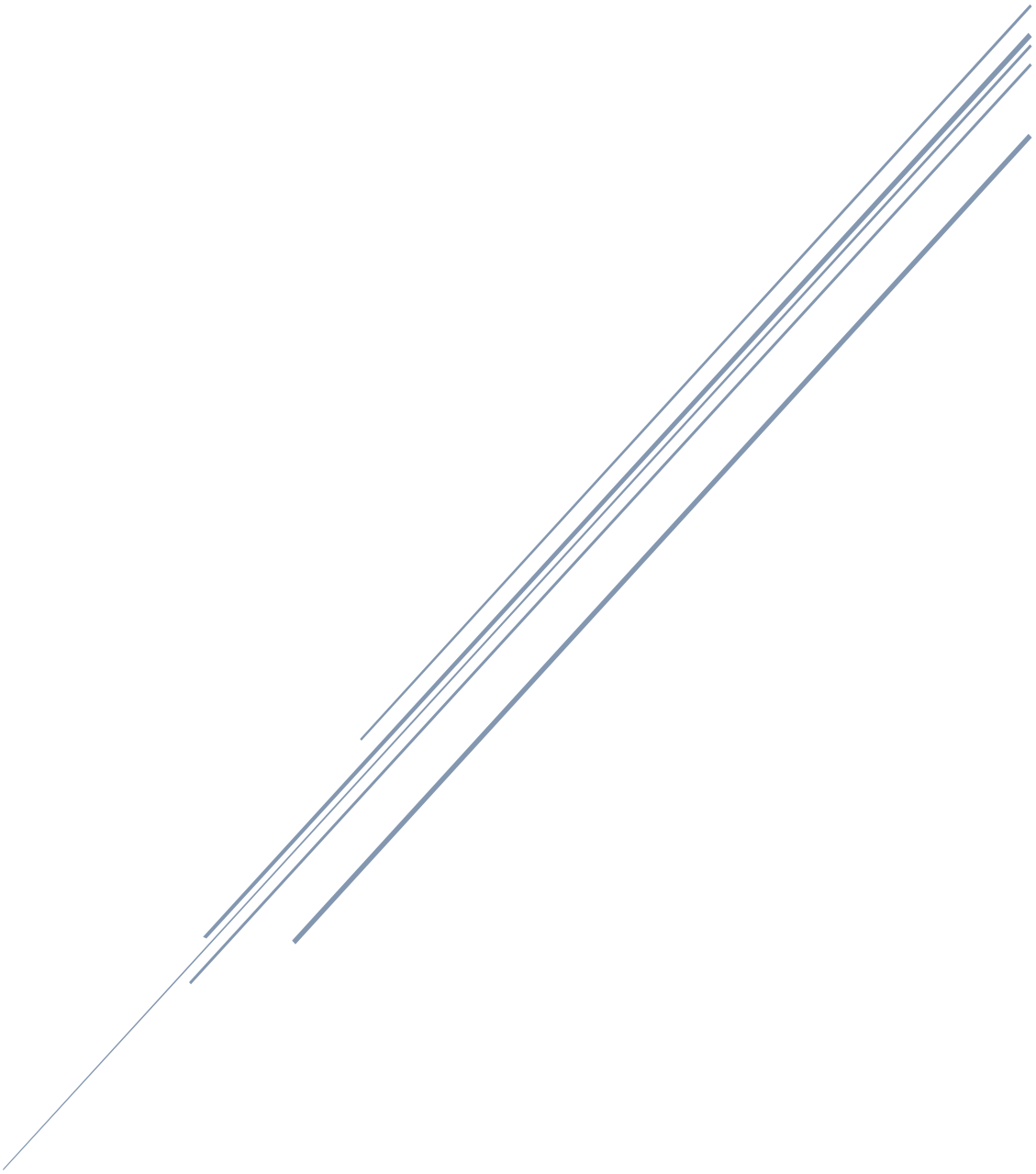
PROGRAMACIÓN DE BASES DE DATOS



Bases de Datos

Contenido

Vistas en SQL

Una vista es una tabla virtual que muestra datos de una o varias tablas, pero sin duplicarlos ni almacenarlos nuevamente.
Funciona como una consulta guardada, permitiendo acceder a los datos sin escribir la consulta completa cada vez.
Imagina que en un hospital hay una base de datos con mucha información sobre pacientes, médicos y tratamientos. Sin embargo, los recepcionistas solo necesitan ver los nombres de los pacientes y las citas, mientras que los médicos necesitan ver el historial completo. Aquí es donde las vistas en SQL nos ayudan.
Las vistas solo contienen datos en tiempo de ejecución y se almacenan en caché. Lo que quiere decir que, ante varias consultas concurrentes, se accede a esa información almacenada en memoria sin necesidad de ejecutar la consulta contra las tablas del servidor una y otra vez.

¿Para qué se usan las vistas?

- Filtrar información: Mostrar solo los datos necesarios, ocultando los que no queremos que todos vean.
- Hacer más fácil el acceso a los datos: En lugar de escribir una consulta compleja cada vez, usamos la vista como si fuera una tabla.
- Mejorar la seguridad: Podemos dar acceso a una vista en lugar de a la tabla completa, restringiendo ciertos datos.
- Unificar datos de varias tablas: Podemos combinar datos de distintas tablas en una sola vista.

Ejemplo de una vista en SQL

Supongamos que tenemos una base de datos de un hospital con una tabla PACIENTES y una tabla CITAS.

```
CREATE TABLE PACIENTES (
    ID_PACIENTE INT PRIMARY KEY IDENTITY,
    NOMBRE VARCHAR(50) NOT NULL,
    APELLIDO VARCHAR(50) NOT NULL,
    FECHA_NACIMIENTO DATE NOT NULL,
```

```
    TELEFONO VARCHAR(15),
    DIRECCION VARCHAR(100),
    SEGURO_MEDICO VARCHAR(50)
);
CREATE TABLE CITAS (
    ID_CITA INT PRIMARY KEY IDENTITY,
    ID_PACIENTE INT NOT NULL,
    FECHA_CITA DATETIME NOT NULL,
    MEDICO VARCHAR(50) NOT NULL,
    MOTIVO VARCHAR(100),
    FOREIGN KEY (ID_PACIENTE) REFERENCES
PACIENTES(ID_PACIENTE)
);
```

Creando una vista

Queremos que la recepción del hospital solo vea el nombre, apellido y la fecha de la próxima cita de cada paciente.

```
CREATE VIEW Vista_Recepcion AS
SELECT P.NOMBRE, P.APELLIDO, C.FECHA_CITA, C.MOTIVO
FROM PACIENTES P
INNER JOIN CITAS C ON P.ID_PACIENTE = C.ID_PACIENTE;
```

Ahora, en vez de escribir la consulta completa cada vez, la recepción puede consultar los datos simplemente así:

```
SELECT * FROM Vista_Recepcion;
```

Beneficios de usar una vista

- Más fácil de usar: No es necesario recordar consultas largas y complicadas.

  Más seguro: No damos acceso directo a los datos sensibles de los pacientes, solo a la información necesaria.
- Más organizado: Podemos crear diferentes vistas según las necesidades de cada área del hospital.

Vistas con condiciones sobre campos

Enlazar varias tablas y seleccionar algunos de sus campos no es la única forma de hacer vistas. En ocasiones, necesitamos establecer determinados criterios que tienen que cumplir los datos que nos devuelve la vista. Es posible crear vistas con condiciones de campo igual que haríamos en una consulta SQL al uso Por ejemplo, queremos una vista que muestre solo los pacientes con seguro médico:

```
CREATE VIEW Vista_Pacientes_Asegurados AS
SELECT NOMBRE, APELLIDO, SEGURO_MEDICO
FROM PACIENTES
WHERE SEGURO_MEDICO IS NOT NULL;
```

Transacciones en SQL

En SQL, una transacción es un conjunto de instrucciones que deben ejecutarse todas juntas o ninguna.
Imagina que estás en una tienda y compras varios productos. Pagas todo junto y, si hay un problema con el pago, cancelas toda la compra.

- ✓ En las transacciones, si todo sale bien, se confirman los cambios con COMMIT.
- ✓ Si hay un problema, se cancela todo con ROLLBACK.

Para qué se pueden usar en la vida real

Transferir dinero de una cuenta a otra en un banco.
Registrar la compra de un cliente y actualizar el stock de productos.
Insertar datos en varias tablas relacionadas.

Funcionamiento de una transacción

La secuencia de ejecución que sigue una transacción es la que se enumera a continuación.

- Comienza con BEGIN TRANSACTION.
- Ejecuta las operaciones (INSERT, UPDATE, DELETE)

- Si todo va bien, usa COMMIT para guardar los cambios.
- Si hay un error, usa ROLLBACK para deshacer todo.

Ejemplo de transacción

Supongamos que tenemos dos cuentas bancarias (la cuenta 1 y la cuenta 2), en una tabla llamada CUENTAS y queremos transferir 100 euros de la cuenta 1 a la cuenta 2. El código de la transacción que llevaría a cabo esa operación es el siguiente:

```
BEGIN TRANSACTION;
UPDATE CUENTAS SET SALDO = SALDO - 100 WHERE ID_CUENTA = 1;
-- Restar dinero a la cuenta 1
UPDATE CUENTAS SET SALDO = SALDO + 100 WHERE ID_CUENTA = 2;
-- Sumar dinero a la cuenta 2
COMMIT; -- Guardamos los cambios
```

Si todo se ejecuta bien, se confirman los cambios con COMMIT. Pero sii hay un problema (por ejemplo, la cuenta 1 no tiene suficiente saldo), los datos pueden quedar inconsistentes.

Capturando errores con TRY...CATCH y ROLLBACK

Para asegurarnos de que no haya problemas, usamos TRY...CATCH para detectar errores.

```
BEGIN TRANSACTION;
BEGIN TRY
    UPDATE CUENTAS SET SALDO = SALDO - 100 WHERE ID_CUENTA
= 1;
    UPDATE CUENTAS SET SALDO = SALDO + 100 WHERE ID_CUENTA
= 2;

    COMMIT; -- Si todo va bien, confirmamos los cambios
    PRINT 'Transacción completada con éxito.';
END TRY
```

```
BEGIN CATCH
    ROLLBACK; -- Si hay un error, cancelamos todo
    PRINT 'Error en la transacción. Se han deshecho los cambios.';
END CATCH;
```

¿Qué es lo que hace el código anterior?

Si todo sale bien, COMMIT guarda los cambios.

Si hay un error, ROLLBACK deshace todo y evita inconsistencias.

¿Dónde se guardan las transacciones en SQL Server?

En memoria: Mientras una transacción está en curso (BEGIN TRANSACTION), los cambios aún no se han guardado de forma permanente en la base de datos.

En el log de transacciones: SQL Server usa un archivo llamado Transaction Log (.ldf) para registrar todos los cambios antes de aplicarlos a la base de datos.

Esto permite recuperar datos en caso de fallos y hacer rollback si es necesario.

En el archivo de base de datos: Cuando se ejecuta COMMIT, los cambios se aplican definitivamente y se escriben en el archivo de base de datos (.mdf o .ndf).

Si se hace ROLLBACK, los cambios se eliminan y la base de datos vuelve al estado anterior a la transacción.

¿Qué son los archivos .mdf **y** .ndf **en SQL Server?**

En SQL Server, la información de una base de datos se almacena en archivos físicos en el disco. Los archivos principales son:

- Archivo .mdf (Primary Data File - Archivo de Datos Principal)

  Es el archivo principal de la base de datos que contiene todos los datos de las tablas, vistas, procedimientos almacenados, índices, etc.

  Siempre hay un único archivo .mdf en cada base de datos.

  Ejemplo:

  Cuando creas una base de datos en SQL Server, se genera automáticamente un archivo .mdf.

  ```
  CREATE DATABASE MiHospital;
  ```

  Esto crea un archivo llamado MiHospital.mdf en el disco, donde se almacenan todas las tablas y datos.

- Archivo .ndf (Secondary Data File - Archivo de Datos Secundario)

Es opcional: No todas las bases de datos lo necesitan.

Se usa cuando el archivo .mdf se está volviendo muy grande y necesitas distribuir los datos en varias unidades de almacenamiento. Puedes tener múltiples archivos .ndf en una base de datos.

Ejemplo:

Si necesitas dividir los datos en varios discos, puedes agregar un .ndf así:

```
ALTER DATABASE MiHospital
ADD FILE (
    NAME = MiHospital_Extra,
    FILENAME = 'D:\SQLData\MiHospital_Extra.ndf',
    SIZE = 5MB,
    MAXSIZE = 50MB,
    FILEGROWTH = 5MB
);
```

Ahora SQL Server distribuirá los datos entre MiHospital.mdf y MiHospital_Extra.ndf.

- Archivo .ldf (Transaction Log File - Archivo de Registro de Transacciones)

No es un archivo de datos, sino un log de transacciones. Guarda un historial de todas las transacciones (INSERT, UPDATE, DELETE) antes de confirmarlas.

Sirve para recuperar la base de datos en caso de fallo.

Cuando creas MiHospital.mdf, también se genera automáticamente un archivo MiHospital.ldf.

```
CREATE DATABASE MiHospital;
```

La sentencia anterior, crea:

MiHospital.mdf (datos)

MiHospital.ldf (registro de transacciones)

En conclusión

Tipo de Archivo	Descripción	Obligatorio
.mdf	Archivo principal que contiene todos los datos	Sí
.ndf	Archivo secundario opcional para distribuir datos	No

Tipo de Archivo	Descripción	Obligatorio
.ldf	Archivo de log de transacciones, usado para recuperación	Sí

Procedimientos Almacenados en SQL Server

Un Procedimiento Almacenado (Stored Procedure) es un bloque de código SQL precompilado que se almacena en la base de datos y que se puede ejecutar cuando sea necesario.

Ventajas de los Procedimientos Almacenados:

Mejoran el rendimiento: Al estar precompilados, se ejecutan más rápido.

Son reutilizables: Pueden ser usados por varias aplicaciones o usuarios.

Ofrecen mayor seguridad: Se pueden restringir los permisos de ejecución sin exponer el código.

Facilitan el mantenimiento: Si necesitas hacer cambios en una operación sobre la base de datos, los haces desde un único lugar.

Reducen el tráfico en la red: Solo se envían los parámetros, no toda la consulta.

¿Para qué se usan los procedimientos almacenados?

Hay varias tareas en las que los procedimientos almacenados resultan de gran utilidad para los administradores de las bases de datos. Entre ellas están:

Insertar, actualizar, eliminar y consultar datos.

Aplicar lógica de negocio en la base de datos.

Automatizar tareas administrativas.

Reducir el tiempo de ejecución de consultas repetitivas.

¿Pueden recibir parámetros?

Sí, un procedimiento almacenado puede recibir parámetros de entrada para trabajar con datos dinámicos.

Ejemplo de un procedimiento con parámetros de entrada:

```
CREATE PROCEDURE ObtenerPedidosCliente
@ID_Cliente INT
```

```
AS
BEGIN
    SELECT * FROM PEDIDOS WHERE CLIE = @ID_Cliente;
END;
```

Cómo ejecutar el procedimiento anterior

```
EXEC ObtenerPedidosCliente 5;
```

Esto devuelve todos los pedidos del cliente con ID = 5.

¿Pueden devolver parámetros?

Sí, un procedimiento puede devolver valores de salida con OUTPUT o devolver un conjunto de resultados (SELECT).

Ejemplo de un procedimiento con parámetro de salida:

```
CREATE PROCEDURE ObtenerTotalPedidosCliente
    @ID_Cliente INT,
    @TotalPedidos INT OUTPUT
AS
BEGIN
    SELECT @TotalPedidos = COUNT(*) FROM PEDIDOS WHERE CLIE
= @ID_Cliente;
END;
```

Cómo ejecutar el procedimiento anterior.

```
DECLARE @Total INT;
EXEC ObtenerTotalPedidosCliente 5, @Total OUTPUT;
PRINT @Total;
```

Esto imprimirá la cantidad de pedidos del cliente 5.

¿Pueden invocar a otros procedimientos?

Sí, un procedimiento almacenado puede llamar a otro procedimiento dentro de su código.

Ejemplo:

```
CREATE PROCEDURE ObtenerPedidosYTotal @ID_Cliente INT
AS
BEGIN
    EXEC ObtenerPedidosCliente @ID_Cliente;
```

```
    DECLARE @Total INT;
    EXEC ObtenerTotalPedidosCliente @ID_Cliente, @Total OUTPUT;
    PRINT 'Total de pedidos: ' + CAST(@Total AS VARCHAR);
END;
```

Cómo lo ejecutamos:

```
EXEC ObtenerPedidosYTotal 5;
```

Llama a ObtenerPedidosCliente y éste, a su vez, a ObtenerTotalPedidosCliente.

¿Pueden invocar funciones?

Sí, un procedimiento almacenado puede llamar a funciones definidas en SQL Server, pero solo funciones escalares y funciones con valor de tabla (TVF).

Ejemplo de un procedimiento que usa una función:

En primer lugar, tenemos la función

```
CREATE FUNCTION ObtenerMontoTotalPedidos (@ID_Cliente INT)
RETURNS INT
AS
BEGIN
    DECLARE @Total INT;
    SELECT @Total = SUM(IMPORTE) FROM PEDIDOS WHERE CLIE = @ID_Cliente;
    RETURN @Total;
END;
sql
```

Y en segundo lugar el procedimiento que la invoca

```
CREATE PROCEDURE MostrarMontoTotalPedidos @ID_Cliente INT
AS
BEGIN
    DECLARE @Total INT;
    SET @Total = dbo.ObtenerMontoTotalPedidos(@ID_Cliente);
    PRINT 'El monto total de pedidos del cliente es: ' + CAST(@Total AS VARCHAR);
END;
```

Y para ejecutar el procedimiento

```
EXEC MostrarMontoTotalPedidos 5;
```

Que muestra el total de pedidos del cliente 5.

Funciones en SQL Server

Una función en SQL Server es un bloque de código SQL que devuelve un valor o un conjunto de valores y se puede reutilizar en consultas. A diferencia de los procedimientos almacenados, las funciones siempre devuelven un resultado.
Ventajas de las funciones en SQL Server:
Se pueden reutilizar en múltiples consultas.
Mejoran la organización del código SQL.
Permiten encapsular lógica de negocio en la base de datos.
Se pueden utilizar dentro de SELECT, WHERE, JOIN y otras cláusulas.

¿Para qué se usan las funciones?

Las funciones en SQL Server se usan para:

- Calcular valores personalizados a partir de datos en las tablas. Transformar y manipular datos.
- Realizar operaciones de agregación y cálculos sobre conjuntos de datos. Devolver valores escalares o tablas completas que pueden ser consultadas como una vista.

¿Pueden recibir parámetros?

Una función puede recibir uno o más parámetros de entrada.
Ejemplo de función con parámetros:

```
CREATE FUNCTION CalcularIVA (@precio DECIMAL(10,2))
RETURNS DECIMAL(10,2)
AS
BEGIN
    RETURN @precio * 0.21;
END;
```

Para invocar una función sql

```
SELECT ID_PRODUCTO, DESCRIPCION, PRECIO,
dbo.CalcularIVA(PRECIO) AS IVA
FROM PRODUCTOS;
```

Calcula el IVA del precio de cada producto.

¿Pueden devolver parámetros?

Sí, pero solo devuelven un valor escalar o una tabla completa.

Tipos de funciones según su valor de retorno:

- Funciones escalares: Devuelven un solo valor.
- Funciones con valor de tabla: Devuelven un conjunto de registros como una tabla.

Ejemplo de función con valor de tabla:

```
CREATE FUNCTION ProductosCaros()
RETURNS TABLE
AS
RETURN (
    SELECT ID_PRODUCTO, DESCRIPCION, PRECIO
    FROM PRODUCTOS
    WHERE PRECIO > 500
);
```

Uso:

```
SELECT * FROM dbo.ProductosCaros();
```

Devuelve todos los productos con precio mayor a 500.

¿Pueden invocar a otras funciones?

Sí, una función puede llamar a otras funciones dentro de su definición.

Ejemplo:

```
CREATE FUNCTION PrecioConIVA (@ID_Producto INT)
RETURNS DECIMAL(10,2)
AS
BEGIN
    DECLARE @precio DECIMAL(10,2);
```

```
    SELECT @precio = PRECIO FROM PRODUCTOS WHERE ID_PRODUCTO
= @ID_Producto;

    RETURN dbo.CalcularIVA(@precio) + @precio;
END;
```

Uso:

```
SELECT dbo.PrecioConIVA(1) AS Precio_Final;
```

Calcula el precio de un producto sumándole el IVA.

¿Pueden invocar procedimientos almacenados?

No, las funciones NO pueden llamar a procedimientos almacenados. SQL Server restringe esto porque las funciones deben ser deterministas y un procedimiento puede hacer cambios en los datos.

Si necesitamos ejecutar un procedimiento dentro de una función, se puede usar una vista en su lugar.

Triggers en Bases de Datos

Un trigger (disparador) es un objeto en la base de datos que se ejecuta automáticamente cuando ocurre un evento específico en una tabla o vista. No es necesario llamarlo manualmente, como ocurre con un procedimiento almacenado.

Se activan en respuesta a eventos como:

INSERT → Cuando se agrega un nuevo registro.

UPDATE → Cuando se actualiza un registro existente.

DELETE → Cuando se elimina un registro.

¿Para qué se usan los triggers?

Automatización de procesos → Ejemplo: Actualizar automáticamente otra tabla al modificar datos.

Control de seguridad → Evitar cambios no permitidos o registrar auditorías.

Mantenimiento de integridad referencial → Asegurar relaciones entre datos sin depender de reglas de claves foráneas.

Registro de cambios (auditoría) → Guardar un historial de modificaciones en la base de datos.

Sintaxis básica de un Trigger en SQL Server

```
CREATE TRIGGER nombre_trigger
ON nombre_tabla
AFTER INSERT, UPDATE, DELETE  -- Especifica el evento que lo activa
AS
BEGIN
    -- Lógica que se ejecutará cuando se active el trigger
END;
```

Tipos de triggers en SQL Server:

FOR o AFTER (por defecto) → Se ejecuta después de la acción (INSERT, UPDATE, DELETE).

INSTEAD OF → Sustituye la acción original, evitando que se realice.

Ejemplo Práctico 1:

Auditoría con un Trigger AFTER INSERT

Queremos registrar automáticamente cada vez que un nuevo paciente es agregado a la tabla PACIENTES.

Creamos una tabla para la auditoría:

```
CREATE TABLE AUDITORIA_PACIENTES (
    ID_AUDITORIA INT IDENTITY PRIMARY KEY,
    ID_PACIENTE INT,
    FECHA_INSERCION DATETIME DEFAULT GETDATE(),
    USUARIO VARCHAR(50)
);
```

Creamos el trigger para registrar las inserciones:

```
CREATE TRIGGER TRG_AUDITORIA_PACIENTES
ON PACIENTES
AFTER INSERT
AS
```

```
BEGIN
    INSERT INTO AUDITORIA_PACIENTES (ID_PACIENTE, USUARIO)
    SELECT ID_PACIENTE, SUSER_NAME() FROM inserted;
END;
```

¿Cómo funciona?

- Cada vez que un nuevo paciente se agrega a PACIENTES, el trigger se activa automáticamente.
- La tabla AUDITORIA_PACIENTES almacena el ID del paciente insertado y el usuario que realizó la acción.

Ejemplo Práctico 2:

Evitar que se eliminen registros con un Trigger INSTEAD OF DELETE

No queremos que se eliminen registros de la tabla PACIENTES, pero sí permitimos su actualización.

Creamos el trigger:

```
CREATE TRIGGER TRG_PREVENIR_BORRADO_PACIENTES
ON PACIENTES
INSTEAD OF DELETE
AS
BEGIN
    PRINT 'No se pueden eliminar pacientes. Solo se permite la actualización.'
END;
```

¿Cómo funciona?

- Si alguien intenta eliminar un paciente, el trigger intercepta la acción y evita que el registro se elimine.
- Se muestra un mensaje indicando que la eliminación está prohibida.

Ejemplo Práctico 3:

Actualizar automáticamente el stock de productos tras un pedido

Cada vez que se hace un pedido, se debe restar la cantidad del producto en la tabla PRODUCTOS.

Creamos el trigger:

```
CREATE TRIGGER TRG_ACTUALIZAR_STOCK
ON PEDIDOS
AFTER INSERT
AS
BEGIN
    UPDATE PRODUCTOS
    SET EXISTENCIAS = EXISTENCIAS - p.CANT
    FROM PRODUCTOS pr
    INNER JOIN inserted p ON pr.ID_FAB = p.FAB AND
pr.ID_PRODUCTO = p.PRODUCTO;
END;
```

¿Cómo funciona?

- Cuando se inserta un nuevo pedido, el trigger se activa.
- Busca el producto correspondiente y le resta la cantidad comprada en el pedido.

Conclusión

- Los triggers son una herramienta poderosa en SQL Server para automatizar tareas, mantener la seguridad e integridad de los datos.
- Pueden usarse para auditar cambios, prevenir eliminaciones y actualizar automáticamente registros.
- Sin embargo, hay que usarlos con cuidado, ya que pueden afectar el rendimiento si se ejecutan con demasiada frecuencia.