acciones del trigger se ejecuten.

Lo que hacemos es, en el cuerpo del trigger, averiguar si alguno de los registros actualizados tiene stock negativo; si es así, volvemos a actualizarlo al valor anterior a la transacción.

Servidor de SQL Server instalado en forma local.

Ingresemos el siguiente lote de comandos en el SQL Server Management Studio:

```
if object_id('libros') is not null
  drop table libros;

create table libros(
  codigo int identity,
  titulo varchar(40),
  autor varchar(30),
  editorial varchar(20),
  stock int,
  constraint pk_libros primary key (codigo)
);

go

insert into libros values('Uno','R. Bach','Planeta',50);
insert into libros values('Ilusiones','R.
Bach','Planeta',15);
insert into libros values('El
aleph','Borges','Emece',10);
insert into libros values('Aprenda PHP','M.
Molina','Nuevo siglo',5);

go

-- Creamos un disparador de actualización sobre la tabla
"libros".
-- Se permite actualizar el stock de varios libros a la
vez; pero ningún "stock"
```

```
-- debe tener un valor negativo. Si algún "stock" queda
con un valor negativo,
-- no debe cambiar, los demás si:
create trigger dis_libros_actualizar
 on libros
 after update
 as
  if exists (select *from inserted where stock<0)
  begin
   update libros set stock=deleted.stock
   from libros
   join deleted
   on deleted.codigo=libros.codigo
   join inserted
   on inserted.codigo=libros.codigo
   where inserted.stock<0
  end;

-- Actualizamos el stock de todos los libros,
restándoles 15:
update libros set stock=stock-15;

-- Veamos el resultado:
select * from libros;
-- Solamente se actualizaron los 2 primeros libros, cuyo
valor
-- de stock era igual o superior a 15; los otros libros
no se actualizaron.
```