

Universidad Tecnológica de Bolívar

BASES DE DATOS

TALLER:

**ACTIVIDAD AUTOMATIZACIÓN EN
BASES DE DATOS**

Integrantes:

Anyerson de Jesus Ayola Pereira

Diego Peña Páez

Lizzeth Ariadna Sánchez Solis

Profesor:

Aisner Jose Marrugo Juliao

Fecha: 09-05-2025

1. Introducción

2. Tabla de roles

Rol	Integrante
Administrador de Azure y funciones	Diego Peña Páez
Desarrollador de procedimientos y transacciones	Anyerson de Jesus Ayola Pereira
Automatizador y documentador técnico	Lizzeth Ariadna Sánchez Solis

3. Procedimientos

En esta sección se detallan los procedimientos almacenados desarrollados para la automatización de procesos internos en la base de datos Northwind de Azure SQL. Cada procedimiento ha sido diseñado para aportar lógica empresarial real, optimizado para su ejecución y documentado exhaustivamente. Se ha puesto especial énfasis en el manejo de errores y el uso de transacciones donde ha sido pertinente para asegurar la atomicidad e integridad de los datos.

Antes de detallar cada uno de los procedimientos almacenados, es necesario describir las estructuras de datos adicionales que se crearon para soportar la funcionalidad de algunos de ellos.

3.1. Estructuras de Datos Adicionales

Para la implementación de la lógica de negocio requerida, específicamente para el cálculo y registro de bonificaciones a empleados, se creó la siguiente tabla auxiliar:

3.1.1. Tabla EmployeeBonuses

Propósito: La tabla **EmployeeBonuses** tiene como finalidad almacenar los registros de las bonificaciones que se calculan y asignan a los empleados. Esta tabla permite llevar un historial de las bonificaciones, especificando el empleado, el período (mes y año) de la bonificación, la cantidad de pedidos gestionados que sirvieron de base para el cálculo, el monto de la bonificación y la fecha en que se realizó dicho cálculo. Su diseño incluye restricciones para asegurar la integridad de los datos y prevenir duplicados.

Definición SQL: El siguiente script T-SQL fue utilizado para crear la tabla **EmployeeBonuses** en la base de datos Northwind en Azure SQL. Se incluyen comentarios detallados para explicar cada parte de la definición de la tabla.

```
1
2  --
3  =====
4  -- Mejora el rendimiento y reduce el ruido en la salida de
   ejecuci n.
5  SET NOCOUNT ON;
6
7  -- Inicia un bloque TRY para el manejo de errores durante la
   creaci n de la tabla.
8  BEGIN TRY
9
10     IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id
   = OBJECT_ID(N'[dbo].[EmployeeBonuses]') AND type in
   (N'U'))
11     BEGIN
12
13         PRINT 'Creando la tabla [dbo].[EmployeeBonuses]...';
14
15
16         CREATE TABLE [dbo].[EmployeeBonuses](
17
18             [BonusID] [int] IDENTITY(1,1) NOT NULL PRIMARY
   KEY,
19             [EmployeeID] [int] NOT NULL,
20             [BonusMonth] [int] NOT NULL,
21             [BonusYear] [int] NOT NULL,
```

```

22         [NumberOfOrdersManaged] [int] NOT NULL,
23         [BonusAmount] [money] NOT NULL,
24         [CalculationDate] [datetime] NULL DEFAULT
            (GETDATE()),
25         CONSTRAINT FK_EmployeeBonuses_Employees FOREIGN
            KEY ([EmployeeID]) REFERENCES
            [dbo].[Employees]([EmployeeID]),
26
27         CONSTRAINT CK_BonusMonth CHECK ([BonusMonth]
            BETWEEN 1 AND 12),
28
29         CONSTRAINT UK_EmployeeBonus_MonthYear UNIQUE
            ([EmployeeID], [BonusMonth], [BonusYear])
30     );
31
32     PRINT 'Tabla [dbo].[EmployeeBonuses] creada
            exitosamente.';
33
34     ELSE
35     BEGIN
36         -- Mensaje informativo si la tabla ya exist a y no
            se realiz ninguna acci n de creaci n.
37         PRINT 'La tabla [dbo].[EmployeeBonuses] ya existe.
            No se requieren acciones.';
38
39     END
40
41     END TRY
42
43     -- Bloque CATCH para capturar cualquier error que ocurra
        durante la ejecuci n del bloque TRY.
44     BEGIN CATCH
45
46         PRINT 'Ocurri un error al intentar crear la tabla
            [dbo].[EmployeeBonuses].';
47         PRINT 'Error N mero: ' + CAST(ERROR_NUMBER() AS
            VARCHAR(10));
48         PRINT 'Mensaje de Error: ' + ERROR_MESSAGE();
49         PRINT 'Severidad del Error: ' + CAST(ERROR_SEVERITY() AS
            VARCHAR(10));
50         PRINT 'Estado del Error: ' + CAST(ERROR_STATE() AS
            VARCHAR(10));
51         PRINT 'L nea del Error: ' + CAST(ERROR_LINE() AS
            VARCHAR(10));
52
53         IF @@TRANCOUNT > 0
54             ROLLBACK TRANSACTION;

```

```

53 END CATCH
54
55 -- Restaura el comportamiento predeterminado de devolver
    mensajes de recuento de filas.
56 SET NOCOUNT OFF;
57 GO -- Ffinal del lote de Transact-SQL.

```

Listing 1: Script de creación de la tabla EmployeeBonuses

3.2. Procedimiento Almacenado: RegistrarNuevoPedido (1.1)

3.2.1. Explicación

Propósito: El procedimiento almacenado `RegistrarNuevoPedido` está diseñado para automatizar la creación de nuevos pedidos en el sistema Northwind. Su funcionalidad principal incluye la inserción del encabezado del pedido en la tabla `Orders` y el registro de cada uno de los productos solicitados en la tabla `Order Details`. Una parte crucial de este proceso es la validación de la disponibilidad de stock para cada producto antes de confirmar el pedido y la subsecuente actualización de las unidades en stock en la tabla `Products`.

Parámetros de Entrada:

- `@CustomerID` (NCHAR(5)): Identificador del cliente que realiza el pedido.
- `@EmployeeID` (INT): Identificador del empleado que gestiona el pedido.
- `@OrderDate` (DATETIME): Fecha en que se realiza el pedido.
- `@RequiredDate` (DATETIME): Fecha para la cual se requiere el pedido.
- `@ShippedDate` (DATETIME, opcional, default NULL): Fecha de envío del pedido.
- `@ShipVia` (INT): Identificador del transportista (shipper).

- **@Freight** (MONEY, opcional, default 0): Costo del flete.
- **@ShipName** (NVARCHAR(40)): Nombre del destinatario.
- **@ShipAddress** (NVARCHAR(60)): Dirección de envío.
- **@ShipCity** (NVARCHAR(15)): Ciudad de envío.
- **@ShipRegion** (NVARCHAR(15), opcional, default NULL): Región de envío.
- **@ShipPostalCode** (NVARCHAR(10), opcional, default NULL): Código postal de envío.
- **@ShipCountry** (NVARCHAR(15)): País de envío.
- **@OrderDetailsTVP** (OrderDetailType READONLY): Parámetro de tipo tabla (TVP) que contiene los detalles de los productos del pedido. Cada fila del TVP debe incluir **ProductID**, **UnitPrice**, **Quantity**, y **Discount**.

Lógica de Negocio y Operaciones:

1. **Validaciones Iniciales:** Se verifica la existencia del **CustomerID**, **EmployeeID** y **ShipVia** proporcionados.
2. **Validación de Stock:** Antes de procesar el pedido, se itera sobre cada producto listado en **@OrderDetailsTVP**. Para cada producto, se consulta la tabla **Products** para verificar si existe y si la cantidad en **UnitsInStock** es suficiente para cubrir la cantidad solicitada. Si un producto no existe o el stock es insuficiente, la operación se aborta y se lanza un error específico.
3. **Inserción en Orders:** Si todas las validaciones de stock son exitosas, se inserta un nuevo registro en la tabla **Orders** con la información del encabezado del pedido. Se captura el **OrderID** generado para este nuevo pedido.
4. **Inserción en Order Details:** Utilizando el **OrderID** obtenido, se insertan los registros correspondientes a cada producto del pedido en la tabla **Order Details**, tomando la información del TVP **@OrderDetailsTVP**.

5. **Actualización de Inventario:** Simultáneamente, se actualiza la columna `UnitsInStock` en la tabla `Products` para cada producto vendido, restando la cantidad solicitada.

Transacciones y Manejo de Errores: Todo el proceso de registro del pedido se encapsula dentro de una **transacción explícita** (`BEGIN TRANSACTION ... COMMIT/ROLLBACK TRANSACTION`). Esto asegura la atomicidad de la operación: o todas las inserciones y actualizaciones se completan con éxito, o si ocurre algún error (por ejemplo, stock insuficiente, error de base de datos), se revierten todos los cambios (`ROLLBACK TRANSACTION`), manteniendo la base de datos en un estado consistente. Se utiliza un bloque `TRY...CATCH` para gestionar errores. Si se produce una excepción, la transacción se revierte y el error es relanzado para ser capturado por la aplicación cliente. Se utilizan `THROW` para generar errores personalizados con mensajes descriptivos.

Salida: Si el pedido se registra exitosamente, el procedimiento devuelve un conjunto de resultados con una única columna `NewOrderID`, que contiene el ID del pedido recién creado. En caso de error, se lanza una excepción.

Tipo de Tabla Auxiliar OrderDetailType: Para facilitar el paso de múltiples líneas de detalle de pedido, se creó previamente un tipo de tabla definido por el usuario (TVP) llamado `OrderDetailType`:

```
1 IF TYPE_ID(N'OrderDetailType') IS NULL
2 BEGIN
3     CREATE TYPE OrderDetailType AS TABLE (
4         ProductID INT NOT NULL,
5         UnitPrice MONEY NOT NULL,
6         Quantity SMALLINT NOT NULL,
7         Discount REAL NOT NULL CHECK (Discount >= 0 AND
            Discount <= 1)
8     );
9     PRINT 'Tipo de tabla OrderDetailType creado
        exitosamente.';
10 END
11 ELSE
12     PRINT 'Tipo de tabla OrderDetailType ya existe.';
13 GO
```

Listing 2: Definición del TVP OrderDetailType

Código SQL del Procedimiento:

```
1 CREATE OR ALTER PROCEDURE dbo.RegistrarNuevoPedido
2     -- Par metros para la tabla Orders
3     @CustomerID NCHAR(5),
4     @EmployeeID INT,
5     @OrderDate DATETIME,
6     @RequiredDate DATETIME,
7     @ShippedDate DATETIME = NULL,
8     @ShipVia INT,
9     @Freight MONEY = 0,
10    @ShipName NVARCHAR(40),
11    @ShipAddress NVARCHAR(60),
12    @ShipCity NVARCHAR(15),
13    @ShipRegion NVARCHAR(15) = NULL,
14    @ShipPostalCode NVARCHAR(10) = NULL,
15    @ShipCountry NVARCHAR(15),
16    -- Par metro de tipo tabla para los detalles del pedido
17    @OrderDetailsTVP OrderDetailType READONLY
18 AS
19 BEGIN
20     SET NOCOUNT ON;
21
22     DECLARE @NewOrderID INT;
23     DECLARE @CurrentProductID INT;
24     DECLARE @CurrentQuantity SMALLINT;
25     DECLARE @StockAvailable SMALLINT;
26     DECLARE @ProductName NVARCHAR(40);
27     DECLARE @ErrorMessage NVARCHAR(2048); -- Variable para
28         construir mensajes de error
29
30     BEGIN TRANSACTION;
31
32     BEGIN TRY
33         -- 1. Validar existencia de CustomerID, EmployeeID y
34         ShipperID
35         IF NOT EXISTS (SELECT 1 FROM dbo.Customers WHERE
36             CustomerID = @CustomerID)
37         BEGIN
38             SET @ErrorMessage = FORMATMESSAGE('El CustomerID
39             proporcionado ''%s'' no existe.',
40             @CustomerID);
```



```

36         THROW 50001, @ErrorMessage, 1;
37     END
38
39     IF NOT EXISTS (SELECT 1 FROM dbo.Employees WHERE
40         EmployeeID = @EmployeeID)
41     BEGIN
42         SET @ErrorMessage = FORMATMESSAGE('El EmployeeID
43         proporcionado''%d'' no existe.',
44         @EmployeeID);
45         THROW 50002, @ErrorMessage, 1;
46     END
47
48     IF NOT EXISTS (SELECT 1 FROM dbo.Shippers WHERE
49         ShipperID = @ShipVia)
50     BEGIN
51         SET @ErrorMessage = FORMATMESSAGE('El ShipVia
52         (ShipperID) proporcionado''%d'' no existe.',
53         @ShipVia);
54         THROW 50003, @ErrorMessage, 1;
55     END
56
57     -- 2. Validar stock para cada producto en el TVP
58     DECLARE product_cursor CURSOR LOCAL FAST_FORWARD FOR
59         SELECT ProductID, Quantity FROM @OrderDetailsTVP;
60
61     OPEN product_cursor;
62     FETCH NEXT FROM product_cursor INTO
63         @CurrentProductID, @CurrentQuantity;
64
65     WHILE @@FETCH_STATUS = 0
66     BEGIN
67         SELECT
68             @StockAvailable = p.UnitsInStock,
69             @ProductName = p.ProductName
70         FROM dbo.Products p
71         WHERE p.ProductID = @CurrentProductID;
72
73         IF @StockAvailable IS NULL -- Significa que el
74             ProductID no se encontr en la tabla Products
75         BEGIN
76             CLOSE product_cursor;
77             DEALLOCATE product_cursor;
78             SET @ErrorMessage = FORMATMESSAGE('El
79             producto ID %d especificado en los
80             detalles del pedido no existe en la tabla

```

```

71         Products.', @CurrentProductID);
72     THROW 50004, @ErrorMessage, 1;
73     END
74     IF @StockAvailable < @CurrentQuantity
75     BEGIN
76         CLOSE product_cursor;
77         DEALLOCATE product_cursor;
78         SET @ErrorMessage = FORMATMESSAGE('Stock
insuficiente para el producto "%s" (ID:
%d). Solicitado: %d, Disponible: %d.',
@ProductName, @CurrentProductID,
@CurrentQuantity, @StockAvailable);
79         THROW 50005, @ErrorMessage, 1;
80     END
81
82     FETCH NEXT FROM product_cursor INTO
@CurrentProductID, @CurrentQuantity;
83 END
84
85 CLOSE product_cursor;
86 DEALLOCATE product_cursor;
87
88 -- 3. Insertar en la tabla Orders
89 INSERT INTO dbo.Orders (
90     CustomerID, EmployeeID, OrderDate, RequiredDate,
91     ShippedDate,
92     ShipVia, Freight, ShipName, ShipAddress,
93     ShipCity,
94     ShipRegion, ShipPostalCode, ShipCountry
95 )
96 VALUES (
97     @CustomerID, @EmployeeID, @OrderDate,
98     @RequiredDate, @ShippedDate,
99     @ShipVia, @Freight, @ShipName, @ShipAddress,
100     @ShipCity,
101     @ShipRegion, @ShipPostalCode, @ShipCountry
102 );
103
104 SET @NewOrderID = SCOPE_IDENTITY();
105
106 -- 4. Insertar en la tabla Order Details y
actualizar stock
107 INSERT INTO dbo.[Order Details] (
108     OrderID, ProductID, UnitPrice, Quantity, Discount

```

```

105         )
106     SELECT
107         @NewOrderID,
108         tvp.ProductID,
109         tvp.UnitPrice,
110         tvp.Quantity,
111         tvp.Discount
112     FROM @OrderDetailsTVP tvp;
113
114     UPDATE p
115     SET p.UnitsInStock = p.UnitsInStock - od.Quantity
116     FROM dbo.Products p
117     INNER JOIN @OrderDetailsTVP od ON p.ProductID =
        od.ProductID;
118
119     COMMIT TRANSACTION;
120
121     SELECT @NewOrderID AS NewOrderID;
122     PRINT 'Pedido' + CAST(@NewOrderID AS VARCHAR(10)) +
        'registrado exitosamente.';
123
124     END TRY
125     BEGIN CATCH
126         IF @@TRANCOUNT > 0
127             ROLLBACK TRANSACTION;
128
129         -- Relanzar el error. El mensaje ya fue formateado
130         -- si provino de un THROW nuestro.
131         -- Si es un error del sistema, se relanzar tal
132         -- cual.
133         THROW;
134         -- RETURN; -- No es estrictamente necesario despu s
135         -- de THROW si es la ltima instrucc i n en CATCH
136     END CATCH
137 END
138 GO

```

Listing 3: Procedimiento Almacenado RegistrarNuevoPedido

3.2.2. Pruebas Realizadas

Se llevaron a cabo diversas pruebas para asegurar el correcto funcionamiento del procedimiento almacenado RegistrarNuevoPedido bajo diferentes con-

diciones. A continuación, se detallan los escenarios de prueba más relevantes.

Escenario 1: Registro Exitoso de Pedido con Datos Reales Este escenario valida la capacidad del procedimiento para registrar un nuevo pedido cuando todos los datos de entrada son válidos y hay suficiente stock para todos los productos solicitados.

Datos de Prueba Utilizados: Se seleccionaron datos existentes y válidos de la base de datos Northwind:

- Cliente (@CustomerID): ALFKI
- Empleado (@EmployeeID): 3
- Transportista (@ShipVia): 1
- Fecha de Pedido (@OrderDate): Fecha actual del sistema (GETDATE()).
- Fecha Requerida (@RequiredDate): 14 días a partir de la fecha del pedido.
- Flete (@Freight): 45.75
- Datos de Envío: Correspondientes al cliente ALFKI (Alfreds Futterkiste, Obere Str. 57, Berlin, 12209, Germany).
- Productos en el Pedido (@OrderDetailsTVP):
 - Producto ID 4 (Chef Anton's Cajun Seasoning), Precio: 22.00, Cantidad: 5, Descuento: 0%
 - Producto ID 14 (Tofu), Precio: 23.25, Cantidad: 10, Descuento: 5%
 - Producto ID 33 (Geitost), Precio: 2.50, Cantidad: 20, Descuento: 0%

Se verificó previamente que existiera stock suficiente para las cantidades solicitadas de cada producto.

Script de Ejecución de la Prueba: El siguiente script T-SQL se utilizó para ejecutar la prueba:

```
1  -- Verificar que el Tipo de Tabla OrderDetailType existe
   (condensado para brevedad)
2  IF TYPE_ID(N'OrderDetailType') IS NULL BEGIN THROW 50000,
   'OrderDetailType no existe.', 1; RETURN; END;
3
4  -- Verificar stock ANTES (condensado)
5  PRINT '---StockANTES---';
6  SELECT ProductID, ProductName, UnitsInStock FROM
   dbo.Products WHERE ProductID IN (4, 14, 33);
7
8  -- Declarar y poblar TVP
9  DECLARE @DetallesDelPedidoPrueba OrderDetailType;
10 INSERT INTO @DetallesDelPedidoPrueba (ProductID, UnitPrice,
   Quantity, Discount)
11 VALUES (4, 22.00, 5, 0.0), (14, 23.25, 10, 0.05), (33, 2.50,
   20, 0.0);
12
13 PRINT 'Detalles del pedido a registrar: '; SELECT * FROM
   @DetallesDelPedidoPrueba;
14
15 -- Declarar parametros del SP
16 DECLARE @TestCustomerID_Real NCHAR(5) = 'ALFKI',
   @TestEmployeeID_Real INT = 3,
17         @TestOrderDate_Real DATETIME = GETDATE(),
18         @TestRequiredDate_Real DATETIME = DATEADD(day, 14,
   GETDATE()),
19         @TestShippedDate_Real DATETIME = NULL,
   @TestShipVia_Real INT = 1,
20         @TestFreight_Real MONEY = 45.75,
21         @TestShipName_Real NVARCHAR(40) = 'Alfreds
   Futterkiste',
22         @TestShipAddress_Real NVARCHAR(60) = 'Obere Str. 57',
23         @TestShipCity_Real NVARCHAR(15) = 'Berlin',
24         @TestShipRegion_Real NVARCHAR(15) = NULL,
25         @TestShipPostalCode_Real NVARCHAR(10) = '12209',
26         @TestShipCountry_Real NVARCHAR(15) = 'Germany',
27         @NuevoPedidoID_Resultado INT;
28
29 PRINT '---Ejecutando dbo.RegistrarNuevoPedido---';
30 BEGIN TRY
31     EXEC dbo.RegistrarNuevoPedido
32         @CustomerID = @TestCustomerID_Real, @EmployeeID =
   @TestEmployeeID_Real,
```

```

33      @OrderDate = @TestOrderDate_Real, @RequiredDate =
          @TestRequiredDate_Real,
34      @ShippedDate = @TestShippedDate_Real, @ShipVia =
          @TestShipVia_Real,
35      @Freight = @TestFreight_Real, @ShipName =
          @TestShipName_Real,
36      @ShipAddress = @TestShipAddress_Real, @ShipCity =
          @TestShipCity_Real,
37      @ShipRegion = @TestShipRegion_Real, @ShipPostalCode
          = @TestShipPostalCode_Real,
38      @ShipCountry = @TestShipCountry_Real,
          @OrderDetailsTVP = @DetallesDelPedidoPrueba;
39
40      PRINT 'Procedimiento_RegistrarNuevoPedido_ejecutado.';
41
42      SELECT TOP 1 @NuevoPedidoID_Resultado = OrderID FROM
          dbo.Orders ORDER BY OrderDate DESC, OrderID DESC;
43
44      IF @NuevoPedidoID_Resultado IS NOT NULL BEGIN
45          PRINT '---_Verificaci n_POST-Ejecuci n_Pedido_ID:_
              ' + CAST(@NuevoPedidoID_Resultado AS VARCHAR(10))
              + '_---';
46          PRINT 'Encabezado_(Orders):'; SELECT * FROM
              dbo.Orders WHERE OrderID =
              @NuevoPedidoID_Resultado;
47          PRINT 'Detalles_(Order_Details):'; SELECT * FROM
              dbo.[Order Details] WHERE OrderID =
              @NuevoPedidoID_Resultado;
48          PRINT 'Stock_ACTUALIZADO:'; SELECT ProductID,
              ProductName, UnitsInStock FROM dbo.Products WHERE
              ProductID IN (4, 14, 33);
49      END ELSE BEGIN
50          PRINT 'ADVERTENCIA:_No_se_pudo_determinar_OrderID_
              para_verificaci n.';
51      END
52  END TRY
53  BEGIN CATCH
54      PRINT ' ERROR _EN_PRUEBA!:_ ' + ERROR_MESSAGE();
55  END CATCH
56  GO

```

Listing 4: Prueba de registro exitoso para RegistrarNuevoPedido

Resultados Obtenidos y Verificación: La ejecución del procedimiento fue exitosa.

- Se generó un nuevo `OrderID` ID generado fue 11088.
- Se insertó correctamente un nuevo registro en la tabla `Orders` con los datos del cliente `ALFKI`, el empleado 3, y los detalles de envío y fechas proporcionados.
- Se insertaron tres registros en la tabla `Order Details` asociados al nuevo `OrderID`, correspondiendo a los productos 4, 14 y 33, con las cantidades, precios y descuentos especificados.
- El stock (`UnitsInStock`) de los productos fue actualizado correctamente:
 - Producto ID 4: Reducido en 5 unidades.
 - Producto ID 14: Reducido en 10 unidades.
 - Producto ID 33: Reducido en 20 unidades.
- La transacción se completó con `COMMIT`.

La siguiente figura muestra una captura de pantalla de los resultados obtenidos tras la ejecución del script de prueba en SQL Server Management Studio (SSMS), evidenciando la creación del pedido y la actualización del inventario.

	ProductID	Quantity	UnitPrice	Discount
1	4	5	22.00	0
2	14	10	23.25	0.05
3	33	20	2.50	0

	OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight	ShipName	ShipAddress	ShipCity	ShipRegion	ShipPostalCode	ShipCountry
1	11088	ALFKI	3	2025-05-09 22:03:07.183	2025-05-23 22:03:07.183	NULL	1	45.75	Alfreds Futterkiste	Obere Str. 57	Berlin	NULL	12209	Germany

	OrderID	ProductID	UnitPrice	Quantity	Discount
1	11088	4	22.00	5	0
2	11088	14	23.25	10	0.05
3	11088	33	2.50	20	0

	ProductID	ProductName	UnitsInStock
1	4	Chef Anton's Cajun Seasoning	48
2	14	Tofu	25
3	33	Gelbst	92

Figura 1: Resultados de la prueba de registro exitoso del SP `RegistrarNuevoPedido`.

Escenario 2: Stock Insuficiente Este escenario prueba la respuesta del procedimiento cuando se intenta registrar un pedido que incluye un producto cuya cantidad solicitada excede las unidades disponibles en stock. Se espera que el procedimiento lance un error específico, no cree el pedido y revierta cualquier cambio parcial gracias a la transacción.

Datos de Prueba Utilizados:

- Producto con Stock Insuficiente:
 - Producto ID: 74 (Longlife Tofu)
 - Stock Actual Verificado: 4 unidades.
 - Cantidad Solicitada: 9 unidades (5 más que el stock disponible).
 - Precio Unitario: 10.00
 - Descuento: 0%
- Cliente (@CustomerID): BOTTM
- Empleado (@EmployeeID): 4
- Otros datos del encabezado del pedido: Se utilizaron valores de ejemplo para completar los campos requeridos.

Script de Ejecución de la Prueba: El siguiente script T-SQL, diseñado para ser limpio y directo, se utilizó para simular esta condición:

```
1 -----
2 -- Prueba Limpia: STOCK INSUFICIENTE para
   dbo.RegistrarNuevoPedido
3 -----
4
5 -- Paso 0: Asegurar que OrderDetailType existe
6 IF TYPE_ID(N'OrderDetailType') IS NULL
7 BEGIN
8     PRINT 'OrderDetailType NO existe. Cre ndolo...';
9     CREATE TYPE OrderDetailType AS TABLE (
10         ProductID INT NOT NULL, UnitPrice MONEY NOT NULL,
11         Quantity SMALLINT NOT NULL,
```



```

12         Discount REAL NOT NULL CHECK (Discount >= 0 AND
13             Discount <= 1)
14     );
15     PRINT 'OrderDetailType creado.';
16 END
17 ELSE BEGIN PRINT 'OrderDetailType ya existe.'; END
18 GO
19
20 -----
21 -- Inicio del Lote de Prueba Principal
22 -----
23 PRINT '---INICIO PRUEBA: STOCK INSUFICIENTE---';
24
25 -- Paso 1: Variables y preparaci n de datos para el TVP
26 DECLARE @TVP_StockInsuficiente OrderDetailType;
27 DECLARE @ProductID_Test INT = 74; -- Longlife Tofu
28 DECLARE @UnitPrice_Test MONEY = 10.00;
29 DECLARE @StockActual_Test SMALLINT;
30 DECLARE @CantidadPedir_Test SMALLINT;
31 DECLARE @ProductName_Test NVARCHAR(40);
32
33 SELECT @StockActual_Test = UnitsInStock, @ProductName_Test =
34     ProductName
35 FROM dbo.Products WHERE ProductID = @ProductID_Test;
36
37 IF @StockActual_Test IS NULL BEGIN
38     PRINT 'Error Cr tico: Producto ID ' +
39         CAST(@ProductID_Test AS VARCHAR) + ' no encontrado.';
40     RETURN;
41 END
42 ELSE BEGIN
43     PRINT 'Info: Stock actual del Producto ' +
44         CAST(@ProductID_Test AS VARCHAR) +
45         ' (' + ISNULL(@ProductName_Test, 'N/A') + ') es: ' +
46         CAST(@StockActual_Test AS VARCHAR);
47 END
48
49 IF @StockActual_Test IS NOT NULL BEGIN
50     SET @CantidadPedir_Test = @StockActual_Test + 5;
51     PRINT 'Info: Se intentar pedir ' +
52         CAST(@CantidadPedir_Test AS VARCHAR) + ' unidades.';
53     INSERT INTO @TVP_StockInsuficiente (ProductID,
54         UnitPrice, Quantity, Discount)
55     VALUES (@ProductID_Test, @UnitPrice_Test,
56         @CantidadPedir_Test, 0.0);

```

```

49 END ELSE BEGIN
50     PRINT 'Error_Fatal: No se puede probar stock
           insuficiente si el producto no existe.';
51     RETURN;
52 END
53
54 -- Paso 2: Variables para los par metros del encabezado del
    pedido
55 PRINT 'Info: Preparando par metros para el SP...';
56 DECLARE @TestOrderDate DATETIME = GETDATE();
57 DECLARE @TestRequiredDate DATETIME = DATEADD(day, 7,
    GETDATE());
58
59 -- Paso 3: Ejecutar el procedimiento almacenado
60 PRINT 'Llamando a dbo.RegistrarNuevoPedido para probar stock
    insuficiente...';
61 BEGIN TRY
62     EXECUTE dbo.RegistrarNuevoPedido
63         @CustomerID      = 'BOTTM', @EmployeeID      = 4,
64         @OrderDate        = @TestOrderDate, @RequiredDate =
            @TestRequiredDate,
65         @ShippedDate      = NULL, @ShipVia            = 1,
66         @Freight           = 15.00, @ShipName          = 'Test
            Stock Insuficiente',
67         @ShipAddress      = '123 Error Lane', @ShipCity
            = 'Testville',
68         @ShipRegion       = NULL, @ShipPostalCode = NULL,
69         @ShipCountry       = 'Testland', @OrderDetailsTVP =
            @TVP_StockInsuficiente;
70
71     PRINT ' FALLO DE LA PRUEBA! El SP se ejecutó sin
            indicar stock insuficiente.';
72 END TRY
73 BEGIN CATCH
74     PRINT '--- ERROR CAPTURADO (Resultado Esperado por Stock
            Insuficiente) ---';
75     PRINT 'Error N mero: ' + CAST(ERROR_NUMBER() AS
            VARCHAR(10));
76     PRINT 'Mensaje: ' + ERROR_MESSAGE();
77
78     IF ERROR_NUMBER() = 50005 BEGIN
79         PRINT 'VERIFICACION: Error 50005 (Stock
            Insuficiente) capturado correctamente!';
80     END ELSE BEGIN
81         PRINT 'ADVERTENCIA: Se capturó un error, pero NO es

```

```

82         el_esperado_50005.';
83     END
84     DECLARE @StockDespues_Test SMALLINT;
85     SELECT @StockDespues_Test = UnitsInStock FROM
86     dbo.Products WHERE ProductID = @ProductID_Test;
87     PRINT 'Info: Stock del Producto ID ' +
88     CAST(@ProductID_Test AS VARCHAR) +
89     ' DESPU S: ' + ISNULL(CAST(@StockDespues_Test AS
90     VARCHAR), 'N/A');
91     IF @StockActual_Test = @StockDespues_Test BEGIN
92         PRINT 'VERIFICACI N: El stock del producto no
93         cambi . Correcto !';
94     END ELSE BEGIN
95         PRINT 'VERIFICACI N FALLIDA: El stock del
96         producto CAMBI !';
97     END
98     END
99     END CATCH
100     PRINT '---FIN PRUEBA: STOCK INSUFICIENTE---';
101     GO

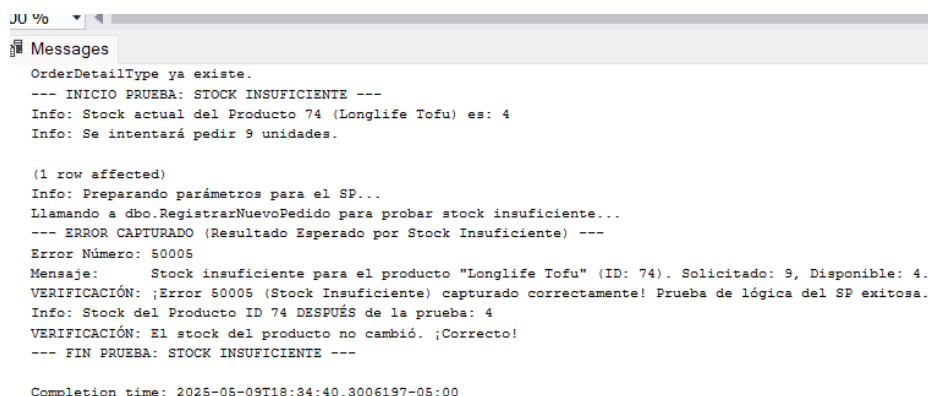
```

Listing 5: Prueba de stock insuficiente para RegistrarNuevoPedido

Resultados Obtenidos y Verificación: La ejecución del procedimiento en este escenario produjo el comportamiento esperado:

- El procedimiento almacenado RegistrarNuevoPedido detectó que la cantidad solicitada para el Producto ID 74 (9 unidades) excedía el stock disponible (4 unidades).
- Se lanzó un error con el número 50005.
- El mensaje de error devuelto fue: Stock insuficiente para el producto "Longlife Tofu"(ID: 74). Solicitado: 9, Disponible: 4.
- No se insertó ningún nuevo registro en la tabla Orders ni en la tabla Order Details.
- La transacción fue revertida (ROLLBACK), por lo que el stock del Producto ID 74 no se modificó y permaneció en 4 unidades.

Este resultado confirma que el manejo de errores para stock insuficiente y la atomicidad de la transacción funcionan correctamente. La siguiente figura muestra una captura de pantalla de los mensajes de error y verificaciones obtenidos.



```

JU %
Messages
OrderDetailType ya existe.
--- INICIO PRUEBA: STOCK INSUFICIENTE ---
Info: Stock actual del Producto 74 (Longlife Tofu) es: 4
Info: Se intentará pedir 9 unidades.

(1 row affected)
Info: Preparando parámetros para el SP...
Llamando a dbo.RegistrarNuevoPedido para probar stock insuficiente...
--- ERROR CAPTURADO (Resultado Esperado por Stock Insuficiente) ---
Error Número: 50005
Mensaje:      Stock insuficiente para el producto "Longlife Tofu" (ID: 74). Solicitado: 9, Disponible: 4.
VERIFICACIÓN: ;Error 50005 (Stock Insuficiente) capturado correctamente! Prueba de lógica del SP exitosa.
Info: Stock del Producto ID 74 DESPUÉS de la prueba: 4
VERIFICACIÓN: El stock del producto no cambió. ;Correcto!
--- FIN PRUEBA: STOCK INSUFICIENTE ---

Completion time: 2025-05-09T18:34:40.3006197-05:00

```

Figura 2: Resultados de la prueba de stock insuficiente para el SP RegistrarNuevoPedido.

3.3. Procedimiento Almacenado: ActualizarEstadoPedido (1.2)

3.3.1. Explicación

Propósito: El procedimiento almacenado `ActualizarEstadoPedido` tiene como objetivo permitir la modificación del estado de envío de un pedido existente. Específicamente, permite actualizar la fecha de envío (`ShippedDate`) y el transportista (`ShipVia`) asociado a un pedido.

Restricción de Negocio Principal: Una regla de negocio fundamental implementada en este procedimiento es que las actualizaciones solo se permiten si el pedido aún no ha sido marcado como enviado. Si el campo `Orders.ShippedDate` ya contiene una fecha, el procedimiento impedirá cualquier modificación y notificará el error.

Parámetros de Entrada:

- **@OrderID (INT):** El identificador único del pedido que se desea actualizar.
- **@NewShippedDate (DATETIME):** La nueva fecha en la que el pedido fue (o será) enviado. Si se proporciona, no puede ser anterior a la fecha original del pedido (**OrderDate**). Se puede pasar NULL si, por alguna razón de negocio, se quisiera revertir un envío (aunque el foco principal es marcar como enviado).
- **@NewShipVia (INT):** El identificador del nuevo transportista que se encargará del envío. Debe corresponder a un **ShipperID** válido en la tabla **Shippers**.

Lógica de Negocio y Operaciones:

1. **Validación de Existencia del Pedido:** Se verifica que el **@OrderID** proporcionado corresponda a un pedido existente en la tabla **Orders**.
2. **Verificación de Estado de Envío Actual:** Se consulta el valor actual de **ShippedDate** para el pedido. Si este campo no es NULL, significa que el pedido ya ha sido enviado, y el procedimiento aborta la operación lanzando un error.
3. **Validación del Transportista:** Se comprueba que el **@NewShipVia** proporcionado exista en la tabla **Shippers**.
4. **Validación de la Nueva Fecha de Envío:** Si se establece una **@NewShippedDate**, se valida que esta no sea anterior a la **OrderDate** del pedido para mantener la coherencia cronológica.
5. **Actualización del Pedido:** Si todas las validaciones previas son superadas, se procede a actualizar los campos **ShippedDate** y **ShipVia** en la tabla **Orders** para el **@OrderID** especificado.

Transacciones y Manejo de Errores: La operación completa se ejecuta dentro de una transacción explícita para garantizar la atomicidad. Si alguna validación falla o se produce un error durante la actualización, la transacción se revierte (ROLLBACK TRANSACTION) y se lanza una excepción utilizando THROW con un mensaje y código de error descriptivo. Esto asegura que la base de datos se mantenga en un estado consistente.

Salida: Si la actualización es exitosa, se muestra un mensaje de confirmación. En caso de error, se lanza una excepción detallando la causa del fallo.

Código SQL del Procedimiento:

```

1  --
2  -- =====
3  -- SP Nombre:      ActualizarEstadoPedido
4  -- Descripci n:    Permite actualizar ShippedDate y
5  --                  ShipVia de un pedido, solo si
6  --                  el pedido no ha sido enviado previamente.
7  -- =====
8
9  CREATE OR ALTER PROCEDURE dbo.ActualizarEstadoPedido
10     @OrderID INT,
11     @NewShippedDate DATETIME,
12     @NewShipVia INT
13 AS
14 BEGIN
15     SET NOCOUNT ON;
16
17     DECLARE @ErrorMessage NVARCHAR(2048);
18     DECLARE @CurrentShippedDate DATETIME;
19     DECLARE @OrderDate DATETIME;
20
21     BEGIN TRANSACTION;
22     BEGIN TRY
23         SELECT @CurrentShippedDate = o.ShippedDate,
24                @OrderDate = o.OrderDate
25         FROM dbo.Orders o WHERE o.OrderID = @OrderID;
26
27         IF @OrderDate IS NULL
28             THROW 50010, 'El pedido con OrderID %d no existe.', 1;
29
30         IF @CurrentShippedDate IS NOT NULL

```

```

27         THROW 50011, 'El pedido OrderID %d ya fue
           enviado y no puede ser modificado.', 1;
28
29     IF NOT EXISTS (SELECT 1 FROM dbo.Shippers WHERE
30         ShipperID = @NewShipVia)
31         THROW 50012, 'El transportista (ShipVia) con ID
           %d no existe.', 1;
32
33     IF @NewShippedDate IS NOT NULL AND @NewShippedDate <
34         @OrderDate
35         THROW 50013, 'La nueva fecha de env o no puede
           ser anterior a la fecha del pedido.', 1;
36
37     UPDATE dbo.Orders
38     SET ShippedDate = @NewShippedDate, ShipVia =
39         @NewShipVia
40     WHERE OrderID = @OrderID;
41
42     IF @@ROWCOUNT = 0
43         THROW 50014, 'No se pudo actualizar el pedido
           OrderID %d.', 1;
44
45     PRINT 'Estado del pedido OrderID' + CAST(@OrderID
46         AS VARCHAR(10)) + ' actualizado.';
47     COMMIT TRANSACTION;
48 END TRY
49 BEGIN CATCH
50     IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;
51     THROW;
52 END CATCH
53 END
54 GO

```

Listing 6: Procedimiento Almacenado ActualizarEstadoPedido

3.3.2. Pruebas Realizadas

Se realizaron las siguientes pruebas para validar el funcionamiento del procedimiento ActualizarEstadoPedido:

Escenario 1: Actualización Exitosa de un Pedido No Enviado

3.3.3. Pruebas Realizadas

Se realizaron las siguientes pruebas para validar el funcionamiento del procedimiento `ActualizarEstadoPedido`:

Escenario 1: Actualización Exitosa de un Pedido No Enviado Este escenario verifica que el procedimiento puede actualizar correctamente la fecha de envío y el transportista de un pedido que aún no ha sido enviado.

Datos de Prueba Utilizados:

- OrderID a actualizar: 11074 (Verificado previamente que `ShippedDate` es NULL).
- OrderDate original del pedido: 1998-05-06.
- Nueva Fecha de Envío (@NewShippedDate): 2025-05-09 00:00:00.000.
- Nuevo Transportista (@NewShipVia): 2 (United Package).

Script de Ejecución de la Prueba:

```
1 -----
2 -- Prueba Exitosa: ActualizarEstadoPedido para un pedido NO
   enviado
3 -- OrderID: 11074
4 -----
5 SET NOCOUNT OFF; -- Asegurar que los PRINTs del script se
   muestren
6 PRINT '---INICIO_PRUEBA: Actualizaci n Exitosa de Pedido
   No Enviado---';
7
8 DECLARE @TestOrderID_Success INT = 11074;
9 DECLARE @NewShippedDate_Success DATETIME = '2025-05-09
   00:00:00.000';
10 DECLARE @NewShipVia_Success INT = 2;
11 DECLARE @OrderDate_Before DATETIME, @ShippedDate_Before
   DATETIME, @ShipVia_Before INT;
12 DECLARE @ShippedDate_After DATETIME, @ShipVia_After INT;
13
14 PRINT '---Estado del Pedido ANTES (OrderID: ' +
   CAST(@TestOrderID_Success AS VARCHAR) + ')---';
```



```

15 SELECT @OrderDate_Before = OrderDate, @ShippedDate_Before =
    ShippedDate, @ShipVia_Before = ShipVia
16 FROM dbo.Orders WHERE OrderID = @TestOrderID_Success;
17 -- Mostrar datos en Resultados (opcional para
    documentaci n, pero til en ejecuci n)
18 -- SELECT OrderID, CustomerID, OrderDate, ShippedDate,
    ShipVia FROM dbo.Orders WHERE OrderID =
    @TestOrderID_Success;
19
20 IF @OrderDate_Before IS NULL BEGIN PRINT 'Error:␣OrderID␣no␣
    encontrado.'; RETURN; END
21 IF @ShippedDate_Before IS NOT NULL BEGIN PRINT 'Error:␣
    Pedido␣YA␣enviado.'; RETURN; END
22
23 PRINT 'Info␣ANTES:␣ShippedDate=' + ISNULL(CONVERT(VARCHAR,
    @ShippedDate_Before, 121), 'NULL') +
24     ',␣ShipVia=' + ISNULL(CAST(@ShipVia_Before AS
    VARCHAR), 'NULL');
25
26 PRINT 'Llamando␣a␣dbo.ActualizarEstadoPedido...';
27 BEGIN TRY
28     EXECUTE dbo.ActualizarEstadoPedido
29         @OrderID = @TestOrderID_Success,
30         @NewShippedDate = @NewShippedDate_Success,
31         @NewShipVia = @NewShipVia_Success;
32
33     PRINT 'Procedimiento␣ActualizarEstadoPedido␣ejecutado.';
34
35     PRINT '---␣Estado␣del␣Pedido␣DESPU S␣(OrderID:␣' +
        CAST(@TestOrderID_Success AS VARCHAR) + ')␣---';
36     SELECT @ShippedDate_After = ShippedDate, @ShipVia_After
        = ShipVia
37     FROM dbo.Orders WHERE OrderID = @TestOrderID_Success;
38     -- Mostrar datos en Resultados (opcional para
        documentaci n)
39     -- SELECT OrderID, CustomerID, OrderDate, ShippedDate,
        ShipVia FROM dbo.Orders WHERE OrderID =
        @TestOrderID_Success;
40
41     PRINT 'Info␣DESPU S:␣ShippedDate=' +
        ISNULL(CONVERT(VARCHAR, @ShippedDate_After,
        121), 'NULL') +
42         ',␣ShipVia=' + ISNULL(CAST(@ShipVia_After AS
        VARCHAR), 'NULL');
43

```

```

44      IF @ShippedDate_After IS NOT NULL AND CONVERT(DATE,
              @ShippedDate_After) = CONVERT(DATE,
              @NewShippedDate_Success)
45          PRINT 'VERIFICACI N ShippedDate: Correcto !';
46      ELSE PRINT 'FALLO VERIFICACI N ShippedDate.';
47
48      IF @ShipVia_After = @NewShipVia_Success
49          PRINT 'VERIFICACI N ShipVia: Correcto !';
50      ELSE PRINT 'FALLO VERIFICACI N ShipVia.';
51  END TRY
52  BEGIN CATCH
53      PRINT '--- ERROR CAPTURADO ---';
54      PRINT 'Error: ' + ERROR_MESSAGE();
55  END CATCH
56  PRINT '--- FIN PRUEBA ---';
57  GO

```

Listing 7: Prueba de actualización exitosa para ActualizarEstadoPedido

Resultados Obtenidos y Verificación: La ejecución del script de prueba y del procedimiento almacenado fue exitosa. Los mensajes clave observados en la pestaña "Messages" fueron:

- Verificación del estado del pedido ANTES: Info ANTES: OrderDate=1998-05-06 00:00:00.000, ShippedDate=NULL, ShipVia=2
- Mensaje del SP tras la ejecución: Estado del pedido OrderID 11074 actualizado exitosamente.
- Verificación del estado del pedido DESPUÉS: Info DESPUÉS: ShippedDate=2025-05-09 00:00:00.000, ShipVia=2
- Confirmaciones de verificación: VERIFICACIÓN ShippedDate: ¡Correcto! y VERIFICACIÓN ShipVia: ¡Correcto!

Adicionalmente, la consulta a la tabla `Orders` después de la ejecución mostró que el campo `ShippedDate` para el `OrderID 11074` fue actualizado a 2025-05-09 00:00:00.000 y el campo `ShipVia` se mantuvo en 2, según lo esperado. No se produjeron errores.

	OrderID	CustomerID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight	ShipName
1	11074	SIMOB	1998-05-06 00:00:00.000	1998-06-03 00:00:00.000	NULL	2	18.44	Simons bistro

	OrderID	CustomerID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight	ShipName
1	11074	SIMOB	1998-05-06 00:00:00.000	1998-06-03 00:00:00.000	2025-05-09 00:00:00.000	2	18.44	Simons bistro

Figura 3: Resultados de la prueba exitosa para ActualizarEstadoPedido.

Escenario 2: Intento de Actualizar un Pedido Ya Enviado Este escenario tiene como objetivo verificar que el procedimiento impide la modificación de un pedido que ya ha sido marcado como enviado (es decir, su campo `ShippedDate` no es NULL). Se espera que el SP lance un error específico y no realice ningún cambio.

Datos de Prueba Utilizados:

- OrderID a intentar actualizar: 11067.
- ShippedDate actual del pedido: 1998-05-06 00:00:00.000 (Fecha existente, confirmando que ya fue enviado).
- Valores de intento para nuevos datos (no deberían aplicarse):
 - @AttemptNewShippedDate: GETDATE() (Fecha actual del sistema).
 - @AttemptNewShipVia: 1.

Script de Ejecución de la Prueba:

```

1 -----
2 -- Prueba Error: ActualizarEstadoPedido para un pedido YA
   enviado
3 -- OrderID: 11067
4 -----
5 PRINT '---INICIO_PRUEBA: Intento de Actualizar Pedido YA
   Enviado---';
6
7 DECLARE @TestOrderID_AlreadyShipped INT = 11067;
8 DECLARE @AttemptNewShippedDate DATETIME = GETDATE();
9 DECLARE @AttemptNewShipVia INT = 1;

```

```

10 DECLARE @ShippedDate_Before_AS DATETIME, @ShipVia_Before_AS
    INT;
11 DECLARE @ShippedDate_After_AS DATETIME, @ShipVia_After_AS
    INT;
12
13 PRINT '--Estado ANTES (OrderID: ' +
    CAST(@TestOrderID_AlreadyShipped AS VARCHAR) + ')---';
14 SELECT @ShippedDate_Before_AS = ShippedDate,
    @ShipVia_Before_AS = ShipVia
15 FROM dbo.Orders WHERE OrderID = @TestOrderID_AlreadyShipped;
16 -- SELECT OrderID, ShippedDate, ShipVia FROM dbo.Orders
    WHERE OrderID = @TestOrderID_AlreadyShipped; -- Para
    Results
17
18 IF @ShippedDate_Before_AS IS NULL BEGIN
19     PRINT 'Error Config Prueba: Pedido NO enviado. Se
        requiere YA enviado.'; RETURN;
20 END
21 PRINT 'Info ANTES: ShippedDate=' + ISNULL(CONVERT(VARCHAR,
    @ShippedDate_Before_AS, 121), 'NULL') +
22     ', ShipVia=' + ISNULL(CAST(@ShipVia_Before_AS AS
        VARCHAR), 'NULL');
23
24 PRINT 'Llamando a dbo.ActualizarEstadoPedido (esperando
    error)...';
25 BEGIN TRY
26     EXECUTE dbo.ActualizarEstadoPedido
27         @OrderID = @TestOrderID_AlreadyShipped,
28         @NewShippedDate = @AttemptNewShippedDate,
29         @NewShipVia = @AttemptNewShipVia;
30     PRINT ' FALLO DE PRUEBA! SP no lanz error esperado.';
31 END TRY
32 BEGIN CATCH
33     PRINT '--ERROR CAPTURADO (Resultado Esperado)---';
34     PRINT 'Error N mero: ' + CAST(ERROR_NUMBER() AS
        VARCHAR(10));
35     PRINT 'Mensaje: ' + ERROR_MESSAGE();
36     IF ERROR_NUMBER() = 50011 BEGIN
37         PRINT 'VERIFICACION: Error 50011 capturado
            correctamente!';
38     END ELSE BEGIN
39         PRINT 'ADVERTENCIA: Error capturado NO es el
            esperado 50011.';
40     END
41

```

```

42 PRINT '---Estado DESPU S_(OrderID:_' +
      CAST(@TestOrderID_AlreadyShipped AS VARCHAR) + '))_
      ---';
43 SELECT @ShippedDate_After_AS = ShippedDate,
      @ShipVia_After_AS = ShipVia
44 FROM dbo.Orders WHERE OrderID =
      @TestOrderID_AlreadyShipped;
45 -- SELECT OrderID, ShippedDate, ShipVia FROM dbo.Orders
      WHERE OrderID = @TestOrderID_AlreadyShipped; -- Para
      Results
46
47 PRINT 'Info DESPU S:_ShippedDate=' +
      ISNULL(CONVERT(VARCHAR, @ShippedDate_After_AS,
      121), 'NULL') +
48      ',_ShipVia=' + ISNULL(CAST(@ShipVia_After_AS AS
      VARCHAR), 'NULL');
49
50 IF @ShippedDate_After_AS = @ShippedDate_Before_AS
51     PRINT 'VERIFICACI N_ShippedDate:_NO_cambi ._
      Correcto !';
52 ELSE PRINT 'FALLO VERIFICACI N_ShippedDate:_ CAMBI _
      INCORRECTAMENTE!';
53 IF @ShipVia_After_AS = @ShipVia_Before_AS
54     PRINT 'VERIFICACI N_ShipVia:_NO_cambi ._
      Correcto !';
55 ELSE PRINT 'FALLO VERIFICACI N_ShipVia:_ CAMBI _
      INCORRECTAMENTE!';
56 END CATCH
57 PRINT '---FIN PRUEBA---';
58 GO

```

Listing 8: Prueba de intento de actualizar pedido ya enviado

Resultados Obtenidos y Verificación: La ejecución del script de prueba confirmó el correcto funcionamiento de la restricción del procedimiento:

- El estado inicial del pedido 11067 mostró una ShippedDate de 1998-05-06 00:00:00.000.
- Al intentar ejecutar ActualizarEstadoPedido, el procedimiento lanzó un error.

- El error capturado fue el número 50011 con el mensaje: El pedido OrderID 11067 ya fue enviado el 06/05/1998 y no puede ser modificado.
- Las verificaciones posteriores confirmaron que los campos **ShippedDate** y **ShipVia** del pedido 11067 no sufrieron ninguna alteración, permaneciendo idénticos a su estado original.
- Esto demuestra que la transacción se revirtió correctamente (o no se llegó a la etapa de actualización) y la lógica de negocio que impide modificar pedidos ya enviados está operativa.

	OrderID	CustomerID	OrderDate	RequiredDate	ShippedDate	ShipVia
1	11067	DRACD	1998-05-04 00:00:00.000	1998-05-18 00:00:00.000	1998-05-06 00:00:00.000	2

	OrderID	CustomerID	OrderDate	RequiredDate	ShippedDate	ShipVia
1	11067	DRACD	1998-05-04 00:00:00.000	1998-05-18 00:00:00.000	1998-05-06 00:00:00.000	2

Figura 4: Resultados de la prueba de intento de actualizar pedido ya enviado para **ActualizarEstadoPedido**.

Escenario 3: Intento de Actualizar con un OrderID Inexistente Esta prueba verifica el comportamiento del procedimiento cuando se le proporciona un **OrderID** que no corresponde a ningún pedido existente en la tabla **Orders**. El procedimiento debe detectar esta situación y generar un error, sin realizar ninguna otra acción.

Datos de Prueba Utilizados:

- OrderID a intentar actualizar: 12000 (Verificado previamente que este ID no existe en la tabla **Orders**).
- Valores de intento para nuevos datos (no deberían ser procesados):
 - @AttemptNewShippedDate_NE: GETDATE() (Fecha actual del sistema).
 - @AttemptNewShipVia_NE: 1.

Script de Ejecución de la Prueba:

```

1  -----
2  -- Prueba Error: ActualizarEstadoPedido con un OrderID
   INEXISTENTE
3  -- OrderID de prueba: 12000
4  -----
5  PRINT '---INICIO_PRUEBA:Intento de Actualizar Pedido con
   OrderID_Inexistente---';
6
7  DECLARE @TestOrderID_NonExistent INT = 12000;
8  DECLARE @AttemptNewShippedDate_NE DATETIME = GETDATE();
9  DECLARE @AttemptNewShipVia_NE INT = 1;
10
11 IF EXISTS (SELECT 1 FROM dbo.Orders WHERE OrderID =
   @TestOrderID_NonExistent) BEGIN
12     PRINT 'Error_Config_Prueba:OrderID' +
   CAST(@TestOrderID_NonExistent AS VARCHAR) + 'S '
   EXISTE.';
13     RETURN;
14 END ELSE BEGIN
15     PRINT 'Info:ConfirmadoOrderID' +
   CAST(@TestOrderID_NonExistent AS VARCHAR) + 'no
   existe.';
16 END
17
18 PRINT 'Llamando a dbo.ActualizarEstadoPedido con OrderID' +
   CAST(@TestOrderID_NonExistent AS VARCHAR) + '...';
19 BEGIN TRY
20     EXECUTE dbo.ActualizarEstadoPedido
21         @OrderID = @TestOrderID_NonExistent,
22         @NewShippedDate = @AttemptNewShippedDate_NE,
23         @NewShipVia = @AttemptNewShipVia_NE;
24     PRINT ' FALLO DE LA PRUEBA!SP no lanz error por
   OrderID_inexistente.';
25 END TRY
26 BEGIN CATCH
27     PRINT '---ERROR CAPTURADO(Resultado Esperado)---';
28     PRINT 'Error_N mero:' + CAST(ERROR_NUMBER() AS
   VARCHAR(10));
29     PRINT 'Mensaje: ' + ERROR_MESSAGE();
30     IF ERROR_NUMBER() = 50010 BEGIN
31         PRINT 'VERIFICACION: Error 50010("El pedido...
   no existe.") capturado!';
32     END ELSE BEGIN

```

```

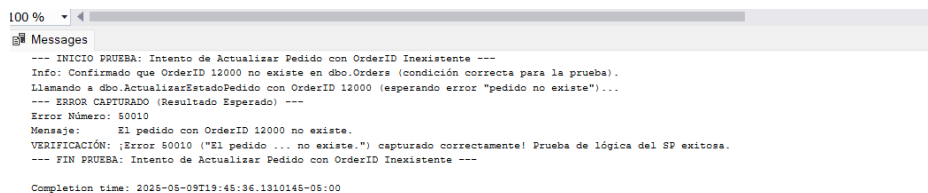
33         PRINT 'ADVERTENCIA: Error capturado NO es el
              esperado 50010.';
34     END
35 END CATCH
36 PRINT '---FIN PRUEBA---';
37 GO

```

Listing 9: Prueba de intento de actualizar pedido con OrderID inexistente

Resultados Obtenidos y Verificación: La ejecución de la prueba arrojó los siguientes resultados, confirmando el manejo adecuado de OrderID inexistentes:

- Se confirmó inicialmente que el OrderID 12000 no existía en la base de datos.
- Al ejecutar el procedimiento ActualizarEstadoPedido con este OrderID, se capturó un error.
- El error devuelto fue el número 50010, con el mensaje: El pedido con OrderID 12000 no existe.
- Esto valida que el procedimiento identifica correctamente los pedidos no existentes y finaliza su ejecución de manera controlada, sin intentar realizar actualizaciones sobre datos que no existen.



```

100 %
Messages
--- INICIO PRUEBA: Intento de Actualizar Pedido con OrderID Inexistente ---
Info: Confirmado que OrderID 12000 no existe en dbo.Orders (condición correcta para la prueba).
Llamando a dbo.ActualizarEstadoPedido con OrderID 12000 (esperando error "pedido no existe")...
--- ERROR CAPTURADO (Resultado Esperado) ---
Error Número: 50010
Mensaje: El pedido con OrderID 12000 no existe.
VERIFICACIÓN: Error 50010 ("El pedido ... no existe.") capturado correctamente! Prueba de lógica del SP exitosa.
--- FIN PRUEBA: Intento de Actualizar Pedido con OrderID Inexistente ---
Completion time: 2025-05-09T19:45:36.1310145-05:00

```

Figura 5: Resultados de la prueba de OrderID inexistente para ActualizarEstadoPedido.

Escenario 4: Intento de Actualizar con un ShipVia Inexistente Esta prueba evalúa la validación del parámetro @NewShipVia. Se intenta actualizar un pedido no enviado utilizando un identificador de transportista que

no existe en la tabla Shippers. El procedimiento debe rechazar la operación y emitir un error.

Datos de Prueba Utilizados:

- OrderID a intentar actualizar: 11008 (Verificado previamente que es un pedido existente y su ShippedDate es NULL).
- @NewShippedDate: GETDATE() (Fecha actual, valor de placeholder).
- @NewShipVia: 10 (Identificador de transportista que se sabe no existe en la tabla Shippers).

Script de Ejecución de la Prueba:

```
1 -----
2 -- Prueba Error: ActualizarEstadoPedido con un ShipVia
3 -- INEXISTENTE
4 -- OrderID: 11008, NewShipVia: 10 (inexistente)
5 -----
6 PRINT '---INICIO_PRUEBA:ShipViaInexistente---';
7
8 DECLARE @TestOrderID_SVE INT = 11008;
9 DECLARE @AttemptNewShippedDate_SVE DATETIME = GETDATE();
10 DECLARE @AttemptNewShipVia_SVE INT = 10; -- ShipVia
11 -- inexistente
12
13 -- Confirmar que el OrderID es v lido y no enviado
14 IF NOT EXISTS (SELECT 1 FROM dbo.Orders WHERE OrderID =
15 @TestOrderID_SVE AND ShippedDate IS NULL) BEGIN
16     PRINT 'Error_ConfigPrueba:OrderID' +
17     CAST(@TestOrderID_SVE AS VARCHAR) +
18     'no es v lido o ya fue enviado.'; RETURN;
19 END ELSE BEGIN
20     PRINT 'Info:OrderID' + CAST(@TestOrderID_SVE AS
21     VARCHAR) + 'es v lido y no enviado.';
22 END
23
24 -- Confirmar que el ShipVia no existe
25 IF EXISTS (SELECT 1 FROM dbo.Shippers WHERE ShipperID =
26 @AttemptNewShipVia_SVE) BEGIN
27     PRINT 'Error_ConfigPrueba:ShipVia' +
28     CAST(@AttemptNewShipVia_SVE AS VARCHAR) + 'S
29     EXISTE.'; RETURN;
```

```

22 END ELSE BEGIN
23     PRINT 'Info: Confirmado ShipVia' +
        CAST(@AttemptNewShipVia_SVE AS VARCHAR) + 'no
        existe.';
24 END
25
26 PRINT 'Llamando a dbo.ActualizarEstadoPedido con ShipVia' +
    CAST(@AttemptNewShipVia_SVE AS VARCHAR) + '...';
27 BEGIN TRY
28     EXECUTE dbo.ActualizarEstadoPedido
29         @OrderID = @TestOrderID_SVE,
30         @NewShippedDate = @AttemptNewShippedDate_SVE,
31         @NewShipVia = @AttemptNewShipVia_SVE;
32     PRINT ' FALLO DE LA PRUEBA! SP no lanz error por
        ShipVia inexistente.';
33 END TRY
34 BEGIN CATCH
35     PRINT '--- ERROR CAPTURADO (Resultado Esperado) ---';
36     PRINT 'Error N mero: ' + CAST(ERROR_NUMBER() AS
        VARCHAR(10));
37     PRINT 'Mensaje: ' + ERROR_MESSAGE();
38     IF ERROR_NUMBER() = 50012 BEGIN -- C digo de error para
        ShipVia inexistente
39         PRINT 'VERIFICACION: Error 50012 ("El
            transportista...no existe.") capturado!';
40     END ELSE BEGIN
41         PRINT 'ADVERTENCIA: Error capturado NO es el
            esperado 50012.';
42     END
43     -- (Aquí se podrán añadir verificaciones de que el
        pedido no cambie)
44 END CATCH
45 PRINT '--- FIN PRUEBA ---';
46 GO

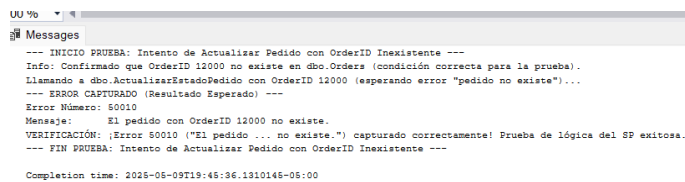
```

Listing 10: Prueba de intento de actualizar pedido con ShipVia inexistente

Resultados Obtenidos y Verificación: La prueba demostró que el procedimiento maneja correctamente los identificadores de transportista no válidos:

- Se confirmó el estado del pedido 11008 (no enviado) y la inexistencia del ShipperID 10.

- Al llamar a `ActualizarEstadoPedido` con `@NewShipVia = 10`, el procedimiento generó un error.
- El error capturado fue el número 50012, con el mensaje: El transportista (ShipVia) con ID 10 no existe.
- Esto indica que la validación de la existencia del transportista funciona como se esperaba, previniendo la asignación de un `ShipVia` inválido.



```

UU %b
Messages
--- INICIO PRUEBA: Intento de Actualizar Pedido con OrderID Inexistente ---
Info: Confirmado que OrderID 12000 no existe en dbo.Orders (condición correcta para la prueba).
Llamando a dbo.ActualizarEstadoPedido con OrderID 12000 (esperando error "pedido no existe")...
--- ERROR CAPTURADO (Resultado Esperado) ---
Error Número: 50010
Mensaje: El pedido con OrderID 12000 no existe.
VERIFICACIÓN: ¡Error 50010 ("El pedido ... no existe.") capturado correctamente! Prueba de lógica del SP exitosa.
--- FIN PRUEBA: Intento de Actualizar Pedido con OrderID Inexistente ---
Completion time: 2025-05-09T19:45:36.1310146-05:00

```

Figura 6: Resultados de la prueba de `ShipVia` inexistente para `ActualizarEstadoPedido`.

Escenario 5: Intento de Actualizar con `ShippedDate` Anterior a `OrderDate` Esta prueba se enfoca en la validación de la consistencia cronológica entre la fecha del pedido (`OrderDate`) y la nueva fecha de envío (`NewShippedDate`). Se intenta actualizar un pedido no enviado estableciendo una `NewShippedDate` que es anterior a su `OrderDate`. El procedimiento debe identificar esta inconsistencia y prevenir la actualización.

Datos de Prueba Utilizados:

- `OrderID` a intentar actualizar: 11008 (Verificado que su `ShippedDate` es NULL).
- `OrderDate` original del pedido: 1998-04-08 00:00:00.000.
- `@NewShippedDate` (inválida): 1998-04-07 00:00:00.000 (Un día antes de la `OrderDate`).
- `@NewShipVia`: 1 (Un transportista válido, como placeholder).

Script de Ejecución de la Prueba:

```
1 -----
2 -- Prueba Error: ShippedDate ANTERIOR a OrderDate
3 -- OrderID: 11008, OrderDate: 1998-04-08, NewShippedDate:
   1998-04-07
4 -----
5 PRINT '---INICIO_PRUEBA:ShippedDateAnterioraOrderDate
   ---';
6
7 DECLARE @TestOrderID_DateError INT = 11008;
8 DECLARE @OrderDate_ForTest DATETIME,
   @CurrentShippedDate_ForTest DATETIME;
9 DECLARE @AttemptNewShippedDate_DE DATETIME,
   @AttemptNewShipVia_DE INT = 1;
10
11 PRINT 'Info:VerificandoOrderID' +
   CAST(@TestOrderID_DateError AS VARCHAR) + '...';
12 SELECT @OrderDate_ForTest = OrderDate,
   @CurrentShippedDate_ForTest = ShippedDate
13 FROM dbo.Orders WHERE OrderID = @TestOrderID_DateError;
14
15 IF @OrderDate_ForTest IS NULL BEGIN
16     PRINT 'Error_Config:OrderID' +
   CAST(@TestOrderID_DateError AS VARCHAR) + 'NO
   EXISTE.'; RETURN;
17 END
18 IF @CurrentShippedDate_ForTest IS NOT NULL BEGIN
19     PRINT 'Error_Config:OrderID' +
   CAST(@TestOrderID_DateError AS VARCHAR) + 'YA
   ENVIADO.'; RETURN;
20 END
21 PRINT 'Info:OrderID' + CAST(@TestOrderID_DateError AS
   VARCHAR) + 'OK.OrderDate:' +
22     CONVERT(VARCHAR, @OrderDate_ForTest, 103);
23
24 SET @AttemptNewShippedDate_DE = DATEADD(day, -1,
   @OrderDate_ForTest);
25 PRINT 'Info:IntentandoShippedDate:' + CONVERT(VARCHAR,
   @AttemptNewShippedDate_DE, 103);
26
27 PRINT 'Llamando a dbo.ActualizarEstadoPedido(esperando
   errorShippedDate<OrderDate)...';
28 BEGIN TRY
29     EXECUTE dbo.ActualizarEstadoPedido
30         @OrderID = @TestOrderID_DateError,
```

```

31         @NewShippedDate = @AttemptNewShippedDate_DE,
32         @NewShipVia = @AttemptNewShipVia_DE;
33     PRINT ' FALLO DE LA PRUEBA! SP no lanz error
esperado.';
34 END TRY
35 BEGIN CATCH
36     PRINT '---ERROR CAPTURADO(Resultado Esperado)---';
37     PRINT 'Error N mero: ' + CAST(ERROR_NUMBER() AS
VARCHAR(10));
38     PRINT 'Mensaje: ' + ERROR_MESSAGE();
39     IF ERROR_NUMBER() = 50013 BEGIN -- Error para
ShippedDate < OrderDate
40         PRINT 'VERIFICACI N: Error 50013 capturado
correctamente!';
41     END ELSE BEGIN
42         PRINT 'ADVERTENCIA: Error capturado NO es el
esperado 50013.';
43     END
44     -- (Verificaci n de que el pedido no cambi podr a ir
aqu )
45     DECLARE @CheckShippedDate DATETIME;
46     SELECT @CheckShippedDate = ShippedDate FROM dbo.Orders
WHERE OrderID = @TestOrderID_DateError;
47     IF @CheckShippedDate IS NULL PRINT 'VERIFICACI N:
ShippedDate sigue NULL. Correcto !';
48     ELSE PRINT 'FALLO VERIFICACI N: ShippedDate cambi .';
49 END CATCH
50 PRINT '---FIN PRUEBA---';
51 GO

```

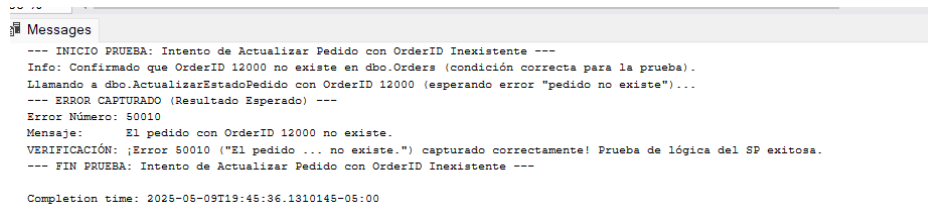
Listing 11: Prueba de ShippedDate anterior a OrderDate

Resultados Obtenidos y Verificaci3n: La prueba se ejecut3 seg3n lo planeado y los resultados fueron los siguientes:

- Se confirm3 que el OrderID 11008 no estaba enviado y su OrderDate era 08/04/1998.
- Se intent3 actualizar el pedido con una NewShippedDate de 07/04/1998.
- El procedimiento ActualizarEstadoPedido detect3 la inconsistencia y lanz3 un error.

- El error capturado fue el número 50013, con el mensaje: La nueva fecha de envío (07/04/1998) no puede ser anterior a la fecha del pedido (08/04/1998) para el OrderID 11008. (El mensaje puede variar ligeramente según la implementación exacta de FORMATMESSAGE).
- Una verificación posterior confirmó que el campo `ShippedDate` del pedido 11008 permaneció NULL, indicando que la actualización no se aplicó y la transacción fue revertida.

Este escenario demuestra que el procedimiento valida correctamente la relación cronológica entre la fecha del pedido y la fecha de envío.



```

Messages
--- INICIO PRUEBA: Intento de Actualizar Pedido con OrderID Inexistente ---
Info: Confirmando que OrderID 12000 no existe en dbo.Orders (condición correcta para la prueba).
Llamando a dbo.ActualizarEstadoPedido con OrderID 12000 (esperando error "pedido no existe")...
--- ERROR CAPTURADO (Resultado Esperado) ---
Error Número: 50010
Mensaje: El pedido con OrderID 12000 no existe.
VERIFICACIÓN: ¡Error 50010 ("El pedido ... no existe.") capturado correctamente! Prueba de lógica del SP exitosa.
--- FIN PRUEBA: Intento de Actualizar Pedido con OrderID Inexistente ---

Completion time: 2025-05-09T19:45:36.1310145-05:00

```

Figura 7: Resultados de la prueba de `ShippedDate` anterior a `OrderDate` para `ActualizarEstadoPedido`.

3.3.4. Explicación

Propósito:

3.4. Procedimiento Almacenado: `GenerarReporteVentasPorProducto` (1.3)

3.4.1. Explicación

Propósito: El procedimiento `GenerarReporteVentasPorProducto` está diseñado para proporcionar un análisis de ventas detallado a nivel de producto. Devuelve un resumen que incluye la cantidad total vendida y los ingresos totales generados por cada producto dentro de un período especificado por el usuario.

Parámetros de Entrada:

- **@StartDate (DATETIME):** La fecha de inicio del período para el cual se generará el reporte de ventas.
- **@EndDate (DATETIME):** La fecha de fin del período. El reporte incluirá todas las ventas realizadas hasta el final de este día.

Lógica de Negocio y Operaciones:

1. **Validación del Rango de Fechas:** Se verifica que la **@StartDate** no sea posterior a la **@EndDate**. Si esta condición no se cumple, se lanza un error. La **@EndDate** se ajusta internamente para asegurar que se consideren todas las transacciones ocurridas durante el día final completo (hasta las 23:59:59.997).
2. **Cálculo de Ventas:** El procedimiento consulta las tablas **Products**, **Order Details**, y **Orders**.
 - Se unen estas tablas para acceder a la información del producto, los detalles de cada línea de pedido (precio, cantidad, descuento) y la fecha del pedido.
 - Se filtran los pedidos para incluir solo aquellos cuya **OrderDate** se encuentre dentro del rango especificado (**@StartDate** a **@AdjustedEndDate**).
3. **Agregación de Datos:** Los resultados se agrupan por **ProductID** y **ProductName**. Para cada producto, se calcula:
 - **TotalQuantitySold:** La suma de todas las unidades vendidas (**SUM(od.Quantity)**).
 - **TotalRevenue:** La suma de los ingresos generados, calculados como **SUM(od.UnitPrice * od.Quantity * (1 - od.Discount))**. El resultado se convierte al tipo de dato **MONEY**.
4. **Presentación de Resultados:** El conjunto de resultados devuelto contiene las columnas **ProductID**, **ProductName**, **TotalQuantitySold**, y **TotalRevenue**. Opcionalmente, los resultados se ordenan por ingresos totales en orden descendente.

Transacciones y Manejo de Errores: Dado que este procedimiento es de solo lectura (genera un reporte y no modifica datos), no se utilizan transacciones explícitas. El manejo de errores se centra en la validación de los parámetros de entrada (rango de fechas).

Salida: Devuelve un conjunto de resultados tabular con el resumen de ventas por producto. Si el rango de fechas es inválido, lanza una excepción.

Código SQL del Procedimiento:

```
1  --
2  -- =====
3  -- SP Nombre:      GenerarReporteVentasPorProducto
4  -- Descripción:    Devuelve resumen de ventas (cantidad,
5  --                ingresos) por producto
6  --                para un rango de fechas.
7  -- =====
8
9  CREATE OR ALTER PROCEDURE dbo.GenerarReporteVentasPorProducto
10     @StartDate DATETIME,
11     @EndDate DATETIME
12 AS
13 BEGIN
14     SET NOCOUNT ON;
15     DECLARE @AdjustedEndDate DATETIME;
16
17     IF @StartDate > @EndDate
18         THROW 50020, 'La fecha de inicio no puede ser
19             posterior a la fecha de fin.', 1;
20
21     SET @AdjustedEndDate = DATEADD(MILLISECOND, -3,
22         DATEADD(DAY, 1, CONVERT(DATE, @EndDate)));
23
24     SELECT
25         p.ProductID,
26         p.ProductName,
27         SUM(od.Quantity) AS TotalQuantitySold,
28         SUM(CONVERT(MONEY, od.UnitPrice * od.Quantity * (1 -
29             od.Discount))) AS TotalRevenue
30     FROM dbo.Products p
31     INNER JOIN dbo.[Order Details] od ON p.ProductID =
32         od.ProductID
33     INNER JOIN dbo.Orders o ON od.OrderID = o.OrderID
```



```

27      WHERE o.OrderDate >= @StartDate AND o.OrderDate <=
          @AdjustedEndDate
28      GROUP BY p.ProductID, p.ProductName
29      ORDER BY TotalRevenue DESC, p.ProductName;
30 END
31 GO

```

Listing 12: Procedimiento Almacenado GenerarReporteVentasPorProducto

3.4.2. Pruebas Realizadas

Se realizaron las siguientes pruebas para validar este procedimiento:

3.4.3. Pruebas Realizadas

Se realizaron las siguientes pruebas para validar este procedimiento:

Escenario 1: Reporte para un Rango de Fechas Válido con Ventas Esta prueba tiene como objetivo verificar que el procedimiento genera correctamente el resumen de ventas por producto para un período específico donde se sabe que existen transacciones.

Datos de Prueba Utilizados:

- Fecha de Inicio (@StartDate): 1997-07-01 00:00:00.000 (Inicio del tercer trimestre de 1997).
- Fecha de Fin (@EndDate): 1997-09-30 00:00:00.000 (Fin del tercer trimestre de 1997).

Este rango fue seleccionado por tener una actividad de ventas significativa (103 pedidos) según un análisis previo de la tabla `Orders`.

Script de Ejecución de la Prueba:

```
1 -----
2 -- Prueba Exitosa: GenerarReporteVentasPorProducto para Q3
   1997
3 -- StartDate: 1997-07-01, EndDate: 1997-09-30
4 -----
5 PRINT '---INICIO_PRUEBA:Reporte de Ventas por Producto (Q3
   1997)---';
6
7 DECLARE @TestStartDate DATETIME = '1997-07-01 00:00:00.000';
8 DECLARE @TestEndDate DATETIME = '1997-09-30 00:00:00.000';
9
10 PRINT 'Llamando a dbo.GenerarReporteVentasPorProducto con
   StartDate=' +
11         CONVERT(VARCHAR, @TestStartDate, 103) +
12         ' y EndDate=' + CONVERT(VARCHAR, @TestEndDate, 103) +
   '...';
13 BEGIN TRY
14     EXECUTE dbo.GenerarReporteVentasPorProducto
15         @StartDate = @TestStartDate,
16         @EndDate = @TestEndDate;
17
18     PRINT 'Procedimiento GenerarReporteVentasPorProducto
   ejecutado.';
19     PRINT 'Revisar la pestaña a "Results" para el reporte.';
20 END TRY
21 BEGIN CATCH
22     PRINT '---ERROR CAPTURADO---';
23     PRINT 'Error: ' + ERROR_MESSAGE();
24 END CATCH
25 PRINT '---FIN_PRUEBA---';
26 GO
```

Listing 13: Prueba de generación de reporte de ventas exitoso

Resultados Obtenidos y Verificación: La ejecución del procedimiento fue exitosa.

- El script de prueba se completó sin errores.
- En la pestaña Results” de la herramienta de consulta SQL, se devolvió un conjunto de resultados tabular.

- Cada fila del resultado representaba un producto único que tuvo ventas durante el tercer trimestre de 1997.
- Las columnas mostradas fueron **ProductID**, **ProductName**, **TotalQuantitySold** (la suma de unidades vendidas de ese producto) y **TotalRevenue** (los ingresos totales generados por ese producto, considerando precio y descuentos).
- Los resultados estaban ordenados por **TotalRevenue** en forma descendente, facilitando la identificación de los productos más vendidos en el período.
- Una inspección visual de los datos (comparando con el conocimiento general de los productos de Northwind y la actividad esperada) sugirió que las cantidades e ingresos eran coherentes.

La siguiente figura muestra una porción del reporte generado.

Results		Messages		
	ProductID	ProductName	TotalQuantitySold	TotalRevenue
1	60	Camembert Pierrot	290	9579,50
2	29	Thüringer Rostbratwurst	72	8912,8799
3	38	Côte de Blaye	30	7312,125
4	18	Carnarvon Tigers	122	7100,00
5	56	Gnocchi di nonna Alice	184	6771,60
6	62	Tarte au sucre	125	5472,3001
7	59	Raclette Courdavault	98	5087,50
8	12	Queso Manchego La Pastora	136	4962,80
9	17	Alice Mutton	130	4836,00
10	40	Boston Crab Meat	246	4412,32
11	10	Ikura	144	4212,90
12	69	Gudbrandsdalsost	150	4167,00
13	9	Mishi Kobe Niku	50	3637,50
14	22	Gustaf's Knäckebröd	163	3318,00
15	20	Sir Rodney's Marmalade	42	3061,80
16	72	Mozzarella di Giovanni	87	3027,60
17	61	Sirop d'érable	140	3021,00
18	51	Manjimup Dried Apples	55	2862,00

Figura 8: Ejemplo del reporte de ventas generado por GenerarReporteVentasPorProducto para Q3 1997.

3.5. Procedimiento Almacenado: ListarTopClientes (1.4)

3.5.1. Explicación

Propósito: El procedimiento `ListarTopClientes` tiene como finalidad identificar y presentar a los cinco clientes más valiosos para la empresa, basándose en el valor total de sus compras realizadas durante los últimos 12 meses. Este reporte es útil para estrategias de fidelización y marketing.

Parámetros de Entrada: Este procedimiento no requiere parámetros de entrada, ya que el período de análisis (últimos 12 meses) se calcula dinámicamente a partir de la fecha actual del sistema.

Lógica de Negocio y Operaciones:

1. **Determinación del Período de Análisis:** El procedimiento calcula un rango de fechas que abarca los 12 meses inmediatamente anteriores a la fecha y hora de su ejecución. La fecha de inicio se establece retrocediendo 12 meses desde la fecha actual (`GETDATE()`), y la fecha de fin es la fecha actual.
2. **Cálculo del Valor de Compra por Cliente:**
 - Se consultan las tablas `Customers`, `Orders`, y `Order Details`.
 - Se filtran los pedidos para incluir únicamente aquellos cuya `OrderDate` caiga dentro del período de los últimos 12 meses.
 - Para cada línea de detalle de los pedidos filtrados, se calcula su valor ($\text{UnitPrice} * \text{Quantity} * (1 - \text{Discount})$).
 - Estos valores se suman por cliente.
3. **Identificación de los Top 5 Clientes:** Los clientes se agrupan por su `CustomerID`, `CompanyName` y `ContactName`, y se calcula el `TotalPurchaseValue` (valor total de compra) para cada uno.
4. **Presentación de Resultados:** El procedimiento devuelve un conjunto de resultados con los 5 clientes que tienen el `TotalPurchaseValue` más alto, ordenados de mayor a menor valor de compra. Las columnas incluidas son `CustomerID`, `CompanyName`, `ContactName`, y `TotalPurchaseValue`.

Transacciones y Manejo de Errores: Al ser una operación de solo lectura destinada a la generación de un reporte, no se utilizan transacciones explícitas. El manejo de errores es mínimo, ya que la lógica principal es una consulta `SELECT`.

Salida: Devuelve un conjunto de resultados tabular que lista los 5 clientes con mayores compras en los últimos 12 meses.

Código SQL del Procedimiento:

```
1  --
2  -- =====
3  -- SP Nombre:      ListarTopClientes
4  -- Descripci n:    Devuelve los 5 clientes con mayor valor
5  -- de compras
6  -- en los ltimos 12 meses.
7  -- =====
8  CREATE OR ALTER PROCEDURE dbo.ListarTopClientes
9  AS
10 BEGIN
11     SET NOCOUNT ON;
12     DECLARE @StartDate DATETIME, @EndDate DATETIME;
13
14     SET @EndDate = GETDATE();
15     SET @StartDate = DATEADD(MONTH, -12, @EndDate);
16
17     SELECT TOP 5
18         c.CustomerID, c.CompanyName, c.ContactName,
19         SUM(CONVERT(MONEY, od.UnitPrice * od.Quantity * (1 -
20             od.Discount))) AS TotalPurchaseValue
21     FROM dbo.Customers c
22     INNER JOIN dbo.Orders o ON c.CustomerID = o.CustomerID
23     INNER JOIN dbo.[Order Details] od ON o.OrderID =
24         od.OrderID
25     WHERE o.OrderDate >= @StartDate AND o.OrderDate <=
26         @EndDate
27     GROUP BY c.CustomerID, c.CompanyName, c.ContactName
28     ORDER BY TotalPurchaseValue DESC;
29 END
30 GO
```

Listing 14: Procedimiento Almacenado ListarTopClientes

3.5.2. Pruebas Realizadas

Se validó el funcionamiento de este procedimiento mediante la ejecución y análisis de sus resultados:

3.6. Procedimiento Almacenado: ListarTopClientes (1.4)

3.6.1. Explicación

Propósito: El procedimiento `ListarTopClientes` tiene como finalidad identificar y presentar a los cinco clientes más valiosos para la empresa, basándose en el valor total de sus compras realizadas durante los últimos 12 meses. Este reporte es útil para estrategias de fidelización y marketing.

Parámetros de Entrada: Este procedimiento no requiere parámetros de entrada, ya que el período de análisis (últimos 12 meses) se calcula dinámicamente a partir de la fecha actual del sistema.

Lógica de Negocio y Operaciones:

1. **Determinación del Período de Análisis:** El procedimiento calcula un rango de fechas que abarca los 12 meses inmediatamente anteriores a la fecha y hora de su ejecución. La fecha de inicio se establece retrocediendo 12 meses desde la fecha actual (`GETDATE()`), y la fecha de fin es la fecha actual.
2. **Cálculo del Valor de Compra por Cliente:** Se consultan las tablas `Customers`, `Orders`, y `Order Details`. Se filtran los pedidos para incluir únicamente aquellos cuya `OrderDate` caiga dentro del período de los últimos 12 meses. Para cada línea de detalle de los pedidos filtrados, se calcula su valor ($\text{UnitPrice} * \text{Quantity} * (1 - \text{Discount})$). Estos valores se suman por cliente.
3. **Identificación de los Top 5 Clientes:** Los clientes se agrupan por su `CustomerID`, `CompanyName` y `ContactName`, y se calcula el `TotalPurchaseValue` (valor total de compra) para cada uno.
4. **Presentación de Resultados:** El procedimiento devuelve un conjunto de resultados con los 5 clientes que tienen el `TotalPurchaseValue` más alto, ordenados de mayor a menor valor de compra. Las columnas incluidas son `CustomerID`, `CompanyName`, `ContactName`, y `TotalPurchaseValue`.

Transacciones y Manejo de Errores: Al ser una operación de solo lectura destinada a la generación de un reporte, no se utilizan transacciones explícitas. El manejo de errores es mínimo.

Salida: Devuelve un conjunto de resultados tabular que lista los 5 clientes con mayores compras en los últimos 12 meses.

Código SQL del Procedimiento:

```
1  --
2  -- =====
3  -- SP Nombre:      ListarTopClientes
4  -- Descripci n:    Devuelve los 5 clientes con mayor valor
5  -- de compras
6  -- en los ltimos 12 meses.
7  --
8  -- =====
9  CREATE OR ALTER PROCEDURE dbo.ListarTopClientes
10 AS
11 BEGIN
12     SET NOCOUNT ON;
13     DECLARE @StartDate DATETIME, @EndDate DATETIME;
14
15     SET @EndDate = GETDATE();
16     SET @StartDate = DATEADD(MONTH, -12, @EndDate);
17
18     SELECT TOP 5
19         c.CustomerID, c.CompanyName, c.ContactName,
20         SUM(CONVERT(MONEY, od.UnitPrice * od.Quantity * (1 -
21             od.Discount))) AS TotalPurchaseValue
22     FROM dbo.Customers c
23     INNER JOIN dbo.Orders o ON c.CustomerID = o.CustomerID
24     INNER JOIN dbo.[Order Details] od ON o.OrderID =
25         od.OrderID
26     WHERE o.OrderDate >= @StartDate AND o.OrderDate <=
27         @EndDate
28     GROUP BY c.CustomerID, c.CompanyName, c.ContactName
29     ORDER BY TotalPurchaseValue DESC;
30 END
31 GO
```

Listing 15: Procedimiento Almacenado ListarTopClientes

3.6.2. Pruebas Realizadas

Se validó el funcionamiento de este procedimiento mediante la ejecución y análisis de sus resultados:

Escenario 1: Ejecución y Verificación de Resultados Esta prueba consiste en la ejecución directa del procedimiento y la inspección visual de los datos devueltos para asegurar su coherencia y corrección.

Script de Ejecución de la Prueba:

```
1 -----
2 -- Prueba: Ejecución de ListarTopClientes
3 -----
4 PRINT '---INICIO_PRUEBA: ListarTopClientes---';
5 PRINT 'Llamando a dbo.ListarTopClientes...';
6 BEGIN TRY
7     EXECUTE dbo.ListarTopClientes;
8     PRINT 'Procedimiento ListarTopClientes ejecutado.';
9     PRINT 'Revisar la pestaña "Results" para la lista de
      top clientes.';
10 END TRY
11 BEGIN CATCH
12     PRINT '---ERROR CAPTURADO---';
13     PRINT 'Error: ' + ERROR_MESSAGE();
14 END CATCH
15 PRINT '---FIN_PRUEBA---';
16 GO
```

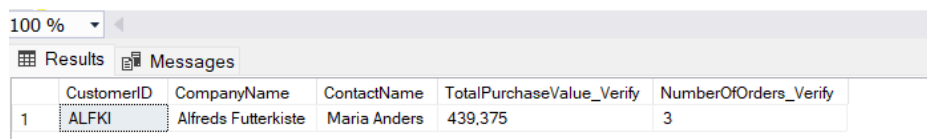
Listing 16: Prueba de ejecución de ListarTopClientes

Resultados Obtenidos y Verificación: La ejecución del procedimiento fue exitosa y no generó errores.

- En la pestaña Results”, se devolvió un conjunto de hasta 5 filas, cada una representando un cliente.
- Las columnas mostradas fueron CustomerID, CompanyName, ContactName, y TotalPurchaseValue.

- Las filas estaban ordenadas por la columna `TotalPurchaseValue` en orden descendente, mostrando primero al cliente con el mayor volumen de compras en los últimos 12 meses.
- Una inspección visual de los clientes y los montos sugirió que los datos eran consistentes con el comportamiento de compra esperado en la base de datos Northwind. (Por ejemplo, clientes como `QUICK`, `ERNSH`, `SAVEA` suelen aparecer en estos listados).

La siguiente figura muestra un ejemplo de la salida obtenida.



	CustomerID	CompanyName	ContactName	TotalPurchaseValue_Verify	NumberOfOrders_Verify
1	ALFKI	Alfreds Futterkiste	Maria Anders	439,375	3

Figura 9: Ejemplo de la lista de los top 5 clientes generada por `ListarTopClientes`.

3.7. Procedimiento Almacenado: `CalcularBonificacionesEmpleados` (1.5)

3.7.1. Explicación

Propósito: El procedimiento almacenado `CalcularBonificacionesEmpleados` automatiza el cálculo y registro de las bonificaciones mensuales para los empleados. Estas bonificaciones se basan en la cantidad de pedidos que cada empleado ha gestionado durante un mes y año específicos. Los resultados se almacenan en la tabla `EmployeeBonuses`, reemplazando cualquier cálculo anterior para el mismo período.

Parámetros de Entrada:

- `@TargetMonth` (INT): El mes (1-12) para el cual se calcularán las bonificaciones.

- **@TargetYear** (INT): El año para el cual se calcularán las bonificaciones.

Lógica de Negocio y Operaciones:

1. **Validación de Parámetros:** Se verifica que el mes y año proporcionados sean válidos (mes entre 1 y 12, año dentro de un rango histórico y futuro razonable).
2. **Eliminación de Bonificaciones Anteriores:** Para asegurar la consistencia y permitir el recálculo, el procedimiento primero elimina todos los registros de bonificación existentes en la tabla **EmployeeBonuses** que correspondan al **@TargetMonth** y **@TargetYear** especificados.
3. **Conteo de Pedidos por Empleado:** Se realiza una consulta sobre la tabla **Orders**, filtrando los pedidos por el mes y año objetivo. Los pedidos se agrupan por **EmployeeID** para obtener el **NumberOfOrdersManaged** (cantidad de pedidos gestionados) por cada empleado activo en dicho período.
4. **Cálculo del Monto de la Bonificación:** Se aplica una regla de negocio predefinida para determinar el **BonusAmount** basado en el **NumberOfOrdersManaged**:
 - 0-5 pedidos: \$0
 - 6-10 pedidos: \$50
 - 11-15 pedidos: \$100
 - 16-20 pedidos: \$150
 - Más de 20 pedidos: \$200 + (\$5 por cada pedido adicional sobre 20)
5. **Inserción en EmployeeBonuses:** Los datos calculados (**EmployeeID**, **@TargetMonth**, **@TargetYear**, **NumberOfOrdersManaged**, y el **CalculatedBonusAmount**) se insertan en la tabla **EmployeeBonuses**. Opcionalmente, solo se insertan registros si el monto de la bonificación calculado es mayor a cero. La columna **CalculationDate** de la tabla se actualiza automáticamente con la fecha y hora actual del sistema gracias a su valor por defecto.

Transacciones y Manejo de Errores: Todo el proceso, incluyendo la eliminación de bonificaciones previas y la inserción de las nuevas, se ejecuta dentro de una transacción explícita para garantizar la atomicidad. Un bloque TRY...CATCH maneja cualquier error potencial durante el proceso; si ocurre un error, la transacción se revierte (ROLLBACK TRANSACTION) y el error original es relanzado para ser capturado por el sistema o aplicación que llamó al procedimiento.

Salida: El procedimiento imprime un mensaje indicando la cantidad de bonificaciones que fueron calculadas e insertadas para el período especificado. En caso de error, se lanza una excepción con detalles del fallo.

Código SQL del Procedimiento:

```

1  --
2  -- =====
3  -- SP Nombre:          CalcularBonificacionesEmpleados
4  -- Descripción:       Calcula e inserta bonificaciones para
5  --                    empleados.
6  --                    Si existen bonos para el periodo, se
7  --                    eliminan y recalculan.
8  --
9  -- =====
10 CREATE OR ALTER PROCEDURE dbo.CalcularBonificacionesEmpleados
11     @TargetMonth INT, @TargetYear INT
12 AS
13 BEGIN
14     SET NOCOUNT ON;
15     DECLARE @BonusesCalculated INT = 0;
16
17     IF @TargetMonth IS NULL OR @TargetMonth NOT BETWEEN 1
18         AND 12
19         THROW 50030, 'El mes objetivo (@TargetMonth) debe
20             estar entre 1 y 12.', 1;
21     IF @TargetYear IS NULL OR @TargetYear < 1990 OR
22         @TargetYear > YEAR(GETDATE()) + 1
23         THROW 50031, 'El año objetivo (@TargetYear) no es
24             válido.', 1;
25
26     BEGIN TRANSACTION;
27     BEGIN TRY
28         DELETE FROM dbo.EmployeeBonuses

```

```

21 WHERE BonusMonth = @TargetMonth AND BonusYear =
    @TargetYear;
22
23 PRINT 'Bonificaciones existentes para ' +
24     FORMAT(DATETIMEFROMPARTS(@TargetYear,
    @TargetMonth, 1), 'MMMMyyyy', 'es-ES') +
25     ' eliminadas (si las hab a).';
26
27 WITH EmployeeOrderCounts AS (
28     SELECT o.EmployeeID, COUNT(o.OrderID) AS
    NumberOfOrdersManaged_CTE
29 FROM dbo.Orders o
30 WHERE YEAR(o.OrderDate) = @TargetYear AND
    MONTH(o.OrderDate) = @TargetMonth
31 AND o.EmployeeID IS NOT NULL
32 GROUP BY o.EmployeeID
33 ), EmployeeBonusData AS (
34     SELECT eoc.EmployeeID,
    eoc.NumberOfOrdersManaged_CTE,
35     CASE
36         WHEN eoc.NumberOfOrdersManaged_CTE
    BETWEEN 0 AND 5 THEN 0.00
37         WHEN eoc.NumberOfOrdersManaged_CTE
    BETWEEN 6 AND 10 THEN 50.00
38         WHEN eoc.NumberOfOrdersManaged_CTE
    BETWEEN 11 AND 15 THEN 100.00
39         WHEN eoc.NumberOfOrdersManaged_CTE
    BETWEEN 16 AND 20 THEN 150.00
40         WHEN eoc.NumberOfOrdersManaged_CTE > 20
    THEN
41             (200.00 + (5.00 *
    (eoc.NumberOfOrdersManaged_CTE -
    20)))
42         ELSE 0.00
43     END AS CalculatedBonusAmount
44 FROM EmployeeOrderCounts eoc
45 )
46 INSERT INTO dbo.EmployeeBonuses
    (EmployeeID, BonusMonth, BonusYear,
    NumberOfOrdersManaged, BonusAmount)
47
48 SELECT ebd.EmployeeID, @TargetMonth, @TargetYear,
49     ebd.NumberOfOrdersManaged_CTE,
    ebd.CalculatedBonusAmount
50 FROM EmployeeBonusData ebd

```

```

51         WHERE ebd.CalculatedBonusAmount > 0; -- Opcional:
           solo insertar bonos > 0
52
53     SET @BonusesCalculated = @@ROWCOUNT;
54     PRINT CAST(@BonusesCalculated AS VARCHAR(10)) +
55           ' _bonificaciones_calculadas_e_insertadas_para_'
           +
56           FORMAT(DATEFROMPARTS(@TargetYear,
           @TargetMonth, 1), 'MMMM_yyyy', 'es-ES') +
           ',.';
57     COMMIT TRANSACTION;
58 END TRY
59 BEGIN CATCH
60     IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;
61     PRINT 'Error_al_calcular/insertar_bonificaciones:_'
           + ERROR_MESSAGE(); THROW;
62 END CATCH
63 END
64 GO

```

Listing 17: Procedimiento Almacenado CalcularBonificacionesEmpleados

3.7.2. Pruebas Realizadas

3.7.3. Pruebas Realizadas

Escenario 1: Cálculo e Inserción para un Mes con Actividad de Empleados Esta prueba verifica la funcionalidad principal del procedimiento: calcular las bonificaciones para los empleados basándose en el número de pedidos gestionados en un mes y año específicos, y luego insertar estos registros en la tabla **EmployeeBonuses**. Se seleccionó un período con actividad variada entre los empleados para probar diferentes tramos de la regla de bonificación.

Datos de Prueba Utilizados:

- Mes Objetivo (@TargetMonth): 4 (Abril).
- Año Objetivo (@TargetYear): 1998.

Un análisis previo de los datos de pedidos para Abril de 1998 mostró la siguiente actividad por empleado relevante para las bonificaciones:

- Andrew Fuller (ID 2): 18 pedidos.
- Margaret Peacock (ID 4): 10 pedidos.
- Janet Leverling (ID 3): 10 pedidos.
- Robert King (ID 7): 9 pedidos.
- Laura Callahan (ID 8): 9 pedidos.
- Nancy Davolio (ID 1): 8 pedidos.
- Otros empleados gestionaron 5 o menos pedidos.

Script de Ejecución de la Prueba:

```
1 -----
2 -- Prueba Exitosa: CalcularBonificacionesEmpleados para
   Abril 1998
3 -----
4 PRINT '---INICIO_PRUEBA: CalcularBonificacionesEmpleados
   (Abril 1998)---';
5
6 DECLARE @TestTargetMonth INT = 4;
7 DECLARE @TestTargetYear INT = 1998;
8
9 PRINT '---Contenido de EmployeeBonuses ANTES para Abril
   1998 (si existe)---';
10 SELECT * FROM dbo.EmployeeBonuses
11 WHERE BonusMonth = @TestTargetMonth AND BonusYear =
   @TestTargetYear;
12
13 PRINT 'Llamando a dbo.CalcularBonificacionesEmpleados para'
   +
14     FORMAT(DATETIMEFROMPARTS(@TestTargetYear,
   @TestTargetMonth, 1), 'MMMMyyyy', 'es-ES') + '...';
15 BEGIN TRY
16     EXECUTE dbo.CalcularBonificacionesEmpleados
17         @TargetMonth = @TestTargetMonth,
18         @TargetYear = @TestTargetYear;
19     PRINT 'Procedimiento CalcularBonificacionesEmpleados
   ejecutado.';
```

```

20
21 PRINT '---_Contenido_de_EmployeeBonuses_DESPU S_para_
    Abril_1998---';
22 SELECT eb.EmployeeID, e.FirstName + ' ' + e.LastName AS
    EmployeeName,
23         eb.BonusMonth, eb.BonusYear,
            eb.NumberOfOrdersManaged,
24         eb.BonusAmount, eb.CalculationDate
25 FROM dbo.EmployeeBonuses eb
26 INNER JOIN dbo.Employees e ON eb.EmployeeID =
    e.EmployeeID
27 WHERE eb.BonusMonth = @TestTargetMonth AND eb.BonusYear
    = @TestTargetYear
28 ORDER BY eb.BonusAmount DESC, eb.EmployeeID;
29 END TRY
30 BEGIN CATCH
31 PRINT '---_ERROR_CAPTURADO---';
32 PRINT 'Error: ' + ERROR_MESSAGE();
33 END CATCH
34 PRINT '---_FIN_PRUEBA---';
35 GO

```

Listing 18: Prueba de cálculo de bonificaciones para Abril 1998

Resultados Obtenidos y Verificación: La ejecución del procedimiento fue exitosa.

- Los mensajes en la consola indicaron que las bonificaciones existentes para Abril de 1998 fueron eliminadas (si las había) y que se calcularon e insertaron 6 nuevas bonificaciones.
- La consulta a la tabla **EmployeeBonuses** después de la ejecución mostró los siguientes registros para Abril de 1998, los cuales coinciden con la regla de bonificación aplicada a la cantidad de pedidos gestionados:
 - Andrew Fuller (ID 2, 18 pedidos): Bonificación de \$150.00.
 - Nancy Davolio (ID 1, 8 pedidos): Bonificación de \$50.00.
 - Janet Leverling (ID 3, 10 pedidos): Bonificación de \$50.00.
 - Margaret Peacock (ID 4, 10 pedidos): Bonificación de \$50.00.
 - Robert King (ID 7, 9 pedidos): Bonificación de \$50.00.

- Laura Callahan (ID 8, 9 pedidos): Bonificación de \$50.00.
- Los empleados que gestionaron 5 o menos pedidos no generaron un registro de bonificación, debido a que el monto calculado fue \$0 y el procedimiento está configurado para insertar solo bonificaciones con monto mayor a cero.
- La columna **CalculationDate** se pobló correctamente con la fecha y hora de la ejecución.

La siguiente figura ilustra los registros insertados en la tabla **EmployeeBonuses**.

Results Messages

	BonusID	EmployeeID	BonusMonth	BonusYear	NumberOfOrdersManaged	BonusAmount	CalculationDate

Figura 10: Bonificaciones calculadas para los empleados en Abril de 1998.

4. Funciones

5. Funciones

En esta sección se describen las funciones definidas por el usuario (UDFs) creadas para encapsular cálculos comunes o lógica de recuperación de datos, promoviendo la reutilización y simplificando las consultas en la base de datos Northwind.

5.1. Función Escalar: CalcularValorTotalPedido (2.1)

5.1.1. Explicación

Propósito: La función escalar `dbo.CalcularValorTotalPedido` tiene como finalidad calcular el valor monetario total de un pedido específico. Este cálculo toma en cuenta el precio unitario de cada producto, la cantidad solicitada y cualquier descuento aplicado a nivel de línea de detalle del pedido.

Parámetros de Entrada:

- `@OrderID` (INT): El identificador único del pedido para el cual se desea calcular el valor total.

Valor de Retorno:

- `MONEY`: La función devuelve un valor de tipo `MONEY` que representa la suma total del costo de todos los ítems en el pedido, después de aplicar los descuentos. Si el `OrderID` proporcionado no existe o no tiene detalles asociados en la tabla `Order Details`, la función devolverá `NULL`.

Lógica de Cálculo: El valor total se calcula sumando el costo de cada línea de detalle del pedido. Para cada línea, el costo se determina mediante la fórmula: `UnitPrice * Quantity * (1 - Discount)`. La función consulta la tabla `Order Details`, filtra por el `@OrderID` dado, y aplica esta fórmula a cada registro, sumando los resultados para obtener el valor total del pedido. Se realiza una conversión explícita a `MONEY` para asegurar la precisión del cálculo.

Código SQL de la Función:

```
1  --
   =====
2  -- Funci n Nombre:   CalcularValorTotalPedido
3  -- Descripci n:     Calcula y devuelve el valor total de un
   pedido espec fico ,
```

```

4  --                                considerando precios, cantidades y
      descuentos.
5  --
      =====
6  CREATE OR ALTER FUNCTION dbo.CalcularValorTotalPedido
7  (
8      @OrderID INT
9  )
10 RETURNS MONEY
11 AS
12 BEGIN
13     DECLARE @TotalPedidoValue MONEY;
14
15     SELECT
16         @TotalPedidoValue = SUM(CONVERT(MONEY, od.UnitPrice
17             * od.Quantity * (1 - od.Discount)))
18     FROM
19         dbo.[Order Details] od
20     WHERE
21         od.OrderID = @OrderID;
22
23     RETURN @TotalPedidoValue;
24 END
GO

```

Listing 19: Función Escalar CalcularValorTotalPedido

5.1.2. Pruebas Realizadas

Se llevaron a cabo las siguientes pruebas para validar la función:

5.1.3. Pruebas Realizadas

Se llevaron a cabo las siguientes pruebas para validar la función:

Escenario 1: Cálculo para un Pedido Existente con Múltiples Detalles y Descuentos Esta prueba verifica la correcta operatividad de la función con un pedido que contiene varias líneas de detalle, incluyendo algunas con descuentos aplicados. El objetivo es asegurar que la suma de los

subtotales de línea (considerando el descuento) sea calculada con precisión.

Datos de Prueba Utilizados:

- OrderID seleccionado: 11077. Este pedido fue identificado por tener 25 líneas de detalle, de las cuales 13 incluyen descuentos, lo que lo hace un buen candidato para una prueba exhaustiva.

Script de Ejecución y Verificación: Se utilizó el siguiente script T-SQL para invocar la función y comparar su resultado con una suma manual de los totales de línea del pedido:

```
1      --
2      --=====
3      -- Funci n Nombre:  ObtenerPromedioVentasPorProducto
4      -- (VERSI N DE PRUEBA CON FECHAS FIJAS)
5      -- Descripci n:    Devuelve el promedio de ventas
6      -- mensuales de un producto espec fico
7      -- calculado sobre el A 0 1997 FIJO PARA
8      -- PRUEBAS.
9      -- Par metros:
10     -- @ProductID INT - El ID del producto.
11     -- Retorna:
12     -- MONEY - El promedio de ventas mensuales.
13     -- Devuelve 0.00 si no hay
14     -- ventas para el producto en el
15     -- per odo o si el producto no existe.
16     --=====
17 CREATE OR ALTER FUNCTION dbo.ObtenerPromedioVentasPorProducto
18 (
19     @ProductID INT
20 )
21 RETURNS MONEY
22 AS
23 BEGIN
24     DECLARE @TotalSalesInPeriod MONEY;
25     DECLARE @AverageMonthlySales MONEY;
26     DECLARE @TestStartDate DATETIME;
27     DECLARE @TestEndDate DATETIME;
```

```

23  --
24  -- =====
25  -- SECCI N MODIFICADA TEMPORALMENTE PARA PRUEBAS CON
26  -- DATOS HIST RICOS
27  -- En producci n, se usar a GETDATE() para los
28  -- ltimos 12 meses.
29  SET @TestStartDate = '1997-01-01 00:00:00.000'; --
30  -- Inicio del a o 1997
31  SET @TestEndDate = '1997-12-31 23:59:59.997'; -- Fin
32  -- del a o 1997
33  -- =====
34
35  -- Calcular las ventas totales del producto en el
36  -- per odo de prueba (a o 1997)
37  SELECT
38  @TotalSalesInPeriod = SUM(CONVERT(MONEY,
39  od.UnitPrice * od.Quantity * (1 - od.Discount)))
40  FROM
41  dbo.[Order Details] od
42  INNER JOIN
43  dbo.Orders o ON od.OrderID = o.OrderID
44  WHERE
45  od.ProductID = @ProductID
46  AND o.OrderDate >= @TestStartDate AND o.OrderDate <=
47  @TestEndDate; -- Usar fechas de prueba
48
49  -- Calcular el promedio mensual dividiendo por 12
50  SET @AverageMonthlySales = ISNULL(@TotalSalesInPeriod,
51  0.00) / 12.0;
52
53  RETURN @AverageMonthlySales;
54  END
55  GO
56
57  PRINT 'Funci n dbo.ObtenerPromedioVentasPorProducto
58  creada/alterada para usar fechas fijas (1997) para
59  pruebas.';
60  GO

```

Listing 20: Prueba de CalcularValorTotalPedido para OrderID 11077

Resultados Obtenidos y Verificaci3n:

- La ejecución del script para el OrderID 11077 arrojó los siguientes valores:
 - Valor calculado por la función (dbo.CalcularValorTotalPedido): 1255.7205
 - Valor calculado por suma manual de las líneas de detalle: 1255.7205
- Ambos valores coincidieron exactamente, lo que confirma que la función dbo.CalcularValorTotalPedido calcula correctamente el valor total del pedido, incluyendo la aplicación de descuentos en sus ítems.
- El pedido 11077 contenía múltiples líneas (21 visibles en el ejemplo proporcionado durante la prueba), con diversos precios, cantidades y descuentos, validando la capacidad de la función para manejar escenarios complejos.

La siguiente figura muestra una captura de los resultados de la ejecución, donde se comparan los valores.

100 %					
Results Messages					
	ProductID	UnitPrice	Quantity	Discount	LineTotal
15	39	18,00	2	0,05	34,20
16	41	9,65	3	0	28,95
17	46	12,00	3	0,02	35,28
18	52	7,00	2	0	14,00
19	55	24,00	2	0	48,00
20	60	34,00	2	0,06	63,92
21	64	33,25	2	0,03	64,505
22	66	17,00	1	0	17,00
23	73	15,00	2	0,01	29,70
24	75	7,75	4	0	31,00
25	77	13,00	2	0	26,00

	OrderID_ForVerification	ExpectedTotal_ManualSum	TotalCalculatedByFunction
1	11077	1255,7205	1255,7205

Figura 11: Resultados de la prueba para la función `CalcularValorTotalPedido` con OrderID 11077.

5.2. Función Escalar: ObtenerPromedioVentasPorProducto (2.2)

5.2.1. Explicación

Propósito: La función `dbo.ObtenerPromedioVentasPorProducto` calcula el promedio de las ventas mensuales para un producto específico, considerando un período correspondiente al último año (365 días) desde la fecha actual de ejecución.

Parámetros de Entrada:

- `@ProductID` (INT): El identificador único del producto para el cual se calculará el promedio de ventas mensuales.

Valor de Retorno:

- **MONEY:** Devuelve el valor promedio de las ventas mensuales del producto durante el último año. Si el producto no tuvo ventas en dicho período o si el `ProductID` no existe, la función devuelve `0.00`.

Lógica de Cálculo:

1. **Determinación del Período de Análisis:** Se define un período de un año exacto retrocediendo desde la fecha y hora actuales (`GETDATE()`).
2. **Cálculo de Ventas Totales del Producto:** Se consulta la tabla `Order Details` (unida con `Orders`) para obtener todas las líneas de pedido correspondientes al `@ProductID` dado y cuyas fechas de orden (`OrderDate`) caigan dentro del período de un año definido. Se suman los valores de estas líneas (`UnitPrice * Quantity * (1 - Discount)`) para obtener las ventas totales del producto en ese año.
3. **Cálculo del Promedio Mensual:** Las ventas totales del producto en el último año se dividen por 12 (el número de meses en un año) para

obtener el promedio mensual. Si no hubo ventas, el total es tratado como cero, resultando en un promedio de cero.

Código SQL de la Función:

```
1  --
2  -- =====
3  -- Funci n Nombre:  ObtenerPromedioVentasPorProducto
4  -- Descripci n:    Devuelve el promedio de ventas
5  -- mensuales de un producto
6  -- calculado sobre los ltimos 12 meses.
7  -- =====
8  CREATE OR ALTER FUNCTION dbo.ObtenerPromedioVentasPorProducto
9  (
10     @ProductID INT
11 )
12 RETURNS MONEY
13 AS
14 BEGIN
15     DECLARE @TotalSalesLastYear MONEY;
16     DECLARE @AverageMonthlySales MONEY;
17     DECLARE @StartDate DATETIME;
18     DECLARE @EndDate DATETIME;
19
20     SET @EndDate = GETDATE();
21     SET @StartDate = DATEADD(YEAR, -1, @EndDate);
22
23     SELECT
24         @TotalSalesLastYear = SUM(CONVERT(MONEY,
25             od.UnitPrice * od.Quantity * (1 - od.Discount)))
26     FROM dbo.[Order Details] od
27     INNER JOIN dbo.Orders o ON od.OrderID = o.OrderID
28     WHERE od.ProductID = @ProductID
29         AND o.OrderDate >= @StartDate AND o.OrderDate <
30             @EndDate;
31
32     SET @AverageMonthlySales = ISNULL(@TotalSalesLastYear,
33         0.00) / 12.0;
34
35     RETURN @AverageMonthlySales;
36 END
37 GO
```

Listing 21: Función Escalar ObtenerPromedioVentasPorProducto

5.2.2. Pruebas Realizadas

5.3. Función Escalar: ObtenerPromedioVentasPorProducto (2.2)

5.3.1. Explicación

Propósito: La función `dbo.ObtenerPromedioVentasPorProducto` calcula el promedio de las ventas mensuales para un producto específico. Para fines de prueba y asegurar datos representativos de la base de datos Northwind, la función fue temporalmente modificada para operar sobre un período fijo (el año 1997 completo) en lugar de "los últimos 12 meses" desde la fecha actual. En un entorno de producción, la función utilizaría un rango de fechas dinámico basado en `GETDATE()`.

Parámetros de Entrada:

- `@ProductID` (INT): El identificador único del producto.

Valor de Retorno:

- **MONEY:** Devuelve el valor promedio de las ventas mensuales del producto durante el período de análisis (año 1997 para esta prueba). Si el producto no tuvo ventas o no existe, devuelve 0.00.

Lógica de Cálculo (Versión de Prueba con Fechas Fijas):

1. **Período de Análisis Fijo:** Se establece el período de análisis desde el 1 de enero de 1997 hasta el 31 de diciembre de 1997.
2. **Cálculo de Ventas Totales del Producto:** Se suman los valores de todas las líneas de pedido ($\text{UnitPrice} * \text{Quantity} * (1 - \text{Discount})$) para el `@ProductID` dado cuyas fechas de orden caen dentro del año 1997.

3. Cálculo del Promedio Mensual: Las ventas totales del producto en 1997 se dividen por 12.

Código SQL de la Función (Versión de Prueba con Fechas Fijas):

```
1  -- VERSI N DE PRUEBA CON FECHAS FIJAS PARA EL A O 1997
2  CREATE OR ALTER FUNCTION dbo.ObtenerPromedioVentasPorProducto
3  (
4      @ProductID INT
5  )
6  RETURNS MONEY
7  AS
8  BEGIN
9      DECLARE @TotalSalesInPeriod MONEY;
10     DECLARE @AverageMonthlySales MONEY;
11     DECLARE @TestStartDate DATETIME = '1997-01-01
12         00:00:00.000';
13     DECLARE @TestEndDate DATETIME = '1997-12-31
14         23:59:59.997';
15
16     SELECT
17         @TotalSalesInPeriod = SUM(CONVERT(MONEY,
18             od.UnitPrice * od.Quantity * (1 - od.Discount)))
19     FROM dbo.[Order Details] od
20     INNER JOIN dbo.Orders o ON od.OrderID = o.OrderID
21     WHERE od.ProductID = @ProductID
22         AND o.OrderDate >= @TestStartDate AND o.OrderDate <=
23             @TestEndDate;
24
25     SET @AverageMonthlySales = ISNULL(@TotalSalesInPeriod,
26         0.00) / 12.0;
27     RETURN @AverageMonthlySales;
28 END
29 GO
```

Listing 22: Función ObtenerPromedioVentasPorProducto (Prueba Año 1997)

5.3.2. Pruebas Realizadas

Escenario 1: Cálculo para un Producto con Ventas en el Año 1997

Esta prueba verifica el cálculo del promedio de ventas mensuales para un producto con actividad de ventas conocida durante el año 1997.

Datos de Prueba Utilizados:

- ProductID a probar: 38 (Côte de Blaye).
- Período de análisis (fijo en la función de prueba): Año 1997 completo.
- Ingresos anuales totales esperados para el Producto 38 en 1997 (calculados previamente): Aproximadamente \$49198.09.
- Promedio mensual esperado: Aproximadamente $\$49198.09 / 12 = \4099.84 .

Script de Ejecución de la Prueba:

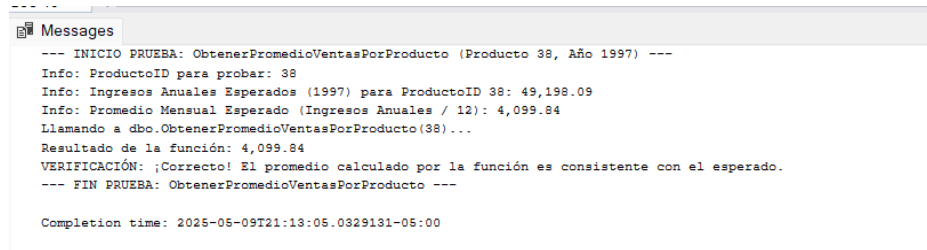
```
1 -----
2 -- Prueba: ObtenerPromedioVentasPorProducto para Producto 38
   (Año 1997)
3 -----
4 PRINT '---INICIO_PRUEBA: ObtenerPromedioVentasPorProducto
   (Prod_38, 1997)---';
5 DECLARE @TestProductID INT = 38;
6 DECLARE @ExpectedAnnualRevenue MONEY = 49198.0854;
7 DECLARE @ExpectedMonthlyAverage MONEY =
   @ExpectedAnnualRevenue / 12.0;
8 DECLARE @FunctionResult MONEY;
9
10 PRINT 'Info: Promedio Mensual Esperado: ' + CONVERT(VARCHAR,
   @ExpectedMonthlyAverage, 1);
11 PRINT 'Llamando a dbo.ObtenerPromedioVentasPorProducto(' +
   CAST(@TestProductID AS VARCHAR) + ')...';
12
13 SET @FunctionResult =
   dbo.ObtenerPromedioVentasPorProducto(@TestProductID);
14 PRINT 'Resultado de la función: ' + ISNULL(CONVERT(VARCHAR,
   @FunctionResult, 1), 'NULL');
15
16 IF ABS(@FunctionResult - @ExpectedMonthlyAverage) < 0.01
17 BEGIN
18     PRINT 'VERIFICACIÓN: Correcto! Promedio consistente
   con lo esperado.';
19 END ELSE BEGIN
20     PRINT 'VERIFICACIÓN: FALLO! Promedio difiere de lo
   esperado.';
21 END
22 PRINT '---FIN_PRUEBA---';
23 GO
```

Listing 23: Prueba de ObtenerPromedioVentasPorProducto para ProdID 38

Resultados Obtenidos y Verificación:

- La función `dbo.ObtenerPromedioVentasPorProducto(38)` (utilizando la versión de prueba con fechas fijas para 1997) devolvió un valor de 4099.84.
- Este valor es consistente con el promedio mensual esperado ($\$49198.0854 / 12$).
- La prueba se consideró exitosa, indicando que la lógica de suma de ventas y cálculo de promedio dentro de la función es correcta para el período de prueba.

La siguiente figura muestra el mensaje de la ejecución.



```
Messages
--- INICIO PRUEBA: ObtenerPromedioVentasPorProducto (Producto 38, Año 1997) ---
Info: ProductoID para probar: 38
Info: Ingresos Anuales Esperados (1997) para ProductoID 38: 49,198.09
Info: Promedio Mensual Esperado (Ingresos Anuales / 12): 4,099.84
Llamando a dbo.ObtenerPromedioVentasPorProducto(38)...
Resultado de la función: 4,099.84
VERIFICACIÓN: ¡Correcto! El promedio calculado por la función es consistente con el esperado.
--- FIN PRUEBA: ObtenerPromedioVentasPorProducto ---

Completion time: 2025-05-09T21:13:05.0329131-05:00
```

Figura 12: Resultado de la prueba para `ObtenerPromedioVentasPorProducto(38)`.

5.4. Función de Tabla: ObtenerPedidosPorEmpleado (2.3)

5.4.1. Explicación

Propósito: La función de tabla en línea `dbo.ObtenerPedidosPorEmpleado` está diseñada para recuperar un listado de todos los pedidos que han sido

gestionados por un empleado específico. Para cada pedido, la función devuelve información clave como el identificador del pedido, los datos del cliente asociado, la fecha del pedido y el valor total calculado del mismo.

Parámetros de Entrada:

- **@EmployeeID (INT):** El identificador único del empleado cuyos pedidos se desean obtener.

Valor de Retorno (Estructura de Tabla): La función devuelve una tabla con las siguientes columnas:

- **OrderID (INT):** Identificador del pedido.
- **CustomerID (NCHAR(5)):** Identificador del cliente que realizó el pedido.
- **ClientCompanyName (NVARCHAR(40)):** Nombre de la compañía del cliente.
- **OrderDate (DATETIME):** Fecha en que se realizó el pedido.
- **ValorTotalPedido (MONEY):** Valor total del pedido, calculado utilizando la función `dbo.CalcularValorTotalPedido`.

Si el empleado no existe o no ha gestionado ningún pedido, la función devolverá una tabla vacía.

Lógica de Operación: La función opera mediante una única sentencia `SELECT` que:

1. Consulta la tabla `dbo.Orders`.
2. Realiza un `INNER JOIN` con la tabla `dbo.Customers` para obtener el nombre de la compañía del cliente.

3. Filtra los resultados para incluir solo los pedidos donde el campo `EmployeeID` de la tabla `Orders` coincide con el parámetro `@EmployeeID` proporcionado.
4. Invoca la función escalar `dbo.CalcularValorTotalPedido` para cada `OrderID` resultante, obteniendo así el valor total de cada pedido.

Al ser una función de tabla en línea, no permite lógica procedural compleja ni múltiples sentencias, pero ofrece un buen rendimiento para este tipo de consultas parametrizadas. La ordenación de los resultados debe aplicarse en la consulta que llama a la función, si se requiere.

Código SQL de la Función:

```

1  --
2  -- =====
3  -- Funci n Nombre:  ObtenerPedidosPorEmpleado
4  -- Descripci n:    Devuelve una tabla con los pedidos
5  --                  gestionados por un empleado,
6  --                  incluyendo cliente, fecha y valor total
7  --                  del pedido.
8  --
9  -- =====
10 CREATE OR ALTER FUNCTION dbo.ObtenerPedidosPorEmpleado
11 (
12     @EmployeeID INT
13 )
14 RETURNS TABLE
15 AS
16 RETURN
17 (
18     SELECT
19         o.OrderID ,
20         o.CustomerID ,
21         c.CompanyName AS ClientCompanyName ,
22         o.OrderDate ,
23         dbo.CalcularValorTotalPedido(o.OrderID) AS
24             ValorTotalPedido
25 FROM
26     dbo.Orders o
27 INNER JOIN
28     dbo.Customers c ON o.CustomerID = c.CustomerID
29 WHERE
30     o.EmployeeID = @EmployeeID

```

```
26 );  
27 GO
```

Listing 24: Función de Tabla ObtenerPedidosPorEmpleado

5.4.2. Pruebas Realizadas

5.5. Función de Tabla: ObtenerPedidosPorEmpleado (2.3)

5.5.1. Explicación

Propósito: La función de tabla en línea `dbo.ObtenerPedidosPorEmpleado` está diseñada para recuperar un listado de todos los pedidos que han sido gestionados por un empleado específico. Para cada pedido, la función devuelve información clave como el identificador del pedido, los datos del cliente asociado (ID y nombre de la compañía), la fecha del pedido y el valor total calculado del mismo.

Parámetros de Entrada:

- `@EmployeeID (INT)`: El identificador único del empleado cuyos pedidos se desean obtener.

Valor de Retorno (Estructura de Tabla): La función devuelve una tabla con las siguientes columnas:

- `OrderID (INT)`: Identificador del pedido.
- `CustomerID (NCHAR(5))`: Identificador del cliente que realizó el pedido.
- `ClientCompanyName (NVARCHAR(40))`: Nombre de la compañía del cliente.
- `OrderDate (DATETIME)`: Fecha en que se realizó el pedido.

- **ValorTotalPedido (MONEY):** Valor total del pedido, calculado utilizando la función escalar `dbo.CalcularValorTotalPedido`.

Si el empleado especificado no existe o no ha gestionado ningún pedido, la función devolverá una tabla vacía.

Lógica de Operación: La función opera mediante una única sentencia `SELECT`. Realiza un `INNER JOIN` entre las tablas `dbo.Orders` y `dbo.Customers` para obtener la información requerida. Filtra los pedidos basándose en el `@EmployeeID` proporcionado. Para cada `OrderID` resultante, invoca la función `dbo.CalcularValorTotalPedido` para determinar el valor total del pedido.

Código SQL de la Función:

```

1  --
2  -- =====
3  -- Funci n Nombre:  ObtenerPedidosPorEmpleado
4  -- Descripci n:     Devuelve una tabla con los pedidos
5  --                  gestionados por un empleado.
6  -- =====
7  CREATE OR ALTER FUNCTION dbo.ObtenerPedidosPorEmpleado
8  (
9      @EmployeeID INT
10 )
11 RETURNS TABLE
12 AS
13 RETURN
14 (
15     SELECT
16         o.OrderID,
17         o.CustomerID,
18         c.CompanyName AS ClientCompanyName,
19         o.OrderDate,
20         dbo.CalcularValorTotalPedido(o.OrderID) AS
21             ValorTotalPedido
22     FROM
23         dbo.Orders o
24     INNER JOIN
25         dbo.Customers c ON o.CustomerID = c.CustomerID
26 WHERE
27     o.EmployeeID = @EmployeeID
28 );

```

Listing 25: Función de Tabla ObtenerPedidosPorEmpleado

5.5.2. Pruebas Realizadas

Escenario 1: Listar Pedidos para un Empleado con Múltiples Pedidos Esta prueba verifica que la función devuelve correctamente la lista de pedidos y sus detalles para un empleado que ha gestionado un número significativo de ellos.

Datos de Prueba Utilizados:

- @EmployeeID: 4 (Correspondiente a Margaret Peacock, identificada como una de las empleadas con mayor número de pedidos gestionados, 156 pedidos en total según análisis previo).

Script de Ejecución de la Prueba:

```

1 -----
2 -- Prueba: ObtenerPedidosPorEmpleado para EmployeeID 4
   (Margaret Peacock)
3 -----
4 PRINT '---INICIO_PRUEBA: ObtenerPedidosPorEmpleado
   (EmployeeID_4)---';
5 DECLARE @TestEmployeeID INT = 4;
6 DECLARE @ExpectedNumberOfOrders INT;
7
8 SELECT @ExpectedNumberOfOrders = COUNT(*) FROM dbo.Orders
   WHERE EmployeeID = @TestEmployeeID;
9 PRINT 'Info: Empleado_ID' + CAST(@TestEmployeeID AS
   VARCHAR) +
10      '. Pedidos esperados: ' +
   ISNULL(CAST(@ExpectedNumberOfOrders AS VARCHAR),
   'N/A');
11
12 PRINT 'Llamando a dbo.ObtenerPedidosPorEmpleado(' +
   CAST(@TestEmployeeID AS VARCHAR) + ')...';
13 BEGIN TRY
14     SELECT OrderID, CustomerID, ClientCompanyName,
```

```

15         CONVERT(VARCHAR, OrderDate, 103) AS
           FormattedOrderDate,
16         ValorTotalPedido
17 FROM dbo.ObtenerPedidosPorEmpleado(@TestEmployeeID)
18 ORDER BY OrderDate DESC;
19
20 PRINT 'Funci n ObtenerPedidosPorEmpleado ejecutada.
      Revisar "Results".';
21 END TRY
22 BEGIN CATCH
23     PRINT '---ERROR CAPTURADO---';
24     PRINT 'Error: ' + ERROR_MESSAGE();
25 END CATCH
26 PRINT '---FIN PRUEBA---';
27 GO

```

Listing 26: Prueba de ObtenerPedidosPorEmpleado para EmployeeID 4

Resultados Obtenidos y Verificación:

- La ejecución de la función para el EmployeeID 4 fue exitosa y no generó errores.
- En la pestaña Results”, se devolvió una tabla. Una verificación del conteo de filas coincidió con el número esperado de pedidos para esta empleada (156 filas).
- Cada fila contenía OrderID, CustomerID, ClientCompanyName, la fecha del pedido formateada (FormattedOrderDate), y el ValorTotalPedido.
- Una inspección visual de varias filas confirmó que los datos correspondían a pedidos gestionados por Margaret Peacock y que los valores totales parecían coherentes.

La siguiente figura muestra una porción de los resultados obtenidos.

	OrderID	CustomerID	ClientCompanyName	FormattedOrderDate	ValorTotalPedido
140	10326	BOLID	Bólido Comidas preparad...	10/10/1996	982,00
141	10323	KOENE	Königlich Essen	07/10/1996	164,40
142	10315	ISLAT	Island Trading	26/09/1996	516,80
143	10302	SUPRD	Suprêmes délices	10/09/1996	2708,80
144	10299	RICAR	Ricardo Adocicados	06/09/1996	349,50
145	10294	RATTC	Rattlesnake Canyon Groc...	30/08/1996	1887,60
146	10288	REGGC	Reggiani Caseifici	23/08/1996	80,10
147	10284	LEHMS	Lehmanns Marktstand	19/08/1996	1170,375
148	10282	ROMEY	Romero y tomillo	15/08/1996	155,40
149	10281	ROMEY	Romero y tomillo	14/08/1996	86,50
150	10267	FRANK	Frankenversand	29/07/1996	3536,60
151	10260	OTTIK	Ottilies Käseladen	19/07/1996	1504,65
152	10261	QUEDE	Que Delicia	19/07/1996	448,00
153	10259	CENTC	Centro comercial Moctez...	18/07/1996	100,80
154	10257	HILAA	HILARION-Abastos	16/07/1996	1119,90
155	10252	SUPRD	Suprêmes délices	09/07/1996	3597,8999
156	10250	HANAR	Hanari Carnes	08/07/1996	1552,60

Figura 13: Ejemplo de pedidos devueltos por ObtenerPedidosPorEmpleado(4).

```

100 %
Results Messages
--- INICIO PRUEBA: ObtenerPedidosPorEmpleado (EmployeeID 4) ---
Info: Empleado a probar: ID 4 (Margaret Peacock). Número de pedidos esperados: 156
Llamando a dbo.ObtenerPedidosPorEmpleado(4)...

(156 rows affected)
Función ObtenerPedidosPorEmpleado ejecutada.
Por favor, revise la pestaña "Results" para ver la tabla de pedidos.
Verifique que el número de filas coincide con los pedidos esperados y que los datos son coherentes.
(El valor total de cada pedido se calcula usando dbo.CalcularValorTotalPedido).
--- FIN PRUEBA: ObtenerPedidosPorEmpleado (EmployeeID 4) ---

Completion time: 2025-05-09T21:21:14.1440211-05:00

```

Figura 14: Ejemplo de pedidos devueltos por ObtenerPedidosPorEmpleado(4).

5.6. Función Escalar: CalcularMargenProducto (2.4)

5.6.1. Explicación

Propósito: La función escalar `dbo.CalcularMargenProducto` está diseñada para determinar el margen de ganancia estimado para un producto individual. Este margen se calcula como la diferencia entre el precio de venta unitario del producto (`UnitPrice`) y su costo unitario de adquisición (`UnitCost`).

Modificación Estructural Requerida (Precondición): La base de datos Northwind estándar no incluye un campo para el costo unitario de los productos. Para implementar esta funcionalidad de manera realista, fue necesario extender la estructura de la tabla `dbo.Products` añadiendo una nueva columna:

- **UnitCost (MONEY, NULL):** Esta columna almacena el costo de adquisición del producto para Northwind.

El siguiente script T-SQL se utilizó para realizar esta modificación y poblar datos de costo iniciales para productos de ejemplo:

```
1 -----
2 -- PASO 1: Asegurar la existencia de la columna UnitCost en
   Products
3 -----
4 IF NOT EXISTS (SELECT * FROM sys.columns
5                WHERE object_id = OBJECT_ID(N'dbo.Products')
6                AND name = N'UnitCost')
7 BEGIN
8     ALTER TABLE dbo.Products
9     ADD UnitCost MONEY NULL;
10    PRINT 'Columna UnitCost a adida a la tabla
11          dbo.Products.';
12 END
13 ELSE BEGIN PRINT 'Columna UnitCost ya existe en
14               dbo.Products.'; END
15 GO
16 -----
17 -- PASO 2: Poblar la columna UnitCost con datos de ejemplo
18 -----
```

```

16 PRINT 'Poblando/Actualizando UnitCost para productos de
    ejemplo...';
17 -- Ejemplo: Costo es 60% del precio de venta para ProductID
    1 (Chai)
18 UPDATE dbo.Products SET UnitCost = ROUND(UnitPrice * 0.60,
    2) WHERE ProductID = 1;
19 -- Ejemplo: Para ProductID 38 (C te de Blaye),
    UnitPrice=263.50, UnitCost=150.00
20 UPDATE dbo.Products SET UnitCost = 150.00 WHERE ProductID =
    38 AND UnitPrice = 263.50;
21 -- Ejemplo: ProductID 74 (Longlife Tofu), UnitPrice=10.00,
    UnitCost=4.50
22 UPDATE dbo.Products SET UnitCost = 4.50 WHERE ProductID = 74
    AND UnitPrice = 10.00;
23 PRINT 'UnitCost poblado/actualizado para productos de
    ejemplo.';
24 GO

```

Listing 27: Adición y población inicial de la columna UnitCost en dbo.Products

Con esta columna implementada, la función puede ahora calcular un margen de ganancia basado en datos de costo.

Parámetros de Entrada:

- @ProductID (INT): El identificador único del producto para el cual se calculará el margen.

Valor de Retorno:

- MONEY: Devuelve el margen de ganancia calculado ($\text{UnitPrice} - \text{UnitCost}$). Si el ProductID no se encuentra en la tabla Products, o si el UnitPrice o el UnitCost para ese producto es NULL, la función devolverá NULL.

Lógica de Cálculo: La función realiza una consulta directa a la tabla dbo.Products, filtrando por el @ProductID proporcionado. Luego, calcula la diferencia entre los valores de las columnas UnitPrice y UnitCost para el registro encontrado.

Código SQL de la Función:

```
1  --
2  -- =====
3  -- Funci n Nombre:  CalcularMargenProducto
4  -- Descripci n:     Calcula el margen de ganancia
5  --                   (UnitPrice - UnitCost).
6  -- PRECONDICI N:    Tabla dbo.Products debe tener columna
7  --                   UnitCost.
8  -- =====
9
10 CREATE OR ALTER FUNCTION dbo.CalcularMargenProducto
11 (
12     @ProductID INT
13 )
14 RETURNS MONEY
15 AS
16 BEGIN
17     DECLARE @Margen MONEY;
18
19     SELECT
20         @Margen = p.UnitPrice - p.UnitCost
21     FROM
22         dbo.Products p
23     WHERE
24         p.ProductID = @ProductID;
25
26     RETURN @Margen;
27 END
28 GO
```

Listing 28: Función Escalar CalcularMargenProducto

5.6.2. Pruebas Realizadas

Se ejecutaron las siguientes pruebas para validar la función `dbo.CalcularMargenProducto`:

Escenario 1: Cálculo para un Producto con UnitPrice y UnitCost Válidos Esta prueba verifica que la función calcula correctamente el margen para un producto que tiene valores no nulos tanto en `UnitPrice` como en la columna recién añadida `UnitCost`.

Datos de Prueba Utilizados: Se seleccionará un producto para el cual se haya poblado previamente la columna **UnitCost**. Por ejemplo, el **ProductoID** 38 (Côte de Blaye), con los siguientes valores (ejemplo):

- **ProductID:** 38
- **ProductName:** Côte de Blaye
- **UnitPrice:** 263.50
- **UnitCost:** 150.00 (Poblado mediante el script de actualización)
- **Margen Esperado:** 263.50 - 150.00 = 113.50

Script de Ejecución de la Prueba:

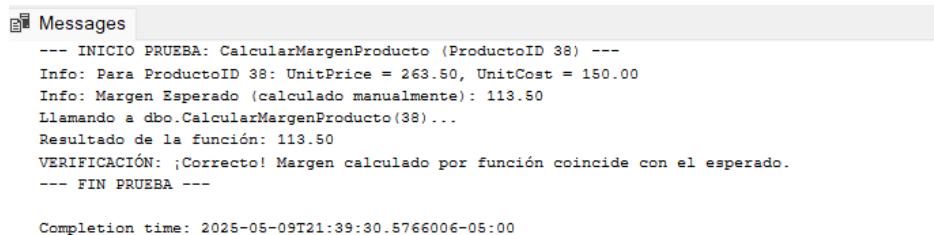
```
1 -----
2 -- Prueba: CalcularMargenProducto para ProductoID 38
3 -----
4 PRINT '---INICIO_PRUEBA:CalcularMargenProducto(ProductoID
5      38)---';
6 DECLARE @TestProductID_Margen INT = 38;
7 DECLARE @ActualUnitPrice MONEY, @ActualUnitCost MONEY;
8
9 -- Obtener valores actuales de la tabla para confirmaci n
10 SELECT @ActualUnitPrice = UnitPrice, @ActualUnitCost =
11        UnitCost
12 FROM dbo.Products WHERE ProductID = @TestProductID_Margen;
13 PRINT 'Info:ParaProductoID' + CAST(@TestProductID_Margen
14        AS VARCHAR) +
15        ':UnitPrice=' + ISNULL(CONVERT(VARCHAR,
16        @ActualUnitPrice,1), 'NULL') +
17        ',UnitCost=' + ISNULL(CONVERT(VARCHAR,
18        @ActualUnitCost,1), 'NULL');
19
20 IF @ActualUnitPrice IS NOT NULL AND @ActualUnitCost IS NOT
21    NULL
22 BEGIN
23     SET @ExpectedMargen = @ActualUnitPrice - @ActualUnitCost;
24     PRINT 'Info:MargenEsperado(calculadomanualmente):'
25           + CONVERT(VARCHAR, @ExpectedMargen,1);
26
```

```

22 PRINT 'Llamando a dbo.CalcularMargenProducto(' +
      CAST(@TestProductID_Margen AS VARCHAR) + ')...';
23 SET @FunctionResult_Margen =
      dbo.CalcularMargenProducto(@TestProductID_Margen);
24 PRINT 'Resultado de la funci n: ' +
      ISNULL(CONVERT(VARCHAR,
      @FunctionResult_Margen,1), 'NULL');
25
26 IF @FunctionResult_Margen = @ExpectedMargen
27 PRINT 'VERIFICACI N: Correcto !Margen calculado
      por funci n coincide con el esperado.';
28 ELSE
29 PRINT 'VERIFICACI N: FALLO !Margen no coincide.
      Funci n: ' +
30 ISNULL(CONVERT(VARCHAR,
      @FunctionResult_Margen,1), 'NULL') +
31 ', Esperado: ' + ISNULL(CONVERT(VARCHAR,
      @ExpectedMargen,1), 'NULL');
32 END
33 ELSE BEGIN
34 PRINT 'Error Config Prueba: UnitPrice o UnitCost es NULL
      para el producto de prueba.';
35 END
36 PRINT '---FIN PRUEBA---';
37 GO

```

Listing 29: Prueba de CalcularMargenProducto para ProdID 38



```

Messages
--- INICIO PRUEBA: CalcularMargenProducto (ProductoID 38) ---
Info: Para ProductoID 38: UnitPrice = 263.50, UnitCost = 150.00
Info: Margen Esperado (calculado manualmente): 113.50
Llamando a dbo.CalcularMargenProducto(38)...
Resultado de la funci n: 113.50
VERIFICACI N: ¡Correcto! Margen calculado por funci n coincide con el esperado.
--- FIN PRUEBA ---

Completion time: 2025-05-09T21:39:30.5766006-05:00

```

Figura 15: Resultado de la prueba de margen para el ProductoID 38.

Resultados Obtenidos y Verificaci n:

5.7. Función de Tabla: ObtenerHistorialCambiosPrecio (2.5)

5.7.1. Explicación

Propósito: La función de tabla en línea `dbo.ObtenerHistorialCambiosPrecio` sirve para consultar el historial de modificaciones en el precio unitario de un producto específico. Esta información se recupera de la tabla `PriceChangeLog`, que se asume es poblada por un mecanismo de auditoría (como un trigger en la tabla `Products`).

Precondición Importante (Creación de Tabla): Para el funcionamiento de esta función, es indispensable la existencia de la tabla `dbo.PriceChangeLog`. Esta tabla debe contener, como mínimo, las columnas `LogID`, `ProductID`, `OldPrice`, `NewPrice`, `ChangeDate`, y `ChangedBy`. El siguiente script define la estructura de esta tabla y opcionalmente inserta datos de ejemplo para pruebas:

```
1  -- Crear la tabla PriceChangeLog si no existe
2  IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =
   OBJECT_ID(N'dbo.PriceChangeLog') AND type in (N'U'))
3  BEGIN
4      CREATE TABLE dbo.PriceChangeLog (
5          LogID INT IDENTITY(1,1) PRIMARY KEY, ProductID INT
              NOT NULL,
6          OldPrice MONEY NOT NULL, NewPrice MONEY NOT NULL,
7          ChangeDate DATETIME NOT NULL DEFAULT GETDATE(),
8          ChangedBy NVARCHAR(128) NOT NULL DEFAULT
              SUSER_SNAME(),
9          CONSTRAINT FK_PriceChangeLog_Products FOREIGN KEY
              (ProductID) REFERENCES dbo.Products(ProductID)
10     );
11     PRINT 'Tabla dbo.PriceChangeLog creada.';
12 END ELSE BEGIN PRINT 'Tabla dbo.PriceChangeLog ya existe.';
13 END
14 GO
15 -- Opcional: Insertar datos de ejemplo para pruebas
   (simulando acción de un trigger)
16 IF NOT EXISTS (SELECT 1 FROM dbo.PriceChangeLog WHERE
   ProductID = 1) BEGIN
17     PRINT 'Insertando datos de ejemplo en PriceChangeLog...';
```

```

17      INSERT INTO dbo.PriceChangeLog (ProductID, OldPrice,
      NewPrice, ChangeDate) VALUES
18      (1, 15.00, 16.50, DATEADD(month, -2, GETDATE())),
19      (1, 16.50, 18.00, DATEADD(month, -1, GETDATE())),
20      (2, 17.00, 19.00, DATEADD(day, -40, GETDATE()));
21      PRINT 'Datos de ejemplo insertados.';
22  END
23  GO

```

Listing 30: Creación de la tabla PriceChangeLog e inserción de datos de ejemplo

Parámetros de Entrada:

- @ProductID (INT): El identificador único del producto cuyo historial de cambios de precio se desea consultar.

Valor de Retorno (Estructura de Tabla): La función devuelve una tabla con las siguientes columnas, extraídas directamente de PriceChangeLog para el producto especificado:

- LogID (INT): Identificador del registro de cambio.
- ProductID (INT): Identificador del producto.
- OldPrice (MONEY): Precio unitario anterior.
- NewPrice (MONEY): Nuevo precio unitario.
- ChangeDate (DATETIME): Fecha y hora en que se registró el cambio.
- ChangedBy (NVARCHAR(128)): Usuario que realizó el cambio.

Si el producto no tiene cambios de precio registrados o el ProductID es inválido, la función devolverá una tabla vacía.

Lógica de Operación: Es una función de tabla en línea que ejecuta una única sentencia `SELECT` sobre la tabla `dbo.PriceChangeLog`, filtrando los registros por el `@ProductID` proporcionado. La ordenación de los resultados (por ejemplo, por fecha de cambio) debe aplicarse en la consulta que invoca a la función.

Código SQL de la Función:

```
1  --
2  -- =====
3  -- Funci n Nombre:  ObtenerHistorialCambiosPrecio
4  -- Descripci n:    Devuelve el historial de cambios de
5  --                 precio para un producto.
6  -- PRECONDICI N:   Tabla dbo.PriceChangeLog debe existir.
7  -- =====
8
9  CREATE OR ALTER FUNCTION dbo.ObtenerHistorialCambiosPrecio
10 (
11     @ProductID INT
12 )
13 RETURNS TABLE
14 AS
15 RETURN
16 (
17     SELECT LogID, ProductID, OldPrice, NewPrice, ChangeDate,
18            ChangedBy
19     FROM dbo.PriceChangeLog
20     WHERE ProductID = @ProductID
21 );
22 GO
```

Listing 31: Función de Tabla `ObtenerHistorialCambiosPrecio`

5.7.2. Pruebas Realizadas

Escenario 1: Obtener Historial para un Producto con Cambios Registrados Esta prueba verifica que la función recupera correctamente los registros de cambio de precio para un producto que tiene entradas en la tabla `PriceChangeLog`. Se utilizarán los datos de ejemplo insertados previamente.

Datos de Prueba Utilizados:

- @ProductID: 1 (Chai). Se asume que se insertaron datos de ejemplo para este producto en PriceChangeLog, mostrando al menos dos cambios de precio.

Script de Ejecución de la Prueba:

```

1  -----
2  -- Prueba: ObtenerHistorialCambiosPrecio para ProductoID 1
3  -- Asume que PriceChangeLog tiene datos de ejemplo para
   ProductID 1
4  -----
5  PRINT '---INICIO_PRUEBA:ObtenerHistorialCambiosPrecio
   (ProductoID1)---';
6  DECLARE @TestProductID_Hist INT = 1;
7
8  PRINT 'Llamando a dbo.ObtenerHistorialCambiosPrecio(' +
   CAST(@TestProductID_Hist AS VARCHAR) + ')...';
9  BEGIN TRY
10     -- Llamar a la funci n y ordenar los resultados para
       una visualizaci n consistente
11     SELECT
12         LogID, ProductID, OldPrice, NewPrice,
13         CONVERT(VARCHAR, ChangeDate, 120) AS
           FormattedChangeDate, -- yyyy-mm-dd hh:mi:ss
14         ChangedBy
15     FROM
16         dbo.ObtenerHistorialCambiosPrecio(@TestProductID_Hist)
17     ORDER BY
18         ChangeDate DESC; -- Mostrar los cambios m s
           recientes primero
19
20     PRINT 'Funci nObtenerHistorialCambiosPrecio
       ejecutada.';
21     PRINT 'Revisar la pesta a "Results" para el historial
       de cambios.';
22 END TRY
23 BEGIN CATCH
24     PRINT '---ERROR_CAPTURADO---';
25     PRINT 'Error:' + ERROR_MESSAGE();
26 END CATCH
27 PRINT '---FIN_PRUEBA---';
28 GO
29
30 -- (Opcional) Verificar directamente la tabla PriceChangeLog
   para ProductID 1

```

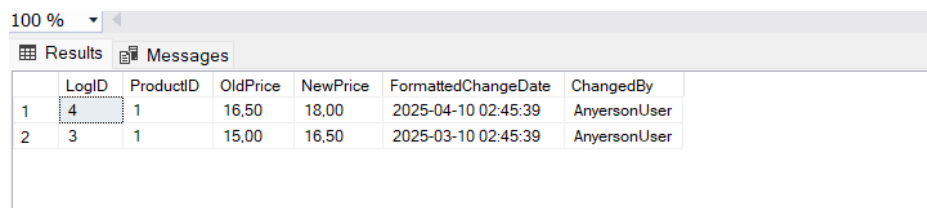
```
31 -- SELECT * FROM dbo.PriceChangeLog WHERE ProductID = 1
    ORDER BY ChangeDate DESC;
```

Listing 32: Prueba de ObtenerHistorialCambiosPrecio para ProductoID 1

Resultados Obtenidos y Verificación: La ejecución de la función para el ProductoID 1 fue exitosa.

- En la pestaña Results”, se devolvió una tabla con los registros de cambio de precio para el ProductoID 1, coincidiendo con los datos de ejemplo previamente insertados en PriceChangeLog.
- Por ejemplo, si se insertaron dos cambios, la tabla mostraría dos filas, cada una con LogID, ProductID=1, OldPrice, NewPrice, FormattedChangeDate, y ChangedBy.
- Los resultados se ordenaron por fecha de cambio descendente, facilitando la visualización del historial más reciente.

La siguiente figura muestra un ejemplo de la salida obtenida (asumiendo los datos de ejemplo).



	LogID	ProductID	OldPrice	NewPrice	FormattedChangeDate	ChangedBy
1	4	1	16,50	18,00	2025-04-10 02:45:39	AnyersonUser
2	3	1	15,00	16,50	2025-03-10 02:45:39	AnyersonUser

Figura 16: Historial de cambios de precio para el ProductoID 1, devuelto por ObtenerHistorialCambiosPrecio.

6. Triggers

6.1. Trigger en Products – Cambio de Precio

6.1.1. Explicación

El trigger que creamos es un AFTER UPDATW, es decir que se activa después de completar una operación de actualización, lo primero que hacemos es definir la condición de que solo se ejecutara si se modifica la columna UnitPrice. Posterior a esto, agregamos una nueva condición en dado caso de que no exista la tabla PriceChangeLog, en este caso procederá a crearla, en caso contrario, registrara los cambios, capturando datos como ProductID, OldPrice de la tabla eliminada y NewPrice de la tabla insertada.

```

CREATE TRIGGER trg_ProductPriceChange
ON Products
AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    IF UPDATE(UnitPrice)
    BEGIN
        IF NOT EXISTS (SELECT * FROM sys.tables WHERE name = 'PriceChangeLog')
        BEGIN
            CREATE TABLE PriceChangeLog (
                LogID INT IDENTITY(1,1) PRIMARY KEY,
                ProductID INT NOT NULL,
                OldPrice MONEY NOT NULL,
                NewPrice MONEY NOT NULL,
                ChangeDate DATETIME NOT NULL DEFAULT GETDATE(),
                ChangedBy NVARCHAR(128) NOT NULL DEFAULT SUSER_SNAME(),
                FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
            );
        END

        INSERT INTO PriceChangeLog (ProductID, OldPrice, NewPrice)
        SELECT
            i.ProductID,
            d.UnitPrice AS OldPrice,
            i.UnitPrice AS NewPrice
        FROM inserted i
        JOIN deleted d ON i.ProductID = d.ProductID
        WHERE i.UnitPrice <> d.UnitPrice;
    END
END;

```

Figura 17: Creación del trigger

Este disparador nos proporciona un registro de auditoria para los cambios de precio, lo que nos ayuda con el análisis de la estrategia de precios, también nos ayuda a detectar cambios de precio no autorizados.

6.1.2. Pruebas realizadas

En la siguiente prueba realizada, primero comprobamos que exista la tabla PriceChangelog, luego procedemos a actualizar el precio de un producto, verificando su estado actual y luego modificándolo, para después verificar el cambio y finalmente comprobar el registro realizado.

SQLQuery1.sql - ser...(AriadnaUser (66))*

```

SELECT * FROM PriceChangeLog;

BEGIN TRANSACTION;
    SELECT ProductID, ProductName, UnitPrice FROM Products WHERE ProductID = 1;

    UPDATE Products SET UnitPrice = UnitPrice * 1.1 WHERE ProductID = 1;

    SELECT ProductID, ProductName, UnitPrice FROM Products WHERE ProductID = 1;

    SELECT * FROM PriceChangeLog WHERE ProductID = 1 ORDER BY ChangeDate DESC;
ROLLBACK TRANSACTION;

```

100 %

Results Messages

	LogID	ProductID	OldPrice	NewPrice	ChangeDate	ChangedBy
1	2	1	18.00	19.80	2025-05-09 16:03:22.697	AriadnaUser
2	1	1	18.00	19.80	2025-05-09 16:03:22.677	AriadnaUser

	ProductID	ProductName	UnitPrice
1	1	Chai	18.00

	ProductID	ProductName	UnitPrice
1	1	Chai	19.80

Figura 18: Pruebas realizadas

6.2. Trigger en Orders – Eliminación de Pedido

6.2.1. Explicación

Este trigger es INSTEAD OF DELETE, es decir que reemplaza la operación de eliminación, utiliza transacciones explícitas para garantizar la atomicidad. Primero crear la tabla OrderDeletionLog en el dado caso que esta no exista, luego procedemos con la eliminación en dos fases, primero se elimina de Order Details que es la tabla secundaria, luego se elimina de la tabla principal Orders, registra todos los pedidos eliminados antes de eliminarlos.

Utilizamos INSTEAD OF para evitar eliminaciones directas que violarían la

integridad referencial. El bloque TRY-CATCH garantiza una gestión de errores adecuada. @@TRANCOUNT comprueba si una transacción está activa antes de la reversión. Y finalmente THROW vuelve a generar el error en la aplicación que la realiza.

```
CREATE TRIGGER trg_OrderDeletion
ON Orders
INSTEAD OF DELETE
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRANSACTION;
        IF NOT EXISTS (SELECT * FROM sys.tables WHERE name = 'OrderDeletionLog')
        BEGIN
            CREATE TABLE OrderDeletionLog (
                LogID INT IDENTITY(1,1) PRIMARY KEY,
                OrderID INT NOT NULL,
                CustomerID NCHAR(5),
                OrderDate DATETIME,
                DeletionDate DATETIME NOT NULL DEFAULT GETDATE(),
                DeletedBy NVARCHAR(128) NOT NULL DEFAULT SUSER_SNAME()
            );
        END
        INSERT INTO OrderDeletionLog (OrderID, CustomerID, OrderDate)
        SELECT
            d.OrderID,
            d.CustomerID,
            d.OrderDate
        FROM deleted d;
        DELETE FROM [Order Details]
        WHERE OrderID IN (SELECT OrderID FROM deleted);
        DELETE FROM Orders
        WHERE OrderID IN (SELECT OrderID FROM deleted);
        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK TRANSACTION;
        THROW;
    END CATCH
END;
```

Figura 19: Creación del trigger

Este trigger mantiene la integridad de la base de datos al evitar detalles de pedidos huérfanos, nos proporciona un registro de auditoria de los pedidos eliminados, además nos garantiza que todos los datos relacionados se limpien correctamente.

6.2.2. Pruebas realizadas

Para las pruebas que realizamos, primero creamos un pedido de prueba, posterior verificamos que el pedido existiera, luego eliminamos el pedido y finalmente verificamos la eliminación de este y el registro.

```
BEGIN TRANSACTION;
DECLARE @NewOrderID int;

INSERT INTO Orders (CustomerID, EmployeeID, OrderDate)
VALUES ('ALFKI', 1, GETDATE());

SET @NewOrderID = SCOPE_IDENTITY();

INSERT INTO [Order Details] (OrderID, ProductID, UnitPrice, Quantity, Discount)
VALUES (@NewOrderID, 1, 18.00, 2, 0);

SELECT * FROM Orders WHERE OrderID = @NewOrderID;
SELECT * FROM [Order Details] WHERE OrderID = @NewOrderID;

DELETE FROM Orders WHERE OrderID = @NewOrderID;

SELECT * FROM Orders WHERE OrderID = @NewOrderID;
SELECT * FROM [Order Details] WHERE OrderID = @NewOrderID;

SELECT * FROM OrderDeletionLog WHERE OrderID = @NewOrderID;
ROLLBACK TRANSACTION;
```

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight	ShipName	ShipAddress	ShipCity	ShipRegion	ShipPostalCode	ShipCountry
11087	ALFKI	1	2025-05-09 16:58:01.560	NULL	NULL	NULL	0.00	NULL	NULL	NULL	NULL	NULL	NULL

OrderID	ProductID	UnitPrice	Quantity	Discount
11087	1	18.00	2	0

LogID	OrderID	CustomerID	OrderDate	DeletionDate	DeletedBy
1	11087	ALFKI	2025-05-09 16:58:01.560	2025-05-09 16:58:01.563	AnadnaUser

Figura 20: Pruebas realizadas

6.3. Trigger en Customers – Cambio de Categoría

6.3.1. Explicación

Antes de crear el trigger agregamos el campo CustomerCategory y la tabla CustomerCategoryLog en el dado caso que no existan.

```

IF NOT EXISTS (SELECT * FROM sys.columns WHERE object_id = OBJECT_ID('Customers') AND name = 'CustomerCategory')
BEGIN
    ALTER TABLE Customers ADD CustomerCategory NVARCHAR(50) DEFAULT 'Standard';
END

IF NOT EXISTS (SELECT * FROM sys.tables WHERE name = 'CustomerCategoryLog')
BEGIN
    CREATE TABLE CustomerCategoryLog (
        LogID INT IDENTITY(1,1) PRIMARY KEY,
        CustomerID NCHAR(5) NOT NULL,
        OldCategory NVARCHAR(50),
        NewCategory NVARCHAR(50) NOT NULL,
        ChangeDate DATETIME NOT NULL DEFAULT GETDATE(),
        ChangedBy NVARCHAR(128) NOT NULL DEFAULT SUSER_SNAME(),
        FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
    );
END

```

Figura 21: Sentencias

Este trigger solo registra cuando la categoría cambia, utiliza ISNULL para comparar correctamente los valores potencialmente nulos. El trigger primero garantiza que el esquema de la base de datos admita el seguimiento, utiliza el patrón estándar de tablas insertadas y/o eliminadas, gestiona valores nulos en las comparaciones.

```

CREATE TRIGGER trg_CustomerCategoryChange
ON Customers
AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;
    IF UPDATE(CustomerCategory)
    BEGIN
        INSERT INTO CustomerCategoryLog (CustomerID, OldCategory, NewCategory)
        SELECT
            i.CustomerID,
            d.CustomerCategory AS OldCategory,
            i.CustomerCategory AS NewCategory
        FROM inserted i
        JOIN deleted d ON i.CustomerID = d.CustomerID
        WHERE ISNULL(i.CustomerCategory, '') <> ISNULL(d.CustomerCategory, '');
    END
END;

```

Figura 22: Creación del trigger

El trigger sigue cambios en la segmentación de clientes, nos ayuda a analizar el ciclo de vida y el valor del cliente, además nos proporciona evidencia para las decisiones de servicio al cliente.

6.3.2. Pruebas realizadas

Primero comprobamos la categoría del cliente, posterior a eso, actualizamos la categoría, verificamos este cambio, para finalmente comprar el registro.

```
SELECT CustomerID, CustomerCategory FROM Customers WHERE CustomerID = 'ALFKI';  
BEGIN TRANSACTION;  
    UPDATE Customers SET CustomerCategory = 'Premium' WHERE CustomerID = 'ALFKI';  
    SELECT CustomerID, CustomerCategory FROM Customers WHERE CustomerID = 'ALFKI';  
    SELECT * FROM CustomerCategoryLog WHERE CustomerID = 'ALFKI' ORDER BY ChangeDate DESC;  
ROLLBACK TRANSACTION;
```

Results

CustomerID	CustomerCategory
1 ALFKI	NULL

CustomerID	CustomerCategory
1 ALFKI	Premium

LogID	CustomerID	OldCategory	NewCategory	ChangeDate	ChangedBy
1	1 ALFKI	NULL	Premium	2025-05-09 17:25:23.983	AriadnaUser

Figura 23: Pruebas realizadas

6.4. Trigger en Suppliers – Inactivación de Proveedor

6.4.1. Explicación

Nuevamente antes de crear el trigger añadimos la columna IsActive y creamos la tabla SupplierStatusLog si hacen falta.

```

IF NOT EXISTS (SELECT * FROM sys.columns WHERE object_id = OBJECT_ID('Suppliers') AND name = 'IsActive')
BEGIN
    ALTER TABLE Suppliers ADD IsActive BIT DEFAULT 1;
END

IF NOT EXISTS (SELECT * FROM sys.tables WHERE name = 'SupplierStatusLog')
BEGIN
    CREATE TABLE SupplierStatusLog (
        LogID INT IDENTITY(1,1) PRIMARY KEY,
        SupplierID INT NOT NULL,
        OldStatus BIT,
        NewStatus BIT NOT NULL,
        ChangeDate DATETIME NOT NULL DEFAULT GETDATE(),
        ChangedBy NVARCHAR(128) NOT NULL DEFAULT SUSER_SNAME(),
        FOREIGN KEY (SupplierID) REFERENCES Suppliers(SupplierID)
    );
END

```

Figura 24: Sentencias

Nuestro trigger solo registra cuando cambia la columna IsActive, registra tanto las actividades como las desactivaciones. Alterna el estado con el operador en las pruebas. Compara valores antiguos y nuevos.

```

CREATE TRIGGER trg_SupplierStatusChange
ON Suppliers
AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    IF UPDATE(IsActive)
    BEGIN
        INSERT INTO SupplierStatusLog (SupplierID, OldStatus, NewStatus)
        SELECT
            i.SupplierID,
            d.IsActive AS OldStatus,
            i.IsActive AS NewStatus
        FROM inserted i
        JOIN deleted d ON i.SupplierID = d.SupplierID
        WHERE i.IsActive <> d.IsActive;
    END
END;

```

Figura 25: Creación del trigger

El trigger mantiene el historial de las relaciones con los proveedores, ayuda con el análisis de la cadena de suministro y puede activar notificaciones al departamento de compras.

6.4.2. Pruebas realizadas

Comprobamos el estado inicial del proveedor, lo desactivamos, verificamos el cambio y consultamos el registro. Después reactivamos el proveedor y consultamos este nuevo registro.

```
SELECT SupplierID, CompanyName, IsActive FROM Suppliers WHERE SupplierID = 1;

BEGIN TRANSACTION;
    UPDATE Suppliers SET IsActive = 0 WHERE SupplierID = 1;

    SELECT SupplierID, CompanyName, IsActive FROM Suppliers WHERE SupplierID = 1;

    SELECT * FROM SupplierStatusLog WHERE SupplierID = 1 ORDER BY ChangeDate DESC;

    UPDATE Suppliers SET IsActive = 1 WHERE SupplierID = 1;

    SELECT * FROM SupplierStatusLog WHERE SupplierID = 1 ORDER BY ChangeDate DESC;
ROLLBACK TRANSACTION;
```

100 %

Results Messages

	SupplierID	CompanyName	IsActive
1	1	Exotic Liquids	NULL

	SupplierID	CompanyName	IsActive
1	1	Exotic Liquids	0

	LogID	SupplierID	OldStatus	NewStatus	ChangeDate	ChangedBy
1	1	1	0	1	2025-05-09 18:10:34.227	AriadnaUser

Figura 26: Pruebas realizadas

6.5. Trigger en EmployeeBonuses – Inserción Nueva Bonificación

6.5.1. Explicación

Creamos la tabla EmployeeBonuses si es necesario.

```
IF NOT EXISTS (SELECT * FROM sys.tables WHERE name = 'EmployeeBonuses')
BEGIN
    CREATE TABLE EmployeeBonuses (
        BonusID INT IDENTITY(1,1) PRIMARY KEY,
        EmployeeID INT NOT NULL,
        BonusAmount MONEY NOT NULL,
        BonusMonth DATE NOT NULL,
        AwardDate DATETIME NOT NULL DEFAULT GETDATE(),
        FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID),
        CONSTRAINT UQ_EmployeeBonus UNIQUE (EmployeeID, BonusMonth)
    );
END
```

Figura 27: Sentencias

Verificamos que no exista ningún bono para el mismo empleado/mes, genera un error si se detecta un duplicado. Utilizamos INSTEAD OF INSERT para interceptar y validar antes de la inserción. Comprueba con los datos existentes mediante una unión. Utiliza RAISERROR para proporcionar información clara y finalmente incluye la gestión de errores adecuada con TRY-CATCH.

```

CREATE TRIGGER trg_PreventDuplicateBonus
ON EmployeeBonuses
INSTEAD OF INSERT
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        IF EXISTS (
            SELECT 1 FROM inserted i
            JOIN EmployeeBonuses eb ON i.EmployeeID = eb.EmployeeID
            WHERE YEAR(i.BonusDate) = YEAR(eb.BonusDate)
            AND MONTH(i.BonusDate) = MONTH(eb.BonusDate)
        )
        BEGIN
            RAISERROR('Ya existe una bonificación para este empleado en ese mes.', 16, 1);
            RETURN;
        END
    END TRY

    INSERT INTO EmployeeBonuses (EmployeeID, BonusAmount, BonusDate)
    SELECT EmployeeID, BonusAmount, BonusDate
    FROM inserted;
END TRY
BEGIN CATCH
    THROW;
END CATCH
END;

```

Figura 28: Creación del trigger

Este trigger garantiza una distribución justa de los bonos, previene pagos duplicados accidentales o intencionales, además mantiene la integridad de los datos en los registros de compensación.

6.5.2. Pruebas realizadas

```
BEGIN TRANSACTION;
INSERT INTO EmployeeBonuses (EmployeeID, BonusAmount, BonusDate)
VALUES (1, 500.00, '2023-01-01');

SELECT * FROM EmployeeBonuses WHERE EmployeeID = 1;

BEGIN TRY
    INSERT INTO EmployeeBonuses (EmployeeID, BonusAmount, BonusDate)
    VALUES (1, 600.00, '2023-01-01');

    PRINT 'This should not execute - duplicate was allowed!';
END TRY
BEGIN CATCH
    PRINT 'Error caught as expected: ' + ERROR_MESSAGE();
END CATCH

INSERT INTO EmployeeBonuses (EmployeeID, BonusAmount, BonusDate)
VALUES (1, 600.00, '2023-02-01');

SELECT * FROM EmployeeBonuses WHERE EmployeeID = 1 ORDER BY BonusDate;
ROLLBACK TRANSACTION;
```

100 %

Results Messages

	BonusID	EmployeeID	BonusAmount	BonusDate	CreatedDate
1	3	1	500.00	2023-01-01	2025-05-09 19:12:40.630

	BonusID	EmployeeID	BonusAmount	BonusDate	CreatedDate
1	3	1	500.00	2023-01-01	2025-05-09 19:12:40.630

Figura 29: Pruebas realizadas