SEMANA 12 – SUBCONSULTAS

Subconsultas (Subqueries)

Una **subconsulta** o *subquery* es una consulta SQL anidada dentro de otra consulta (la consulta principal). Se utiliza para devolver datos que se usarán en la consulta principal.

---

1. `IN`

El operador `IN` se usa para determinar si un valor coincide con cualquier valor en una lista o en los resultados de una subconsulta.

Teoría	Código de Ejemplo

Selecciona datos donde una columna coincide con **cualquier valor** devuelto por la subconsulta.	sql
	-- Encontrar empleados que trabajan en departamentos en 'Ventas'
	SELECT *
	FROM Empleados
	WHERE DepartamentoID IN (
	    SELECT DepartamentoID
	    FROM Departamentos
	    WHERE NombreDepartamento = 'Ventas'
	);

2. `EXISTS`

El operador `EXISTS` se usa para probar la **existencia** de filas en el resultado de la subconsulta. Devuelve `TRUE` si la subconsulta devuelve una o más filas.

Teoría	Código de Ejemplo

Selecciona datos de la tabla principal **solo si la subconsulta devuelve al menos una fila** para cada fila de la tabla principal. Se usa a menudo para correlacionar datos entre tablas.	sql
	-- Encontrar clientes que han realizado al menos un pedido
	SELECT NombreCliente
	FROM Clientes c
	WHERE EXISTS (
	    SELECT 1
	    FROM Pedidos p
	    WHERE p.ClienteID = c.ClienteID
	);

Funciones de Fecha (Date Functions - SQL Server/T-SQL)

Las funciones de fecha se usan para manipular y extraer partes de valores de fecha y hora.

Función	Descripción	Código de Ejemplo
`GETDATE()`	Devuelve la fecha y hora actual del sistema.	`SELECT GETDATE();`
`DAY(fecha)`	Devuelve el día del mes (1-31) de una fecha específica.	`SELECT DAY('2025-11-17');` -- Resultado: 17
`MONTH(fecha)`	Devuelve el mes (1-12) de una fecha específica.	`SELECT MONTH('2025-11-17');` -- Resultado: 11
`YEAR(fecha)`	Devuelve el año de una fecha específica.	`SELECT YEAR('2025-11-17');` -- Resultado: 2025
`DATEADD(parte, num, fecha)`	Agrega o resta un **número** (`num`) a una **fecha** dada, especificando la **parte** de la fecha (ej: *day*, *month*, *year*).	`SELECT DATEADD(month, 3, GETDATE());` -- Agrega 3 meses a la fecha actual.
`DATEDIFF(parte, fecha1, fecha2)`	Devuelve la diferencia entre dos fechas en la **parte** especificada (ej: *day*, *month*, *year*).	`SELECT DATEDIFF(year, '1990-01-01', GETDATE());` -- Años transcurridos.

Funciones de Cadena (String Functions - SQL Server/T-SQL)

Las funciones de cadena se usan para manipular y formatear datos de texto.

Función	Descripción	Código de Ejemplo
`CONCAT(c1, c2, ...)`	Une dos o más expresiones de cadena en una sola cadena.	`SELECT CONCAT(Nombre, ' ', Apellido) FROM Clientes;`
`LEN(cadena)`	Devuelve el número de caracteres de una expresión de cadena.	`SELECT LEN('SQL');` -- Resultado: 3
`SUBSTRING(cadena, inicio, largo)`	Extrae una subcadena de una cadena. `inicio` es la posición de inicio (1-basado).	`SELECT SUBSTRING('BaseDatos', 5, 5);` -- Resultado: Datos
`LOWER(cadena)`	Convierte todos los caracteres de una cadena a minúsculas.	`SELECT LOWER('Ejemplo');` -- Resultado: ejemplo
`UPPER(cadena)`	Convierte todos los caracteres de una cadena a mayúsculas.	`SELECT UPPER('Ejemplo');` -- Resultado: EJEMPLO
`LTRIM(cadena)`	Elimina los espacios en blanco iniciales (a la izquierda) de una cadena.	`SELECT LTRIM(' Hola ');` -- Resultado: 'Hola '
`RTRIM(cadena)`	Elimina los espacios en blanco finales (a la derecha) de una cadena.	`SELECT RTRIM(' Hola ');` -- Resultado: ' Hola'

🔄 Funciones de Conversión (Conversion Functions - SQL Server/T-SQL)

Las funciones de conversión se usan para cambiar explícitamente el tipo de datos de una expresión.

1. `CAST`

`CAST` es la función de conversión estándar ANSI SQL.

Teoría	Código de Ejemplo
Convierte una expresión de un tipo de dato a otro. Sintaxis: `CAST(expresion AS tipo_dato)`. Es generalmente preferida por su portabilidad.	`sql -- Convertir un número entero a VARCHAR SELECT 'El ID es: ' + CAST(1234 AS VARCHAR(10));  -- Convertir DATETIME a DATE SELECT CAST(GETDATE() AS DATE); `

2. `CONVERT`

`CONVERT` es específica de SQL Server (T-SQL) y ofrece la opción adicional de un **estilo** al convertir tipos como fecha/hora o moneda.

Teoría	Código de Ejemplo
Convierte una expresión de un tipo de dato a otro. Sintaxis: `CONVERT(tipo_dato, expresion [, estilo])`. El parámetro `estilo` permite formatos específicos.	`sql -- Convertir DATETIME a VARCHAR en formato d/m/a (estilo 103) SELECT CONVERT(VARCHAR(10), GETDATE(), 103); `

🛠 Problema Práctico de Aplicación

Escenario:

Necesitas obtener una lista de productos vendidos en el año actual que tienen nombres de más de 10 caracteres y concatenar su nombre con su código en mayúsculas.

- EJECUCION **********

--
==========

-- EJEMPLO DE CÓDIGO SQL COMPLETO

-- APLICANDO SUBCONSULTAS, FUNCIONES DE FECHA, CADENA Y CONVERSIÓN

```
--
================================================================================
==========


-- 1. Consulta Principal: Recuperar información de productos vendidos
SELECT
    -- A. Funciones de Cadena (CONCAT, UPPER, LOWER, SUBSTRING, LEN)
    -- Genera un código de informe usando la inicial del nombre, los 4 primeros del código y el año.
    CONCAT(
        UPPER(SUBSTRING(p.Nombre, 1, 1)),           -- Primera letra en MAYÚSCULAS
        '-',
        SUBSTRING(p.CodigoProducto, 1, 4),          -- Cuatro primeros caracteres del código
        '-',
        LOWER(p.Categoria)                          -- Categoría en minúsculas
    ) AS CodigoInforme,

    -- B. Funciones de Conversión (CAST)
    -- Muestra el precio con una precisión decimal fija.
    CAST(p.Precio AS DECIMAL(10, 2)) AS PrecioUnitario,

    -- C. Funciones de Cadena (LEN)
    LEN(p.Nombre) AS LongitudNombre,

    -- D. Funciones de Fecha (YEAR, MONTH)
    YEAR(pe.FechaPedido) AS AnioPedido,
    MONTH(pe.FechaPedido) AS MesPedido,

    -- E. Funciones de Fecha (GETDATE, DATEDIFF)
    -- Calcula cuántos días han pasado desde el pedido hasta hoy.
    DATEDIFF(day, pe.FechaPedido, GETDATE()) AS DiasTranscurridos
```

```
FROM
    Productos p
INNER JOIN
    Pedidos pe ON p.ProductoID = pe.ProductoID

-- 2. Cláusula WHERE con SUBCONSULTA IN (Filtrado por productos de una categoría específica)
WHERE
    p.CategoriaID IN (
        -- Subconsulta: Encontrar los IDs de categorías llamadas 'Electrónica' o 'Ropa'
        SELECT CategoriaID
        FROM Categorias
        WHERE NombreCategoria IN ('Electrónica', 'Ropa')
    )

    -- 3. Aplicación de Funciones de Fecha (DATEADD)
    -- Incluye solo pedidos que se realizaron hace más de 365 días (un año)
    AND pe.FechaPedido < DATEADD(year, -1, GETDATE())

    -- 4. Funciones de Cadena (LTRIM, RTRIM) y CONVERSION (CONVERT)
    -- Ejemplo de limpieza de datos si el nombre tuviera espacios, y conversión de fecha
    AND RTRIM(LTRIM(p.Nombre)) IS NOT NULL

    -- 5. SUBCONSULTA EXISTS (Filtrado por existencia de un requisito relacionado)
    -- Solo incluye productos donde el precio sea mayor al precio promedio de su categoría.
    AND EXISTS (
        SELECT 1
        FROM Productos p2
        WHERE p2.CategoriaID = p.CategoriaID
        GROUP BY p2.CategoriaID
        HAVING p.Precio > AVG(p2.Precio)
```

```
    )

ORDER BY
    -- Ordenar usando la función de conversión CONVERT para un formato específico de fecha
    CONVERT(VARCHAR(10), pe.FechaPedido, 120); -- Estilo 120: yyyy-mm-dd
```

SEMANA 13 Transact SQL

⚙ Función vs. Procedimiento Almacenado para Cálculos

La utilidad de una **función** en lugar de un procedimiento almacenado para realizar cálculos en SQL Server se basa en tres puntos clave: **reusabilidad en consultas**, **devolución de un valor único** y **composición**.

Característica	Función Escalar	Procedimiento Almacenado
Uso Principal	Realizar cálculos y devolver un valor único.	Ejecutar lógica compleja, modificar datos, o devolver conjuntos de resultados.
Composición	**Puede usarse directamente** en cláusulas `SELECT`, `WHERE` y `HAVING`.	**No puede usarse** directamente dentro de estas cláusulas; debe ejecutarse con `EXEC`.
Devolución	Debe devolver un **valor único** (escalar) o una **tabla**.	Puede devolver múltiples conjuntos de resultados, parámetros de salida, o un valor de estado.
Transacciones	**No puede** modificar el estado de la base de datos (p. ej., usar `INSERT`, `UPDATE`, `DELETE`).	**Sí puede** modificar datos y gestionar transacciones (`COMMIT`, `ROLLBACK`).

Exportar a Hojas de cálculo

¿Por qué es útil una función para cálculos?

1. **Integración Directa en Consultas:** Puedes tratar una función como una columna calculada dentro de una consulta. Por ejemplo:

SQL

```
SELECT
    Nombre,
    dbo.CalcularImpuesto(Precio) AS PrecioConImpuesto -- Uso
directo
FROM Productos;
```

2. **Reutilización:** Si el cálculo (ej. calcular la edad a partir de una fecha de nacimiento) se necesita en varias consultas, la función lo centraliza, facilitando el mantenimiento.
3. **Encapsulamiento:** Simplifica la consulta principal al encapsular la lógica compleja del cálculo, haciendo el código más legible.

---

2. 📝 Manejo de Variables en T-SQL con Datos Dinámicos

El manejo de **variables** (`DECLARE @miVariable tipo_dato;`) en T-SQL es crucial para trabajar con datos dinámicos, ya que te ayuda a:

1. **Almacenar Valores Temporales:** Permiten guardar y reutilizar valores intermedios (como el resultado de un cálculo o el ID de una fila recién insertada) sin tener que recalcularlos o consultarlos repetidamente.
2. **Control de Flujo:** Son esenciales para las estructuras de control (`IF`, `WHILE`) y la lógica condicional dentro de los procedimientos almacenados.

   SQL

   ```
   DECLARE @PromedioVentas DECIMAL(10, 2);
   SET @PromedioVentas = (SELECT AVG(TotalVenta) FROM Pedidos);

   IF @PromedioVentas > 1000
   BEGIN
       -- Ejecuta lógica de alto rendimiento
   END
   ```

3. **Creación de SQL Dinámico:** Las variables son fundamentales para construir cadenas de consulta SQL que cambian en tiempo de ejecución (SQL dinámico). Esto es útil cuando el usuario define los criterios de filtro o el orden de la consulta.

   Advertencia: Al usar SQL dinámico, siempre sanitiza las entradas del usuario para prevenir ataques de **Inyección SQL**.

---

3. 🔓 Ventajas de los Cursores en T-SQL

Un **cursor** permite procesar las filas de un conjunto de resultados una por una, de forma secuencial.

Generalmente, se recomienda evitar los cursores porque son inherentemente lentos (*operaciones fila por fila* o *row-by-row*) en comparación con las operaciones basadas en conjuntos (*set-based*), como `JOIN` o `SELECT` simples.

Sin embargo, los cursores son ventajosos y a veces necesarios en situaciones muy específicas donde las técnicas basadas en conjuntos no funcionan de manera eficiente o son imposibles:

1. **Operaciones Iterativas con Lógica Externa:** Cuando necesitas realizar una acción en cada fila que **depende del resultado de la operación anterior** o que requiere llamar a un sistema externo o a un procedimiento almacenado para cada fila.
2. **Mantenimiento Administrativo:** En scripts de administración de bases de datos (DBA) para realizar tareas como:
    - Generar un script de respaldo o mantenimiento para cada base de datos en el servidor.
    - Procesar una lista de bases de datos para cambiar permisos.
3. **Lógica Compleja de Negocio:** Cuando la lógica de negocio es intrínsecamente iterativa y no puede expresarse fácilmente de manera *set-based*. Por ejemplo, simular un bucle que actualiza el saldo de una cuenta después de cada transacción secuencial.

Regla de Oro: Solo utiliza un cursor cuando hayas agotado todas las opciones basadas en conjuntos (tablas temporales, CTEs, *joins*).

EJECUCION**********

```
--
============================================================
======

-- SCRIPT T-SQL DE DEMOSTRACIÓN

-- Incluye: Variables, Funciones, Procedimientos Almacenados, Condicionales, y
Cursor

-- Asume la existencia de las tablas: Productos (ID, Nombre, Precio, Stock), Pedidos
(ID, Total)

--
============================================================
======
```

```
-- 1. Definición de una Función Escalar (Cálculo)

-- Utilidad: Calcular el precio final incluyendo un impuesto fijo.

IF OBJECT_ID('dbo.CalcularPrecioFinalConImpuesto') IS NOT NULL DROP
FUNCTION dbo.CalcularPrecioFinalConImpuesto;

GO


CREATE FUNCTION dbo.CalcularPrecioFinalConImpuesto

(

    @PrecioBase DECIMAL(10, 2)

)

RETURNS DECIMAL(10, 2)

AS

BEGIN

    DECLARE @Impuesto DECIMAL(4, 2) = 0.18; -- 18% de impuesto

    DECLARE @PrecioFinal DECIMAL(10, 2);


    SET @PrecioFinal = @PrecioBase * (1 + @Impuesto);


    RETURN @PrecioFinal;

END

GO


-- 2. Definición del Procedimiento Almacenado Principal (Automatización y Lógica)

IF OBJECT_ID('dbo.ProcesarInventarioCritico') IS NOT NULL DROP PROCEDURE
dbo.ProcesarInventarioCritico;

GO
```

```
CREATE PROCEDURE dbo.ProcesarInventarioCritico

AS

BEGIN

    -- Declaración de Variables (Manejo de Datos Dinámicos)

    DECLARE @ProductoID INT;

    DECLARE @NombreProducto VARCHAR(100);

    DECLARE @StockActual INT;

    DECLARE @Precio DECIMAL(10, 2);

    DECLARE @UmbralCritico INT = 10;

    DECLARE @Mensaje VARCHAR(200);


    -- 3. Uso de Cursor (Procesamiento Fila por Fila)

    -- Situación: Se requiere un cursor para actualizar una tabla externa o llamar a un SP externo por cada fila.

    -- Aquí, simularemos solo el recorrido y el mensaje.

    DECLARE ProductoCursor CURSOR FOR

    SELECT ProductoID, Nombre, Stock, Precio

    FROM Productos

    WHERE Stock <= @UmbralCritico; -- Filtra solo el inventario crítico


    OPEN ProductoCursor;

    FETCH NEXT FROM ProductoCursor INTO @ProductoID, @NombreProducto, @StockActual, @Precio;


    WHILE @@FETCH_STATUS = 0
```

```
BEGIN
    -- 4. Uso de Condicionales (Personalización de la Lógica)
    IF @StockActual < 5
    BEGIN
        SET @Mensaje = CONCAT(
            'ALERTA MAXIMA: ',
            @NombreProducto,
            ' tiene solo ',
            CAST(@StockActual AS VARCHAR),
            ' unidades. Precio Final: ',
            -- 5. Llamada a la Función Escalar
            CAST(dbo.CalcularPrecioFinalConImpuesto(@Precio) AS VARCHAR)
        );

        -- Simulación de una acción compleja (ej. mandar un email)
        PRINT @Mensaje;

        -- Ejemplo de uso de CAST y CONVERT
        SELECT
            CONVERT(VARCHAR, GETDATE(), 108) AS HoraRegistro,
            @Mensaje AS Detalle;
    END
    ELSE
    BEGIN
        SET @Mensaje = CONCAT('Alerta Media: ', @NombreProducto, ' necesita ser
revisado.');
```

```
            PRINT @Mensaje;

        END


        FETCH NEXT FROM ProductoCursor INTO @ProductoID, @NombreProducto,
@StockActual, @Precio;

    END


    CLOSE ProductoCursor;

    DEALLOCATE ProductoCursor;


    -- Lógica basada en conjuntos después del cursor (Si es posible, se prefiere)

    -- Ejemplo: Actualizar un campo de 'Prioridad' para todos los productos críticos de
forma eficiente.

    UPDATE Productos

    SET PrioridadReposicion = 1

    WHERE Stock <= @UmbralCritico;


END

GO


-- EJEMPLO DE EJECUCIÓN (ASUMIENDO DATOS)


-- 1. Crear tablas de ejemplo (si no existen)

IF OBJECT_ID('Productos') IS NULL

CREATE TABLE Productos (

    ProductoID INT PRIMARY KEY,
```

```
    Nombre VARCHAR(100),
    Precio DECIMAL(10, 2),
    Stock INT,
    PrioridadReposicion INT DEFAULT 0
);
GO

-- 2. Insertar datos de ejemplo
TRUNCATE TABLE Productos;
INSERT INTO Productos (ProductoID, Nombre, Precio, Stock) VALUES
(101, 'Teclado Mecánico', 120.00, 3),    -- Crítico (Stock < 5)
(102, 'Mouse Inalámbrico', 45.50, 8),    -- Alerta Media
(103, 'Monitor 4K', 450.00, 25),         -- Stock Normal
(104, 'Webcam HD', 70.00, 1);            -- Crítico (Stock < 5)
GO

-- 3. Ejecutar el Procedimiento Almacenado
EXEC dbo.ProcesarInventarioCritico;
GO

-- 4. Verificar la actualización realizada por el Procedimiento
SELECT * FROM Productos;
GO
```

SEMANA 14 - Manejo de transacciones

¡Excelente! Aquí tienes el código T-SQL de demostración, seguido de la explicación teórica sobre la importancia de las transacciones.

💾 Código T-SQL de Transacciones

El siguiente código ejemplifica una transacción con sus comandos clave (`BEGIN TRANSACTION`, `COMMIT`, `ROLLBACK`) en un escenario de transferencia de fondos.

SQL

```
--
=============================================================================
-- SCRIPT T-SQL: DEMOSTRACIÓN DE TRANSACCIONES Y ROLLBACK
-- Asume la existencia de la tabla Cuentas (ID, Saldo)
--
=============================================================================

-- 1. Crear tabla de ejemplo (si no existe)
IF OBJECT_ID('Cuentas') IS NULL
CREATE TABLE Cuentas (
    ID INT PRIMARY KEY,
    NombreCliente VARCHAR(100),
    Saldo DECIMAL(10, 2)
);
GO

-- 2. Inicializar datos
TRUNCATE TABLE Cuentas;
INSERT INTO Cuentas (ID, NombreCliente, Saldo) VALUES
(1, 'Alice', 1000.00), -- Cuenta Origen
(2, 'Bob', 50.00);     -- Cuenta Destino
GO

-- Declarar variables
DECLARE @ID_Origen INT = 1;
DECLARE @ID_Destino INT = 2;
DECLARE @MontoTransferencia DECIMAL(10, 2) = 200.00;
DECLARE @SaldoOrigen DECIMAL(10, 2);

-- Mostrar saldos iniciales
PRINT '--- SALDOS INICIALES ---';
SELECT ID, NombreCliente, Saldo FROM Cuentas WHERE ID IN (@ID_Origen,
@ID_Destino);

-- ==================== INICIO DE LA TRANSACCIÓN ====================
BEGIN TRANSACTION T1;

BEGIN TRY
    -- 1. Restar el monto de la cuenta de origen
    UPDATE Cuentas
    SET Saldo = Saldo - @MontoTransferencia
    WHERE ID = @ID_Origen;

    -- Verificar el saldo de la cuenta de origen después de la resta
    SELECT @SaldoOrigen = Saldo FROM Cuentas WHERE ID = @ID_Origen;
```

```
    -- 2. Validación de Integridad (Simulación de un fallo)
    IF @SaldoOrigen < 0
    BEGIN
        -- Si el saldo es negativo (fondos insuficientes), forzamos un
error.
        -- Esto desencadenará el bloque CATCH y el ROLLBACK.
        RAISERROR('Error: Fondos insuficientes en la cuenta de
origen.', 16, 1);
    END

    -- 3. Sumar el monto a la cuenta de destino (Solo si la resta fue
exitosa)
    UPDATE Cuentas
    SET Saldo = Saldo + @MontoTransferencia
    WHERE ID = @ID_Destino;

    -- Si ambas operaciones tienen éxito, confirmamos los cambios.
    COMMIT TRANSACTION T1;
    PRINT '--- TRANSACCIÓN EXITOSA (COMMIT) ---';

END TRY
BEGIN CATCH
    -- Si ocurre algún error (incluyendo RAISERROR), se ejecuta el
ROLLBACK.
    IF @@TRANCOUNT > 0
        ROLLBACK TRANSACTION T1;

    PRINT '--- TRANSACCIÓN FALLIDA (ROLLBACK) ---';
    -- Mensaje de error (opcional)
    PRINT ERROR_MESSAGE();
END CATCH
-- ===================== FIN DE LA TRANSACCIÓN ====================

-- Mostrar saldos finales
PRINT '--- SALDOS FINALES ---';
SELECT ID, NombreCliente, Saldo FROM Cuentas WHERE ID IN (@ID_Origen,
@ID_Destino);
```

📓 Teoría de Transacciones en SQL

Las transacciones son el corazón de la gestión de bases de datos, adhiriéndose al principio **ACID** (Atomicidad, Consistencia, Aislamiento y Durabilidad), que garantiza la fiabilidad de los datos.

1. ¿Por qué es importante utilizar transacciones al realizar múltiples operaciones SQL?

La importancia radica en el principio de **Atomicidad**: asegurar que un conjunto de operaciones (a menudo llamadas "unidades de trabajo") se complete en su totalidad o no se complete en absoluto.

- **Integridad de Datos:** Si estás transfiriendo dinero, necesitas garantizar que la cantidad sea **restada** de una cuenta Y **sumada** a otra. Si el sistema falla después

de la resta pero antes de la suma, el dinero desaparecería. Una transacción impide este estado inconsistente.
- **Coherencia Lógica:** Mantiene las reglas y las restricciones de la base de datos. Una transferencia solo es válida si se realiza como una única operación lógica, no como dos operaciones separadas.
- **Punto de Recuperación:** Establece un punto de guardado (el inicio de la transacción) al cual se puede volver si algo sale mal.

---

2. ¿Cómo ayuda el comando `ROLLBACK` a mantener la integridad de los datos?

El comando `ROLLBACK` es la herramienta que materializa la **Atomicidad** y la **Consistencia**.

- **Revierte Cambios:** El `ROLLBACK` **deshace** todas las modificaciones de datos que se hayan realizado desde el inicio de la transacción (`BEGIN TRANSACTION`) hasta el momento en que se ejecuta. Es como un botón "deshacer" para la base de datos.
- **Manejo de Errores:** En un entorno de producción, cualquier operación fallida (por ejemplo, fondos insuficientes, *deadlocks*, o errores del sistema) debe provocar un `ROLLBACK`. Esto garantiza que la base de datos regrese a su último estado válido, manteniendo la integridad referencial y la coherencia lógica.
- **Ejemplo:** En el código anterior, si el saldo se vuelve negativo después del primer `UPDATE`, el `ROLLBACK` asegura que la base de datos se comporte como si la transacción nunca hubiera comenzado: el dinero no se resta y la cuenta de destino no se toca.

---

3. ¿En qué situaciones sería crítico utilizar `COMMIT` para asegurar los cambios en la base de datos?

El comando `COMMIT` finaliza exitosamente la transacción y hace que todos los cambios sean **permanentes** y **visibles** para otros usuarios. Es crítico utilizar `COMMIT` en cualquier situación donde la **Durabilidad** y la **Consistencia** sean vitales.

Situación Crítica	**Razón por la que el COMMIT es clave**
Transferencias Bancarias/Contabilidad	Asegura que la doble entrada contable (débito y crédito) se registre de forma inmutable.

Situación Crítica	Razón por la que el COMMIT es clave
Actualización de Inventario	Garantiza que la venta de un producto se refleje como una reducción en el inventario y una adición en las ventas al mismo tiempo.
Confirmación de Pedidos	Al pasar un pedido de `PENDIENTE` a `CONFIRMADO`, se deben actualizar varias tablas (estado, existencias, historial). El `COMMIT` garantiza que todas estas actualizaciones ocurran juntas.
Grandes Migraciones de Datos	Si se insertan o actualizan miles de registros, un `COMMIT` periódico o final asegura que el trabajo no se pierda en caso de un fallo del sistema.

EJECUCION{**********

```
-- ====================================================================
-- SCRIPT T-SQL DE DEMOSTRACIÓN
-- Incluye: Variables, Funciones, Procedimientos Almacenados, Condicionales, y Cursor
-- Asume la existencia de las tablas: Productos (ID, Nombre, Precio, Stock), Pedidos (ID, Total)
-- ====================================================================

-- 1. Definición de una Función Escalar (Cálculo)
-- Utilidad: Calcular el precio final incluyendo un impuesto fijo.
IF OBJECT_ID('dbo.CalcularPrecioFinalConImpuesto') IS NOT NULL DROP FUNCTION dbo.CalcularPrecioFinalConImpuesto;
GO

CREATE FUNCTION dbo.CalcularPrecioFinalConImpuesto
(
    @PrecioBase DECIMAL(10, 2)
```

```
)
RETURNS DECIMAL(10, 2)
AS
BEGIN
    DECLARE @Impuesto DECIMAL(4, 2) = 0.18; -- 18% de impuesto
    DECLARE @PrecioFinal DECIMAL(10, 2);

    SET @PrecioFinal = @PrecioBase * (1 + @Impuesto);

    RETURN @PrecioFinal;
END
GO

-- 2. Definición del Procedimiento Almacenado Principal (Automatización y Lógica)
IF OBJECT_ID('dbo.ProcesarInventarioCritico') IS NOT NULL DROP PROCEDURE
dbo.ProcesarInventarioCritico;
GO

CREATE PROCEDURE dbo.ProcesarInventarioCritico
AS
BEGIN
    -- Declaración de Variables (Manejo de Datos Dinámicos)
    DECLARE @ProductoID INT;
    DECLARE @NombreProducto VARCHAR(100);
    DECLARE @StockActual INT;
    DECLARE @Precio DECIMAL(10, 2);
    DECLARE @UmbralCritico INT = 10;
    DECLARE @Mensaje VARCHAR(200);

    -- 3. Uso de Cursor (Procesamiento Fila por Fila)
    -- Situación: Se requiere un cursor para actualizar una tabla externa o llamar a un SP
externo por cada fila.
```

```
    -- Aquí, simularemos solo el recorrido y el mensaje.
    DECLARE ProductoCursor CURSOR FOR
    SELECT ProductoID, Nombre, Stock, Precio
    FROM Productos
    WHERE Stock <= @UmbralCritico; -- Filtra solo el inventario crítico

    OPEN ProductoCursor;
    FETCH NEXT FROM ProductoCursor INTO @ProductoID, @NombreProducto,
@StockActual, @Precio;

    WHILE @@FETCH_STATUS = 0
    BEGIN
        -- 4. Uso de Condicionales (Personalización de la Lógica)
        IF @StockActual < 5
        BEGIN
            SET @Mensaje = CONCAT(
                'ALERTA MAXIMA: ',
                @NombreProducto,
                ' tiene solo ',
                CAST(@StockActual AS VARCHAR),
                ' unidades. Precio Final: ',
                -- 5. Llamada a la Función Escalar
                CAST(dbo.CalcularPrecioFinalConImpuesto(@Precio) AS VARCHAR)
            );

            -- Simulación de una acción compleja (ej. mandar un email)
            PRINT @Mensaje;

            -- Ejemplo de uso de CAST y CONVERT
            SELECT
                CONVERT(VARCHAR, GETDATE(), 108) AS HoraRegistro,
```

```
            @Mensaje AS Detalle;
        END
        ELSE
        BEGIN
            SET @Mensaje = CONCAT('Alerta Media: ', @NombreProducto, ' necesita ser revisado.');
            PRINT @Mensaje;
        END

        FETCH NEXT FROM ProductoCursor INTO @ProductoID, @NombreProducto, @StockActual, @Precio;
    END

    CLOSE ProductoCursor;
    DEALLOCATE ProductoCursor;

    -- Lógica basada en conjuntos después del cursor (Si es posible, se prefiere)
    -- Ejemplo: Actualizar un campo de 'Prioridad' para todos los productos críticos de forma eficiente.
    UPDATE Productos
    SET PrioridadReposicion = 1
    WHERE Stock <= @UmbralCritico;

END
GO

-- ================================================================
-- EJEMPLO DE EJECUCIÓN (ASUMIENDO DATOS)
-- ================================================================

-- 1. Crear tablas de ejemplo (si no existen)
IF OBJECT_ID('Productos') IS NULL
```

```
CREATE TABLE Productos (
    ProductoID INT PRIMARY KEY,
    Nombre VARCHAR(100),
    Precio DECIMAL(10, 2),
    Stock INT,
    PrioridadReposicion INT DEFAULT 0
);
GO

-- 2. Insertar datos de ejemplo
TRUNCATE TABLE Productos;
INSERT INTO Productos (ProductoID, Nombre, Precio, Stock) VALUES
(101, 'Teclado Mecánico', 120.00, 3),    -- Crítico (Stock < 5)
(102, 'Mouse Inalámbrico', 45.50, 8),    -- Alerta Media
(103, 'Monitor 4K', 450.00, 25),         -- Stock Normal
(104, 'Webcam HD', 70.00, 1);            -- Crítico (Stock < 5)
GO

-- 3. Ejecutar el Procedimiento Almacenado
EXEC dbo.ProcesarInventarioCritico;
GO

-- 4. Verificar la actualización realizada por el Procedimiento
SELECT * FROM Productos;
GO
```

SEMANA 15 - Triggers

1. 🎯 Diferencia Principal: `AFTER` vs. `INSTEAD OF`

La diferencia fundamental reside en **cuándo** se ejecuta el *trigger* con respecto a la acción de modificación de datos (INSERT, UPDATE, DELETE).

Característica	AFTER Trigger (o FOR Trigger)	INSTEAD OF Trigger
Tiempo de Ejecución	Se ejecuta **después** de que la acción de modificación (INSERT/UPDATE/DELETE) se ha completado en la tabla de destino.	Se ejecuta **en lugar de** (INSTEAD OF) la acción de modificación en la tabla de destino.
Integridad de Datos	La acción de datos se realiza primero. Si hay fallas de integridad (ej. *Primary Key* duplicada), la acción **falla** antes de que se ejecute el *trigger*.	La acción de datos **no se realiza** en la tabla base. El *trigger* es responsable de ejecutar la lógica de datos.
Tablas Aplicables	Tablas y Vistas.	**Tablas** y **Vistas** (es el único *trigger* que funciona con vistas que no son actualizables de forma natural).

Escenarios de Uso

Tipo de Trigger	Situación Preferida	Ejemplo
`AFTER`	Para mantener la **consistencia de datos** y la **auditoría** después de que una operación ha sido validada por las restricciones de la base de datos.	Después de una `INSERT` en la tabla `Pedidos`, se ejecuta un *trigger* `AFTER INSERT` para **actualizar el *stock*** en la tabla `Productos` y **registrar un evento** en la tabla `Auditoria`.
`INSTEAD OF`	Para **interceptar la operación** y redirigirla a otra lógica, especialmente para **simplificar la modificación de vistas complejas** (que involucran múltiples tablas).	Al intentar hacer un `UPDATE` en una **vista** que une `Clientes` y `Direcciones`, el *trigger* `INSTEAD OF UPDATE` intercepta la solicitud y la divide en dos `UPDATE` separados, uno para cada tabla subyacente.

---

2. 🔁 Evitar la Recursividad Infinita y Manejar Triggers Anidados

Prevención de Recursividad Infinita

La **recursividad infinita** ocurre cuando un *trigger* ejecuta una acción que, a su vez, activa el mismo *trigger* de forma repetida.

La mejor práctica para prevenirla es usar la variable de entorno **`@@NESTLEVEL`**.

Técnica	Descripción	Código T-SQL
`@@NESTLEVEL`	Esta variable devuelve la profundidad de anidamiento de la ejecución actual. Si es mayor a 1, significa que el *trigger* fue llamado por otro *trigger* o por sí mismo, y se puede salir.	`sql IF @@NESTLEVEL > 1  RETURN; `
Desactivar Recursividad	SQL Server permite desactivar la recursividad de *triggers* a nivel de base de datos (`RECURSIVE_TRIGGERS`). Sin embargo, es una configuración **global** y menos flexible.	`ALTER DATABASE [DBName] SET RECURSIVE_TRIGGERS OFF;`

Manejo de Triggers Anidados

Los *triggers* **anidados** ocurren cuando un *trigger* en la Tabla A realiza una acción que activa un *trigger* en la Tabla B. SQL Server permite hasta **32 niveles de anidamiento** por defecto.

- **Configuración:** El manejo global de *triggers* anidados se controla con la opción **`nested triggers`** del servidor, que está **activada** por defecto.
- **Mejor Práctica:** Si necesitas control preciso sobre el anidamiento, usa el comando **`SET NOCOUNT ON`** al inicio del *trigger* para reducir el tráfico de red, y asegúrate de que la lógica anidada esté bien documentada y probada para evitar efectos secundarios inesperados.

---

3. □ Impacto en el Rendimiento y Mitigación

Los *triggers* pueden tener un impacto negativo significativo en el rendimiento, ya que añaden una capa de ejecución de código *extra* y una sobrecarga de *logging* a cada operación de modificación de datos.

Impacto Negativo Clave

1. **Bloqueo de Transacciones:** El *trigger* se ejecuta dentro de la misma transacción que la operación de datos. Mientras el *trigger* se ejecuta, la transacción permanece abierta, lo que puede aumentar el tiempo de bloqueo en las tablas.
2. **Operaciones I/O Adicionales:** Si el *trigger* realiza operaciones complejas (ej. consultas, actualizaciones de otras tablas), consume recursos de disco (I/O) y CPU.
3. **Filas vs. Conjuntos:** El código dentro de los *triggers* debe estar optimizado para trabajar con **conjuntos de filas** (usando las tablas especiales **`inserted`** y **`deleted`**), y no debe intentar procesar filas una por una (evitar **cursores**).

Estrategias de Mitigación

Estrategia	Descripción	Beneficio
Trabajo Basado en Conjuntos	Siempre use las tablas `inserted` y `deleted` con operaciones como `JOIN` para trabajar con todas las filas modificadas de una vez, en lugar de iterar.	**Mejora drástica** en el rendimiento de operaciones masivas.
Minimizar la Lógica	Mueva la lógica compleja o el trabajo de auditoría pesada a un **procedimiento almacenado** y ejecútelo **fuera de línea** (ej. usando SQL Server Agent), dejando el *trigger* solo para registrar un evento.	**Reduce el tiempo** que la transacción principal permanece bloqueada.
Uso de `IF UPDATE()`	Si el *trigger* debe ejecutarse solo cuando se actualizan columnas específicas, use `IF UPDATE(NombreColumna)` para salir rápidamente si las columnas irrelevantes se modificaron.	**Evita la ejecución innecesaria** de la lógica del *trigger*.
`SET NOCOUNT ON`	Incluya esto al principio del *trigger* para evitar que SQL Server envíe un mensaje de "filas afectadas" al cliente por cada instrucción dentro del *trigger*.	**Reduce el tráfico** y la sobrecarga del servidor.

EJECUCION **********

```
-- ================================================================
-- SCRIPT T-SQL: DEMOSTRACIÓN DE TRIGGERS, RECURSIVIDAD Y RENDIMIENTO
-- Incluye: Trigger AFTER, Trigger INSTEAD OF, Prevención de Recursividad
-- ================================================================

-- 1. CONFIGURACIÓN INICIAL Y CREACIÓN DE TABLAS
-- Se crean tablas simples para simular Productos y Ventas

-- Desactivar mensajes de filas afectadas para mejorar el rendimiento
SET NOCOUNT ON;

IF OBJECT_ID('Productos') IS NOT NULL DROP TABLE Productos;
IF OBJECT_ID('Ventas') IS NOT NULL DROP TABLE Ventas;
IF OBJECT_ID('Vw_InventarioCritico') IS NOT NULL DROP VIEW Vw_InventarioCritico;
GO

CREATE TABLE Productos (
    ProductoID INT PRIMARY KEY,
    Nombre VARCHAR(100),
    Stock INT,
    Precio DECIMAL(10, 2)
);

CREATE TABLE Ventas (
    VentaID INT PRIMARY KEY IDENTITY(1,1),
    ProductoID INT,
    CantidadVendida INT,
    FechaVenta DATETIME DEFAULT GETDATE()
```

```
);
GO

-- 2. DATOS DE EJEMPLO
INSERT INTO Productos (ProductoID, Nombre, Stock, Precio) VALUES
(101, 'Laptop', 50, 1200.00),
(102, 'Monitor', 100, 300.00),
(103, 'Teclado', 20, 50.00);
GO

-- 3. TRIGGER AFTER (CON AUDITORÍA Y LÓGICA DE INVENTARIO)
-- Se ejecuta DESPUÉS de una inserción en la tabla Ventas.

IF OBJECT_ID('trg_AfterInsert_VentasActualizaStock') IS NOT NULL DROP TRIGGER
trg_AfterInsert_VentasActualizaStock;
GO

CREATE TRIGGER trg_AfterInsert_VentasActualizaStock
ON Ventas
AFTER INSERT
AS
BEGIN
    -- Mejor Práctica: SET NOCOUNT ON (Ya se estableció al inicio del script)

    -- ⚠ PREVENCIÓN DE RECURSIVIDAD ⚠
    -- Si el trigger fue llamado por otro trigger, salimos para evitar un bucle infinito.
    -- Esto es crítico si el trigger realiza una acción que podría volver a activarlo.
    IF @@NESTLEVEL > 1
        RETURN;

    -- 🚀 MITIGACIÓN DE RENDIMIENTO: TRABAJO BASADO EN CONJUNTOS 🚀
```

```
    -- Actualiza la tabla Productos con un solo UPDATE para todas las ventas insertadas
    UPDATE P
    SET P.Stock = P.Stock - I.CantidadVendida
    FROM Productos P
    INNER JOIN inserted I ON P.ProductoID = I.ProductoID;

    -- Lógica de Auditoría (Ejemplo de operación secundaria)
    -- Podría insertarse en una tabla de LOG.
    PRINT 'Trigger AFTER ejecutado: Stock de productos actualizado exitosamente.';

END
GO

-- 4. TRIGGER INSTEAD OF (PARA VISTAS COMPLEJAS O LÓGICA DE REDIRECCIÓN)
-- Se crea una vista simple. INSTEAD OF es vital para vistas que unen múltiples tablas.
CREATE VIEW Vw_InventarioCritico AS
SELECT
    ProductoID,
    Nombre,
    Stock
FROM
    Productos
WHERE
    Stock < 50;
GO

IF OBJECT_ID('trg_InsteadOfDelete_InventarioCritico') IS NOT NULL DROP TRIGGER
trg_InsteadOfDelete_InventarioCritico;
GO

CREATE TRIGGER trg_InsteadOfDelete_InventarioCritico
```

```
ON Vw_InventarioCritico
INSTEAD OF DELETE
AS
BEGIN
    -- El INSTEAD OF intercepta el DELETE de la vista y ejecuta nuestra propia lógica.

    -- 🚀 TRABAJO BASADO EN CONJUNTOS 🚀
    -- En lugar de ELIMINAR el producto, solo se marca como INACTIVO en la tabla base (Productos).
    -- La tabla 'deleted' en un INSTEAD OF contiene los datos que el usuario INTENTÓ eliminar.

    -- Este código previene la eliminación física de un producto al usar la vista.
    UPDATE P
    SET P.Stock = 0 -- O un campo Estatus = 'Inactivo'
    FROM Productos P
    INNER JOIN deleted D ON P.ProductoID = D.ProductoID;

    PRINT 'Trigger INSTEAD OF ejecutado: Se interceptó la eliminación y se estableció el stock a 0.';

END
GO

-- ================================================================
-- PRUEBAS DE EJECUCIÓN
-- ================================================================

PRINT '--- PRUEBA 1: INSERT que activa el TRIGGER AFTER ---';
-- La inserción actualizará el Stock en Productos (101: 50 -> 40)
INSERT INTO Ventas (ProductoID, CantidadVendida) VALUES (101, 10);
```

```
SELECT 'Stock Actualizado (AFTER)' AS Test, Stock, Nombre FROM Productos WHERE ProductoID = 101;

SELECT * FROM Ventas WHERE VentaID = (SELECT MAX(VentaID) FROM Ventas);


PRINT '--- PRUEBA 2: DELETE que activa el TRIGGER INSTEAD OF ---';

-- Intentar ELIMINAR un producto de la vista. Esto solo actualizará el Stock a 0.

DELETE FROM Vw_InventarioCritico WHERE ProductoID = 103;


SELECT 'Resultado INSTEAD OF' AS Test, Stock, Nombre FROM Productos WHERE ProductoID = 103;
```