	UNIVERSIDAD DON BOSCO **FACULTAD DE ESTUDIOS TECNOLOGICOS** **ESCUELA DE COMPUTACION**
CICLO 1-2016	**CLASE Nº10**
	Tema: Vistas, Implementación de Procedimientos almacenados y desencadenadores **Materia**: Base de datos **Docente:** Blanca Iris Cañas

Vistas

Las vistas tiene una tendencia a ser utilizadas mucho o poco: en raras ocasiones se utilizan en su justa medida. Las vistas se podrían utilizar para:

a. Reducir la complejidad aparente de la base de datos para los usuarios finales

b. Prevenir la selección de columnas confidenciales a la vez que permite el acceso a otros datos importantes En el fondo, una vista no es más que una consulta almacenada. lo extraordinario es que podemos combinar y hacer corresponder datos desde tablas base (o desde otras vistas) para crear lo que, en general, funciona como cualquier otra tabla base.

Al crear una vista, Microsoft SQL Server comprueba la existencia de los objetos a los que se hace referencia en su definición. El nombre de la vista debe ajustarse a las normas para los identificadores. Opcionalmente, es posible especificar un nombre de propietario para la vista. Debe establecer una convención de denominación coherente para distinguir las vistas de las tablas. Por ejemplo, puede agregar la palabra “vista” como sufijo de cada objeto vista que cree. De este modo podrá distinguir fácilmente entre objetos similares (tablas y vistas) al consultar la vista **INFORMATION_SCHEMA.TABLES**.

Sintaxis

CREATE VIEW *nombreVista* [(*columna* [,*n*])] [WITH {ENCRYPTION | SCHEMABINDING | VIEW_METADATA} [,n]] AS
instrucciónSelect

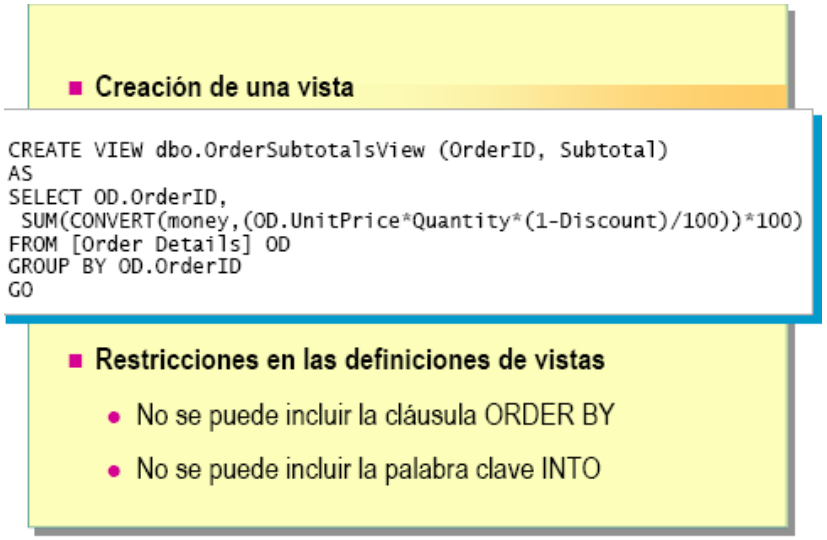
[WITH CHECK OPTION]

Para poder ejecutar la instrucción CREATE VIEW es necesario ser miembro de la función de administradores del sistema (**sysadmin**), de la función propietario de la base de datos (**db_owner**) o de la función administrador de lenguaje de definición de datos (**db_ddladmin**), o bien tener el permiso CREATE VIEW. También es necesario tener el permiso SELECT en todas las tablas o vistas a las que la vista haga referencia.

Para evitar situaciones en las que el propietario de una vista y el propietario de las tablas subyacentes sean distintos, se recomienda que el usuario **dbo** (propietario de base de datos) sea el propietario de todos los objetos de la base de datos. Especifique siempre el usuario **dbo** como propietario al crear el objeto pues, de lo contrario, usted, es decir, su nombre de usuario, será el propietario.

El contenido de una vista se especifica con una instrucción SELECT. Con algunas excepciones, las vistas pueden ser tan complejas como se requiera. Debe especificar los nombres de columna en las situaciones siguientes:

- Alguna de las columnas de la vista se deriva de una expresión aritmética, de una función integrada o de una constante.
- Hay columnas con el mismo nombre en las tablas que se van a combinar.



Ejemplo

Este es un ejemplo de una vista que crea una columna (**Subtotal**) que calcula los subtotales de un pedido a partir de las columnas **UnitPrice**, **Quantity** y **Discount** de la tabla **Order Details**.

```
CREATE VIEW dbo.OrderSubtotalsView (OrderID, Subtotal)
AS
      SELECT OD.OrderID,
      SUM(CONVERT (money,(OD.UnitPrice*Quantity*(1-Discount)/100))*100)
      FROM [Order Details] OD
      GROUP BY OD.OrderID
GO
```

En este ejemplo se consulta la vista para ver los resultados.

```
SELECT * FROM OrderSubtotalsView
```

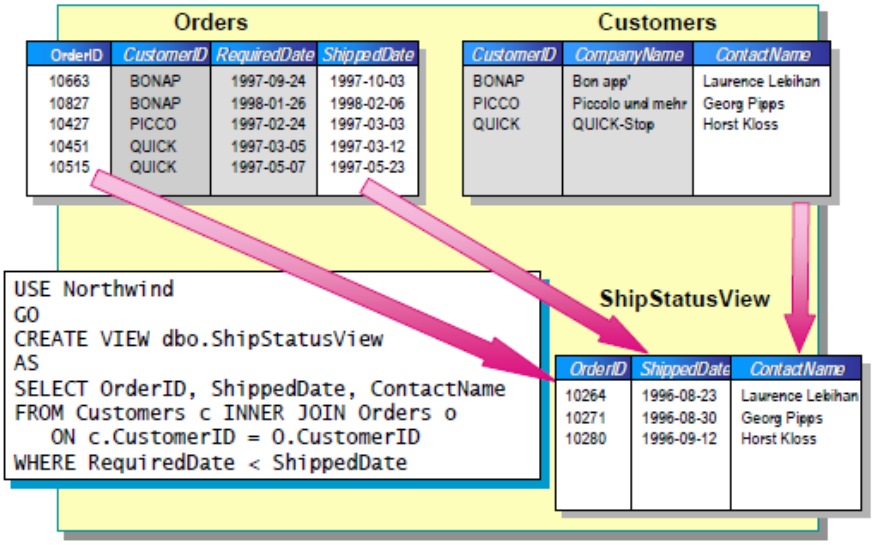
Resultado

	OrderID	Subtotal
1	10248	440.00
2	10249	1863.40
3	10250	1552.60
4	10251	654.06
5	10252	3597.90
6	10253	1444.80
7	10254	556.62
8	10255	2490.50

El resultado da un total de 830 filas

Vista de tablas combinadas

A menudo se crean vistas para conseguir una forma más práctica de ver información centralizada de dos o más tablas combinadas.




```
CREATE VIEW dbo.ShipStatusView
AS
      SELECT OrderID, ShippedDate, ContactName
      FROM Customers c INNER JOIN Orders o
      ON c.CustomerID=o.CustomerID
      WHERE RequiredDate<ShippedDate
GO
```

Resultado

	OrderID	ShippedDate	ContactName
1	10280	1996-09-12 00:00:00.000	Christina Berglund
2	10924	1998-04-08 00:00:00.000	Christina Berglund
3	10970	1998-04-24 00:00:00.000	Martín Sommer
4	10827	1998-02-06 00:00:00.000	Laurence Lebihan
5	10663	1997-10-03 00:00:00.000	Laurence Lebihan
6	10578	1997-07-25 00:00:00.000	Victoria Ashworth
7	10726	1997-12-05 00:00:00.000	Ann Devon
8	10264	1996-08-23 00:00:00.000	Maria Larsson
9	10807	1998-01-30 00:00:00.000	Paolo Accorti
10	10777	1998-01-21 00:00:00.000	André Fonseca

El resultado da un total de 37 filas

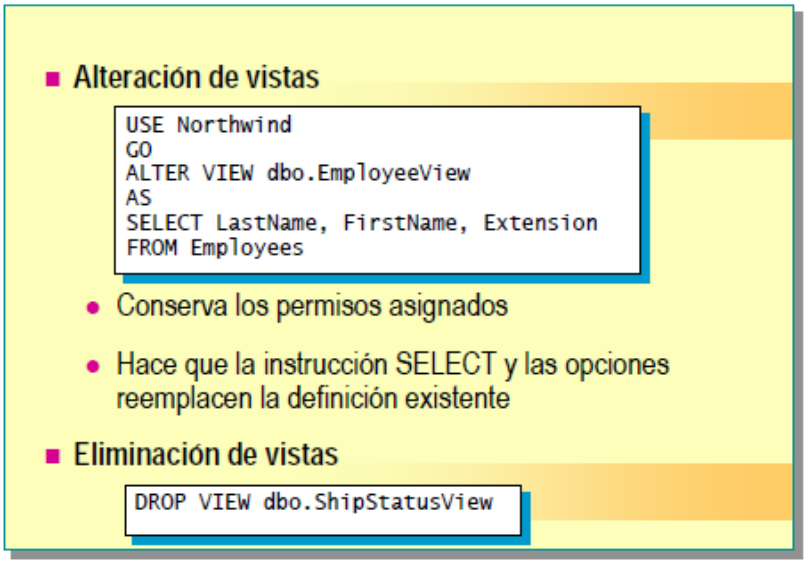
Modificación y eliminación de vistas

A menudo, las vistas se alteran como respuesta a las peticiones de información adicional por parte de los usuarios o a causa de cambios en la definición de las tablas subyacentes. Para alterar una vista puede quitarla y volverla a crear, o bien puede ejecutar la instrucción **ALTER VIEW.**

Eliminación de vistas

Si ya no necesita una vista, puede quitar su definición de la base de datos con la instrucción DROP VIEW. Al quitar una vista se quita su definición y todos los permisos que tenga asignados. Además, si los usuarios consultan vistas que hagan referencia a la vista quitada, obtendrán un mensaje de error. Sin embargo, al quitar una tabla que hace referencia a una vista, ésta no se quita automáticamente. Es necesario quitarla de forma explícita. Se utiliza la instrucción DROP VIEW.

Ejemplo.



Ejemplo

Agregar un nuevo campo (CompanyName de la tabla Customers) a la consulta ShipStatusView

```
ALTER VIEW dbo.ShipStatusView
AS
      SELECT OrderID, ShippedDate, ContactName, CompanyName
      FROM Customers c INNER JOIN Orders o
      ON c.CustomerID=o.CustomerID
      WHERE RequiredDate<ShippedDate
GO
```

Resultado

	OrderID	ShippedDate	ContactName	CompanyName
1	10280	1996-09-12 00:00:00.000	Christina Berglund	Berglunds snabbköp
2	10924	1998-04-08 00:00:00.000	Christina Berglund	Berglunds snabbköp
3	10970	1998-04-24 00:00:00.000	Martín Sommer	Bólido Comidas preparadas
4	10827	1998-02-06 00:00:00.000	Laurence Lebihan	Bon app'
5	10663	1997-10-03 00:00:00.000	Laurence Lebihan	Bon app'
6	10578	1997-07-25 00:00:00.000	Victoria Ashworth	B's Beverages
7	10726	1997-12-05 00:00:00.000	Ann Devon	Eastern Connection

Eliminar la vista ShipStatusView

```
DROP VIEW ShipStatusView
```

Programación con Transact-SQL

Transact-SQL no es realmente un lenguaje de programación similar a las herramientas de tercera y cuarta generación sin embargo permite utilizar SQL para realizar tareas complejas que requieren saltos, bucles, decisiones. Transact-SQL se utiliza a menudo en la creación de procedimientos almacenados y triggers de tal forma que las aplicaciones clientes que se conectan a SQL Server solo se preocupan por la presentación de los datos para el usuario final, mientras que la lógica de los procesos se maneja en el servidor.

¿Qué es un procedimiento almacenado?

Un procedimiento almacenado es un grupo de instrucciones T-SQL que se guardan juntas en la base de datos como una única unidad de código gracias a ello se puede pasar información en tiempo de ejecución mediante parámetros y recuperar información como un conjunto de resultados o parámetros de salida.

Un procedimiento almacenado se compila la primera vez que se ejecuta.

Variables:
Las variables locales se identifican como aquellos objetos que comienzan con el carácter arroba '@' una vez; las variables globales se identifican como los objetos que tienen 2 arrobas al inicio '@@', como ejemplo de variables globales tenemos: @@rowcount, @@error.

Las variables locales se declaran al inicio de un proceso por lotes o un procedimiento almacenado, la forma de asignarle valores a una variable es con la instrucción SELECT.

El control de flujo en Transact-SQL

Construcción	Descripción
IF..ELSE	Define una decisión.
GOTO etiqueta	Define un salto incondicional
WAITFOR	Establece un tiempo para la ejecución de una instrucción. El tiempo puede ser un intervalo de retardo o un instante especificado de ejecución (una hora concreta del dia)
WHILE	Bucle básico de SQL
BREAK	Acompaña al bucle WHILE y le indica finalizarlo inmediatamente.
CONTINUE	Acompaña al bucle WHILE y le indica continuar con la siguiente iteración.
RETURN [n]	Salida incondicional del procedimiento o proceso por lotes, se puede definir un número entero como estado devuelto y puede asignarse a cualquier variable.
BEGIN..END	Utilizado en conjunto con IF..ELSE o WHILE para agrupar un conjunto de instrucciones.
CASE	Implementada en la instrucción SELECT y UPDATE y permite realizar consultas y actualizaciones condicionales.

PRINT

Es una instrucción para imprimir un dato en la pantalla, la sintaxis es:

PRINT "cadena"; cadena puede ser también una variable de tipo varchar.

Por ejemplo: PRINT ‘Hola a todos’

RAISERROR

Es similar a PRINT, pero permite especificar un número de error y la severidad del mensaje. RAISERROR también permite que los errores se registren en el servicio de sucesos de Windows NT haciendo posible leerlos a través del visor de sucesos de Windows NT.

La sintaxis es:
RAISERROR({id_mensaje | cadena_mensaje}, severidad, estado [, argumento1 [,argumento2]]) WITH LOG.

Después de llamar a RAISERROR, la variable global @@ERROR tendrá el valor de id_mensaje, si no se pasa ningún id_mensaje, asumirá 5000.

Operadores

Los operadores aritméticos permitidos son:
+, -, *, /, %; se utilizan de forma similar a otros lenguajes de programación; los operadores binarios válidos son: & (AND), | (OR), ^ (OR exclusivo), ~ (NOT binario) y se utilizan solo en datos de tipo int o tinying.

Los operadores de comparación son:
= (igual), > (mayor que), < (menor que), >= (mayor o igual que), <= (menor o igual que), <> (distinto)

Creación de procedimientos almacenados (Store Procedures)

La instrucción para crear procedimientos almacenados es la siguiente:

Sintaxis básica para crear un procedimiento almacenado.

```
CREATE PROCEDURE  nombre_procedimiento
 {
@parametro1  tipo_dato ,
@parametro2  tipo_dato ,
@parametro3  tipo_dato , [OUTPUT]
…
@parametroN  tipo_dato
}
AS
Instrucción_SQL1 ,
Instrucción_SQL2 ,
Instruccion_SQL3,
…
Instrucción_SQLN
```

Es necesario aclarar, que un procedimiento almacenado puede recibir parámetros de entrada y devolver parámetros de salida.

Argumentos:

- **nombre_procedimiento.**

Es el nombre del nuevo procedimiento almacenado. Los nombres de procedimiento deben seguir las reglas de los identificadores y deben ser únicos en la base de datos y para su propietario.

- **@parametro.**
  Es un parámetro del procedimiento. En una instrucción Create Procedure, se pueden declarar uno o más parámetros. El usuario debe proporcionar el valor de cada Parámetro declarado cuando se ejecuta el procedimiento, a menos que se haya definido un valor predeterminado para el parámetro. Un procedimiento almacenado puede tener un máximo de 2.100 parámetros. Especificar el nombre de parámetro con un signo (@) como el primer carácter.

- **tipo_dato.**
  Es el tipo de datos del parámetro.

- **OUTPUT**
  Indica que se trata de un parámetro de retorno. Utilice los parámetros OUTPUT para devolver información al procedimiento que llama. Un parámetro de salida que utilice la palabra clave OUTPUT puede ser un marcador de posición de cursor.

- **AS**
  Son las acciones que va a llevar a cabo el procedimiento.

- **instruccion_sql**
  Es cualquier número y tipo de instrucciones Transact-SQL que se incluirán en el procedimiento.

Reglas de procedimientos almacenados

Entre las reglas para la programación de procedimientos almacenados, cabe citar las siguientes:
La propia definición CREATE PROCEDURE puede incluir cualquier número y tipo de instrucciones SQL, excepto las siguientes instrucciones CREATE, que no pueden ser utilizadas nunca dentro de un procedimiento almacenado:

CREATE DEFAULT	CREATE TRIGGER
CREATE PROCEDURE	CREATE VIEW
CREATE RULE	

Se puede crear otros objetos de base de datos dentro de un procedimiento almacenado. Puede hacer referencia a un objeto creado en el mismo procedimiento almacenado, siempre que se cree antes de que se haga referencia al objeto.
Puede hacer referencia a tablas temporales dentro de un procedimiento almacenado.
Si ejecuta un procedimiento almacenado que llama a otro procedimiento almacenado, el procedimiento al que se llama puede tener acceso a todos los objetos creados por el primer procedimiento, incluidas las tablas temporales.
Si se ejecuta un procedimiento almacenado remoto que realiza cambios en un servidor, los cambios no se podrán deshacer. Los procedimientos almacenados remotos no intervienen en las transacciones.
El número máximo de parámetros en un procedimiento almacenado es de 1,024.
El número máximo de variables locales en un procedimiento almacenado está limitado únicamente por la memoria disponible.

Ventajas de los procedimientos almacenados

- Compartir la lógica de la aplicación
- Permite una programación modular

- Protege detalles del esquema de base de datos
- Puede utilizarse como mecanismo de seguridad
- Permite una ejecución más rápida
- Puede reducir el tráfico de red

Normas para la creación de procedimientos almacenados

- El usuario dbo debe ser el propietario de todos los procedimientos almacenados
- Un procedimiento almacenado para una tarea
- Crear, probar y depurarlo en el servidor
- Evitar usar el prefijo sp_ en el nombre de los procedimientos almacenados
- Usar la misma configuración de conexión para todos los procedimientos almacenados
- Minimizar el uso de los procedimientos almacenados temporales

Usando parámetros de entrada

- Capturar valores de parámetros inválidos
- Usar valores predeterminados apropiados

Ejemplos:

Ejemplo 1

1. Crear una base de datos con el nombre: Clase11_NumCarnet
2. Crear las siguientes tablas:

   Tabla: Empleado
   Campos
   - Numero_Emp
   - Nombre_Emp
   - Apellido_Emp
   - Sexo_Emp
   - Salario_Emp
   - Numero_Dep

   Tabla: Departamento
   Campos:
   - Numero_Dep
   - Nombre_Dep
3. Crear las relaciones
4. Agregar datos a las tablas
5. Crear el siguiente procedimiento almacenado

```
CREATE PROCEDURE ListaEmpleados
@departamento varchar(15) = 'Investigación',
```

```
@sexo varchar(1)= 'F',
@salario money= 500
AS
SELECT Nombre_Emp, ApellidoPat_Emp, Sexo_Emp, Salario_Emp, Nombre_Dep
FROM Empleado E
INNER JOIN Departamento D ON E.Numero_Dep= D.Numero_Dep
WHERE Salario_Emp>= @salario AND sexo_Emp= @sexo AND
Nombre_Dep= @departamento
```

Ejecutando procedimientos almacenados con parámetros de entrada

• Pasando valores por referencia

```
EXEC ListaEmpleados
@salario= 1000,
@departamento= 'Administración'--dato almacenado en la tabla,
@sexo= 'M'--dato almacenado en la tabla
```

Pasando valores por posición

```
EXEC ListaEmpleados 'Administración', 'M', 1000
```

Retornando valores con parámetros de salida

Creando el procedimientos almacenado

```
CREATE PROCEDURE Depto
@NombreDep varchar (15),
@NumEmp int OUTPUT
AS
SELECT@NumEmp= COUNT(*) FROM Empleado E INNER JOIN Departamento D
ON E.Numero_Dep= D.Numero_Dep WHERE NombreD= @NombreDep
GO

--Imprime el total de empleados encontrados en la consulta
DECLARE @total int
EXEC Depto 'Administración', @total OUTPUT
SELECT  'El resultado es:', @total
```

@@ROWCOUNT

Devuelve el número de filas afectadas por la última instrucción.

Las instrucciones Transact-SQL pueden establecer el valor de @@ROWCOUNT de las siguientes maneras:

- Establecer @@ROWCOUNT en el número de filas afectadas o leídas. Las filas pueden o no enviarse al cliente.
- Conservar @@ROWCOUNT de la anterior ejecución de una instrucción.
- Restablecer @@ROWCOUNT en 0 y no devolver el valor al cliente.

Ejemplo 2

Crear las siguientes tablas:

Tabla: Producto

Campos:
- idproducto
- nombreproducto

Tabla: Proveedor

Campos:
- idproveedor
- nombreproveedor

Tabla: Catalogo

Campos:
- idprodcat
- idproducto
- idproveedor
- precio

Tabla: Bodega

Campos:
- idprodcat
- fecha
- preciocompra
- precioventa
- unidades

Para insertar datos en la tabla bodega, se creara un procedimiento almacenado que toma 5 parámetros de entrada los cuales son:

@fecha de tipo datetime, @proveedor,@producto de tipo char, @ganancia de tipo decimal y @unidades de tipo entero(int).

Ejecute el siguiente query para crear el procedimiento almacenado “producto_bodega”:

```
-- PROCEDIMIENTO ALMACENADO QUE INSERTA PRODUCTOS EN BODEGA
```

```
CREATE PROCEDURE PRODUCTO_BODEGA
  @fecha datetime,
  @proveedor char(8),
  @producto char(8),
  @ganancia decimal(2,2),
  @unidades int

AS

  DECLARE @idprodcat char(16)
  DECLARE @precio decimal(8,2)

  SELECT @idprodcat = idprodcat FROM catalogo WHERE
  idproducto=@producto AND idproveedor=@proveedor

   IF @@ROWCOUNT=0 GOTO error

  SELECT @precio = precio FROM catalogo WHERE idprodcat = @idprodcat

  INSERT INTO bodega VALUES
  (@idprodcat,@fecha,@precio,@precio+(@precio*@ganancia),@unidades)

  RETURN (0)   --Retorna 0, indica una ejecución exitosa.

error:
  PRINT 'no existe un producto en el catalogo'
  RETURN (1) –Objeto no encontrado
GO
```

Algunas características del procedimiento almacenado creado anteriormente son:

a) La línea **declare** @idprodcat char(16) esta declarando una variable local utilizada dentro del procedimiento almacenado. No confundir una variable local con un parámetro de entrada o de salida.
b) Para asignarle un valor a una variable se utiliza la sentencia **SELECT** seguido de la consulta o valor que será almacenado en dicha variable. Por ejemplo:

   - SELECT @X = 10 (aquí se le asigna a la variable X el valor de 10).
   - SELECT @precio = precio from catalogo where idprodcat = @idprodcat
     (Aquí se le está asignando a la variable precio el dato que retorna la consulta hecha a la tabla catalogo. En este caso el query retorna el campo precio de dicha tabla).

c) La línea **if @@rowcount** = 0 indica que si el ultimo query realizado no devolvió ningún registro (@@rowcount=0) entonces se hace un salto incondicional a la etiqueta error.

d) Si hubo error la sentencia **print** muestra el mensaje 'No existe un producto en el catalogo', de lo contrario se insertan los datos.

Ejemplo 3

```
USE northwind
GO
CREATE PROC DevuelveClientes
```

```
@cond nvarchar(10),
@cuantos int output
AS
SET nocount ON
SELECT *  FROM Customers WHERE customerid LIKE @cond
SET @cuantos=@@rowcount

GO

--Ejecutando el procedimiento

DECLARE @var_cuantos int
EXEC DevuelveClientes '%A%',@var_cuantos OUTPUT
SELECT 'Hay ' , @var_cuantos
```

Donde:

SET NOCOUNT

Evita que se devuelva el mensaje que muestra el recuento del número de filas afectadas por una instrucción o un procedimiento almacenado de Transact-SQL como parte del conjunto de resultados.
Si se establece SET NOCOUNT en ON, no se devuelve el recuento. Cuando SET NOCOUNT es OFF, sí se devuelve ese número.

Ejemplo 4

Ahora realizaremos un procedimiento almacenado que hace referencia a la tabla **Pedido de la base de datos Bodega** de tal forma que si insertamos la venta de un producto, primero se debe verificar que haya existencias de dicho producto en nuestra base de datos; si hay existencia suficientes se permitirá la acción y se actualizarán las existencias en la tabla producto para que exista consistencia entre los datos (si vendo productos, disminuye mi existencias).

El procedimiento almacenado sería el siguiente:

```
CREATE PROC actualizar_existencia
@idproducto int,
@unidades int
AS
  DECLARE @existencia int

  SELECT @existencia = existencias
  FROM producto where idprod = @idproducto

  BEGIN TRAN

  IF (@existencia<@unidades)
 GOTO errores

  UPDATE PRODUCTO
  SET existencias = existencias - @unidades
  WHERE idprod = @idproducto
```

```
  COMMIT TRAN
  GOTO FIN

  errores:
  ROLLBACK TRAN --se elimina la transaccion

  fin:
    PRINT 'Existencias actualizada'
    return 0
```

Donde:
BEGIN TRANSACTION

Marca el punto de inicio de una transacción local explícita.

COMMIT TRANSACTION

Marca el final de una transacción correcta, implícita o explícita

ROLLBACK TRANSACTION

Borra todas las modificaciones de datos realizadas desde el inicio de la transacción o hasta un punto de almacenamiento. También libera los recursos que retiene la transacción.

@@ERROR

- Devuelve el número de error de la última instrucción Transact-SQL ejecutada.
- Devuelve 0 si la instrucción Transact-SQL anterior no encontró errores.
- Devuelve un número de error si la instrucción anterior encontró un error. Si el error era uno de los errores de la vista de catálogo sys.messages, entonces @@ERROR contendrá el valor de la columna sys.messages.message_id column para dicho error.

Ejemplo utilizando @@error

```
-- Si existe la tabla la borramos.
IF NOT OBJECTPROPERTY(OBJECT_ID('ejemplo'),'IsTable') IS NULL
   DROP TABLE ejemplo
GO
-- Creamos la tabla con los elementos necesarios
-- para nuestro propósito ( los dos últimos campos).

CREATE TABLE ejemplo
                id int not null identity(1,1)  PRIMARY KEY,
                Nombre varchar(50) not null,
                Telefono varchar(9) null,
                Conexion int null,
                FechaHora datetime default GETDATE()
GO
```

```
-- Crearemos el procedimiento almacenado que sirva para almacenar
-- los datos en la tabla cliente (primero si existe lo borramos).

IF NOT OBJECTPROPERTY(OBJECT_ID('InsertaClientes'),'Isprocedure') IS NULL
   DROP PROCEDURE Insertaclientes
GO
CREATE PROC Insert_ejemplo
@Nombre varchar(50),
@Telefono varchar(10),
@id int OUTPUT

AS
   BEGIN TRAN

      INSERT INTO ejemplo (Nombre,Telefono) VALUES
         (@Nombre,@Telefono)

         IF @@ERROR<>0 GOTO Deshacer
         -- Devolvemos el código autonumérico del id
         SET @id = SCOPE_IDENTITY()

COMMIT TRAN
 GOTO fin
Deshacer:
   ROLLBACK TRAN
  fin:
  PRINT 'Registro Agregado'
GO
-- Comprobar que funciona....

DECLARE @id int
EXEC Insert_ejemplo 'Miguel Sigaran', '2226-7895',@id OUTPUT
PRINT @id
EXEC Insert_ejemplo 'Maria Carmen Gil ',' 7895-2545',@id OUTPUT
PRINT @id
GO
```

TRIGGERS

Tablas inserted y deleted

En muchas ocasiones las operaciones de actualización de la tabla que suponen inserción y borrado de filas se llevan a cabo en cadena, normalmente por la propia naturaleza de las sentencias implicadas.

En este tipo de situaciones el trigger que deba responder a estas acciones puede necesitar valorar qué cambios se han producido sobre la tabla de manera que se pueda obrar en consecuencia. Para ello se necesitaría disponer de algún modo de información del estado de la tabla antes y después de las modificaciones efectuadas. Para ello SQL Server proporciona dos tablas temporales denominadas: **INSERTED y DELETED**

- Tablas temporales residentes en memoria
- La tabla deleted almacena copias de las filas afectadas por las instrucciones DELETE y UPDATE
- La tabla inserted almacena copias de las filas afectadas durante las instrucciones INSERT y UPDATE

Consideraciones de rendimiento:

- Los desencadenadores o triggers trabajan rápidamente porque las tablas inserted y deleted están en caché
- El tiempo de ejecución está determinado por:
  - ✓ El número de tablas que son referenciadas
  - ✓ El número de filas que son afectadas

EJEMPLO. Uso de la tabla inserted

```
CREATE TRIGGER Agregar_Autor

ON autores

FOR INSERT

AS

-- Seleccionando el nombre y apellido del Nuevo autor

DECLARE @nuevo_Nombre VARCHAR(100)

SELECT @nuevo_Nombre = (SELECT Nombre + '  ' + Apellido FROM Inserted)
-- Imprimiendo el nombre del Nuevo actor

PRINT 'Nuevo Autor"' + @ nuevo_Nombre + '" fue agregado.'
```

EJEMPLO. Uso de la tabla deleted

```
CREATE TRIGGER Eliminar _Autor

ON autores

FOR DELETE

AS

-- Seleccionando el nombre y apellido del Nuevo autor

DECLARE @elim_Nombre VARCHAR(100)

SELECT @elim_Nombre = (SELECT Nombre + '  ' + Apellido FROM Inserted)
-- Imprimiendo el nombre del Nuevo actor

PRINT 'El  Autor"' + @ eli_Nombre + '" fue eliminado de la tabla.'
```

Otro ejemplo. Uso de la tabla deleted

```
CREATE TRIGGER empl_tm
ON empl
FOR UPDATE
AS
INSERT emp_historico
SELECT emp_id, dept_id, puesto,salario
FROM deleted

--Ejemplo
--Dato agregado
insert into empl values(1,'Carlos', 'Miranda',1,'Jefe',125)

--Dato agregado cuando se activa el trigger
update empl set salario=135.50 where emp_id=1
```

Forzando la integridad de datos

```
CREATE TRIGGER elimina_reserva

ON prestamo FOR INSERT

AS

IF (SELECT r.codigo FROM reservación r INNER JOIN inserted i ON r.codigo=i.codigo

AND r.isbn= i.isbn) >0

BEGIN

DELETE reservación FROM reservación r INNER JOIN inserted i ON r.codigo=i.codigo

AND r.isbn= i.isbn

END
```

Préstamo

isbn	copia	codigo
1	1	1
4	1	7
4	2	4
3	1	1

Fila Insertada

Fila Eliminada

Reservación

isbn	codigo
1	1
1	2
2	1
4	7

Propiedades ACID

ACID, conformado por las siglas provenientes de Atomicity, Consistency, Isolation y Durability. En español, Atomicidad, Consistencia, Aislamiento y Durabilidad, son un conjunto de propiedades necesarias para que un conjunto de instrucciones, sean consideradas como una transacción en un sistema de gestión de bases de datos.
Una transacción es un conjunto de órdenes que se ejecutan formando una unidad de trabajo, es decir, en forma indivisible o atómica. Un ejemplo de una transacción compleja es la transferencia de fondos de una cuenta a otra, la cual implica múltiples operaciones individuales.

Si un sistema supera la prueba ACID, significa que es fiable.

Atomicidad: Significa que el sistema permite operaciones atómicas. Una operación atómica es aquella que si está formada por operaciones más pequeñas, se consideran como un paquete indivisible. Deben ejecutarse todas correctamente, o en el caso de que alguna de ellas no pueda hacerlo, el efecto de las que ya se han ejecutado no debe hacerse notar, debe deshacerse, como si el conjunto de las operaciones no se hubieran realizado.
La atomicidad está íntimamente ligada al concepto de transacción de los sistemas gestores de bases de datos. En un SGBD, cuando se indica que un conjunto de operaciones forman una transacción, o se ejecutan todas correctamente, o el SGBD deshará los cambios, como si la transacción nunca se hubiera iniciado. No obstante, atomicidad y transacción no son sinónimas. Mientras atomicidad es una propiedad, la transacción es el mecanismo que utilizan los SGBD para lograr la atomicidad.

Consistencia: Integridad. Esta propiedad asegura que sólo se empieza aquello que se puede acabar. Por lo tanto se ejecutan aquellas operaciones que no van a romper las reglas y directrices de integridad de la base de datos. Sostiene que cualquier transacción llevará a la base de datos desde un estado válido a otro también válido.

Aislamiento: Propiedad que asegura que una operación no puede afectar a otras. Esto asegura que la realización de dos transacciones sobre la misma información sean independientes y no generen ningún tipo de error.

Durabilidad: Propiedad que asegura que una vez realizada la operación, ésta persistirá y no se podrá deshacer aunque falle el sistema.