Tema 3 - Manejo de Transacciones

1. Manejo de Transacciones

1.1. Definición: ¿Qué es una Transacción?

En el contexto de base de datos, una **transacción** no es simplemente una consulta individual (como `INSERT` u `UPDATE`). Se define como una **unidad de trabajo indivisible** que agrupa un conjunto de operaciones de lectura y de escritura.

El objetivo principal de una transacción es asegurar que es "paquete de trabajo" se ejecute de forma completa y correcta, o que no se ejecute en absoluto. Esto garantiza que la base de datos nunca quede en un estado intermedio o inconsistente, incluso frente a errores del sistema, fallos de hardware o problemas de concurrencia.

1.2. El Propósito: Las Propiedades ACID

El "para qué sirve" de las transacciones se resume en el acrónimo **ACID**, que representa las cuatro garantías fundamentales que una transacción debe proveer para proteger la integridad de los datos:

1. **Atomicidad (Atomicity):** Es la propiedad de "todo o nada". Asegura que todas las operaciones dentro de la transacción se completen con éxito (`COMMIT`) o, si una sola de ellas falla, que todas las operaciones anteriores se reviertan (`ROLLBACK`), dejando la base de datos en su estado original.
2. **Consistencia (Consistency):** Asegura que la base de datos siempre pase de un estado válido a otro estado válido. Una transacción no puede violar las reglas de integridad definidas (como `CHECK`, `FOREIGN KEY`, `NOT NULL`). Si una operación viola una de estas reglas, la transacción debe fallar (y el `ROLLBACK` atómico se encarga de mantener la consistencia).
3. **Aislamiento (Isolation):** Define cómo y cuándo los cambios producidos por una transacción son visibles para otras transacciones concurrentes (que se ejecutan al mismo tiempo). Evita que dos usuarios "se pisen" los datos mutuamente.

4. **Durabilidad (Durability):** Garantiza que una vez la transacción ha sido confirmada (`COMMIT`) , sus cambios son permanentes y sobrevivirán a cualquier fallo posterior del sistema (como un corte de energía).

1.3. **Aplicación en la Base de Datos "tienda_ropa"**

Para nuestro proyecto, el escenario con mayor riesgo y que justifica de forma crítica el uso de transacciones es el proceso de "Registrar una Venta".

Si analizamos el esquema de nuestra base de datos, una venta no es una única operación, sino una secuencia de, al menos, tres operaciones lógicas que deben ser **atómicas:**

1. **INSERT en** `ticket` : Se debe crear el encabezado de la venta, registrando el DNI del cliente y la fecha.
2. **INSERT en** `detalle_ticket` : Se deben registrar los productos específicos que componen esa venta (cantidad, subtotal) y vincularlos al `ticket` creado en el paso 1.
3. **UPDATE en** `producto` : Se debe descontar el `stock` de los productos vendidos para mantener el inventario actualizado.

El Riesgo (El Problema): ¿Qué sucedería si el sistema registra el `ticket` (Paso 1) y el `detalle_ticket` (Paso 2), pero produce un error (ej: el producto 2 no tiene suficiente stock, o el sistema se reinicia) antes de poder ejecutar el `UPDATE` del `stock` (Paso 3)?

La Consecuencia (Inconsistencia): La base de datos quedaría en un estado corrupto:

- El cliente tendría un ticket de una venta que aparentemente fue exitosa.
- Pero el `stock` de los productos nunca se descontaría.

Esto generaría inconsistencias graves en el inventario, la facturación y los reportes gerenciales.

> **La Solución (La Transacción):** Al envolver estos tres pasos dentro de un bloque `BEGIN TRANSACTION ... COMMIT/ROLLBACK` , garantizamos que si el Paso 3 falla, automáticamente se revertirán el Paso 1 y el Paso 2. El cliente no tendrá un ticket fantasma y el stock seguirá siendo correcto.

---

2. Tareas: Implementación de Transacciones

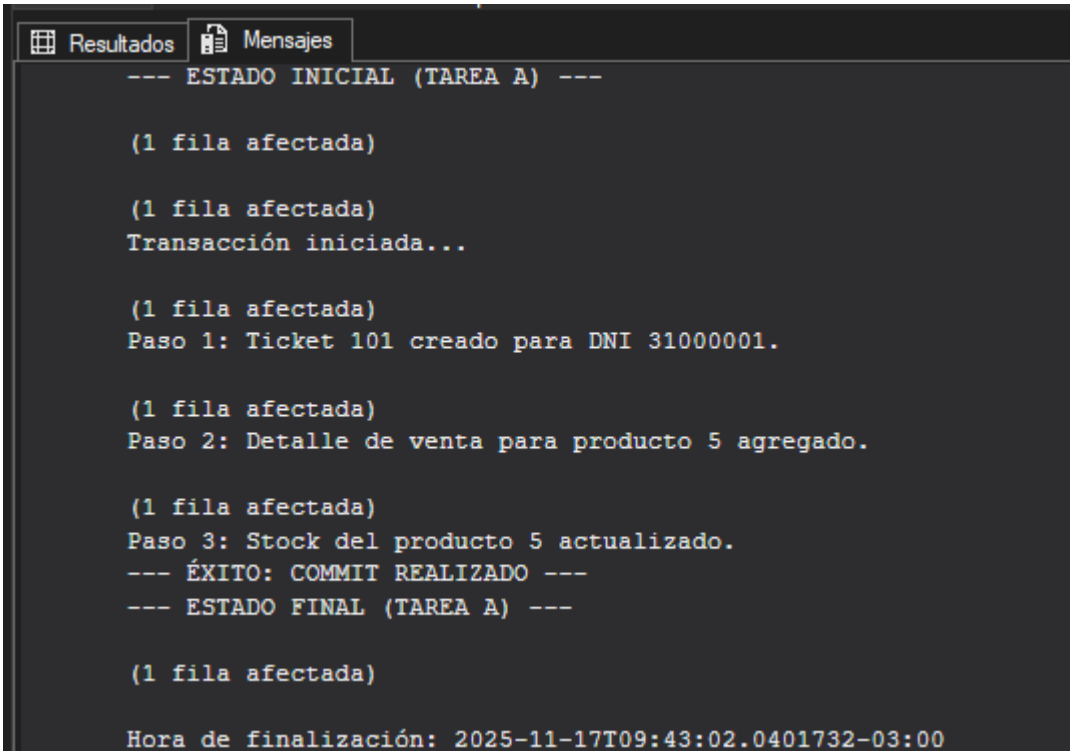
Script de Transacción Exitosa (COMMIT)

Objetivo: Demostrar cómo las operaciones (`INSERT` en `ticket`, `INSERT` en `detalle_ticket` y `UPDATE` en `producto`) se ejecutan como un bloque atómico y exitoso usando `COMMIT`.

Script (Script-Venta-Exitosa.sql):

```
USE tienda_ropa;
GO

PRINT '--- ESTADO INICIAL ---';
-- Revisamos el DNI el primero usuario (31000001) de la tabla usuario y el p
roducto 5
SELECT TOP 1 dni, nombre_usuario FROM usuario;
SELECT nombre_producto, stock, precio FROM producto WHERE id_produc
to = 5;
GO

BEGIN TRY
   BEGIN TRANSACTION;
   PRINT 'Transacción iniciada...';

   -- 1. Tarea A: Insertar en Tabla A (ticket)
   -- Usamos el DNI VÁLIDO '31000001'
   INSERT INTO ticket (dni) VALUES (31000001);

   DECLARE @NuevoTicketID INT = SCOPE_IDENTITY();
   PRINT 'Paso 1: Ticket ' + CAST(@NuevoTicketID AS VARCHAR) + ' cread
o para DNI 31000001.';

   -- 2. Tarea B: Insertar en Tabla B (detalle_ticket)
   -- Vendemos 2 'Buzo Hoodie Negro' (ID 5). Precio 9980 c/u = Subtotal 1
9960
   INSERT INTO detalle_ticket (cantidad, subtotal, id_ticket, id_producto)
   VALUES (2, 19960, @NuevoTicketID, 5);
   PRINT 'Paso 2: Detalle de venta para producto 5 agregado.';
```

```
    -- 3. Tarea C: Actualizar Tabla C (producto)
    -- Descontamos el stock
    UPDATE producto
    SET stock = stock - 2
    WHERE id_producto = 5;
    PRINT 'Paso 3: Stock del producto 5 actualizado.';

    COMMIT TRANSACTION;
    PRINT '--- ÉXITO: COMMIT REALIZADO ---';

END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT '--- ERROR: ROLLBACK REALIZADO ---';
    PRINT 'Mensaje de Error: ' + ERROR_MESSAGE();
END CATCH
GO

PRINT '--- ESTADO FINAL (TAREA A) ---';
-- El stock debe ser 63
SELECT nombre_producto, stock FROM producto WHERE id_producto = 5;
GO
```



Script de Transacción Fallida (ROLLBACK)

Objetivo: Provocar intencionalmente un error para demostrar que la transacción es atómica. Si un paso falla, el `CATCH` ejecuta un `ROLLBACK`, revirtiendo **TODOS** los pasos anteriores, incluso los que sí fueron exitosos.

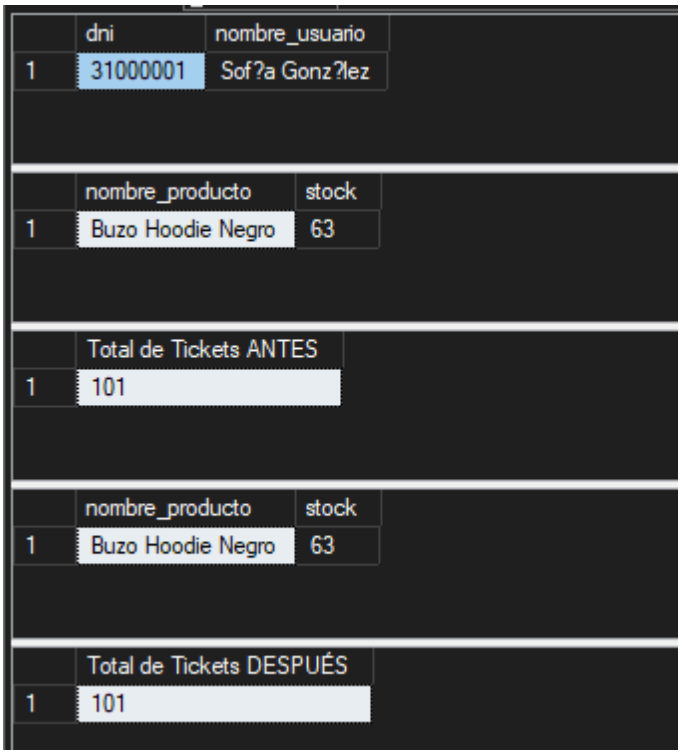
El Error: Forzaremos un error intentando vender un producto que no existe (`id_producto = 9999`), lo cual viola la restricción de Clave Foránea (`FOREIGN KEY`) en la tabla `detalle_ticket`.

Script (Script-Venta-Fallida.sql):

```
USE tienda_ropa;
GO
DECLARE @DniValido INT = 31000001; -- ←—— (Usamos el mismo usuario del caso de exito)
DECLARE @ProductoID_Ok INT = 5;     -- (Usamos producto 5 'Buzo Hoodie Negro')
DECLARE @ProductoID_Falla INT = 9999; -- (Producto inexistente para forzar error)
-- ---------------------------------------------

PRINT '--- ESTADO INICIAL ---';
SELECT dni, nombre_usuario FROM usuario WHERE dni = @DniValido;
```

```
SELECT nombre_producto, stock FROM producto WHERE id_producto = @ProductoID_Ok;
SELECT COUNT(*) AS 'Total de Tickets ANTES' FROM ticket;

BEGIN TRY
    BEGIN TRANSACTION;
    PRINT 'Transacción iniciada...';

    -- 1. Tarea A: Insertar en Tabla A (ticket) (Este paso FUNCIONA)
    INSERT INTO ticket (dni) VALUES (@DniValido);
    DECLARE @NuevoTicketID INT = SCOPE_IDENTITY();
    PRINT 'Paso 1: Ticket ' + CAST(@NuevoTicketID AS VARCHAR) + ' creado para DNI ' + CAST(@DniValido AS VARCHAR);

    -- 2. Tarea B: Insertar en Tabla B (Este paso FUNCIONA)
    INSERT INTO detalle_ticket (cantidad, subtotal, id_ticket, id_producto)
    VALUES (1, 1500.00, @NuevoTicketID, @ProductoID_Ok);
    PRINT 'Paso 2: Detalle de venta (producto OK) agregado.';

    -- 3. Tarea B (continuación): Insertar producto FANTASMA (Este paso FALLA)
    PRINT 'Paso 3: Intentando agregar producto inexistente (ID ' + CAST(@ProductoID_Falla AS VARCHAR) + ')...';
    INSERT INTO detalle_ticket (cantidad, subtotal, id_ticket, id_producto)
    VALUES (1, 100.00, @NuevoTicketID, @ProductoID_Falla); -- ¡ERROR DE FK!

    -- 4. Tarea C: Actualizar Tabla C (Esta línea NUNCA se ejecuta)
    UPDATE producto SET stock = stock - 1 WHERE id_producto = @ProductoID_Ok;
    PRINT 'Paso 4: Stock actualizado (NO DEBERÍA LLEGAR AQUÍ).';

    COMMIT TRANSACTION;

END TRY
BEGIN CATCH
    -- 5. El error del Paso 3 nos envía directamente aquí
    ROLLBACK TRANSACTION;
```

```
    PRINT '--- ERROR DETECTADO: ROLLBACK REALIZADO ---';
    PRINT 'Mensaje de Error: ' + ERROR_MESSAGE();
END CATCH

PRINT '--- ESTADO FINAL (SIN CAMBIOS) ---';
SELECT nombre_producto, stock FROM producto WHERE id_producto = @
ProductoID_Ok;
SELECT COUNT(*) AS 'Total de Tickets DESPUÉS' FROM ticket;
```



	dni	nombre_usuario
1	31000001	Sof?a Gonz?lez

	nombre_producto	stock
1	Buzo Hoodie Negro	63

	Total de Tickets ANTES
1	101

	nombre_producto	stock
1	Buzo Hoodie Negro	63

	Total de Tickets DESPUÉS
1	101

```
--- ESTADO INICIAL ---

(1 fila afectada)

(1 fila afectada)

(1 fila afectada)
Transacción iniciada...

(1 fila afectada)
Paso 1: Ticket 108 creado para DNI 31000001

(1 fila afectada)
Paso 2: Detalle de venta (producto OK) agregado.
Paso 3: Intentando agregar producto inexistente (ID 9999)...

(0 filas afectadas)
--- ERROR DETECTADO: ROLLBACK REALIZADO ---
Mensaje de Error: The INSERT statement conflicted with the FOREIGN KEY constraint "fk_detalle_producto". The conflict occurred in database "tienda_ropa", table "dbo.producto", column 'id_producto'
--- ESTADO FINAL (SIN CAMBIOS) ---

(1 fila afectada)

(1 fila afectada)

Hora de finalización: 2025-11-17T09:25:13.7508292-03:00
```

3. Conclusión sobre el manejo de Transacciones

Al finalizar las pruebas de transacción exitosa (`COMMIT` exitoso) y transacción fallida (`ROLLBACK` forzado), se extraen las siguientes conclusiones fundamentales.

3.1 Las Transacciones como Requisito de Negocio

El análisis conceptual (Sección 1) y las pruebas prácticas (Sección 2) demuestran que el Manejo de Transacciones **no es una optimización opcional, sino un requisito indispensable** para la lógica de negocio del sistema.

Operaciones críticas como el "Registro de Venta" involucran múltiples tablas (`ticket` , `detalle_ticket` , `producto`) que deben actualizarse de forma simultanea. Sin un bloque transaccional, una falla en mitad del proceso (como un `id_producto` incorrecto o un corte de sistema) dejaría la base de datos en un **estado inconsistente.** Esto generaría "datos huérfanos" (tickets sin detalles) o inconsistencias graves de inventario (ventas registradas sin el descuento de stock correspondiente), invalidando la fiabilidad de los reportes.

3.2 Garantía de Atomicidad (ACID)

La transacción fallida (`script_transaq_fallido.sql`) validó exitosamente la propiedad de **Atomicidad** (la 'A' de ACID). Al provocar un error de `FOREIGN KEY` , el bloque `CATCH` interceptó el fallo y ejecutó el `ROLLBACK TRANSACTION` .

La captura de pantalla del resultado final demuestra que **todos** los cambios (incluyendo el `INSERT` en la tabla `ticket` que SÍ había funcionado en el Paso 1) fueron revertidos, protegiendo la integridad de la base de datos y devolviéndola a su estado original, como si la operación nunca hubiera ocurrido.

3.3. Conclusión Final

El uso del bloque `TRY...CATCH` junto con los comandos `BEGIN TRANSACTION` , `COMMIT` y `ROLLBACK` (demostrado en los scripts) se establece como el **patrón de diseño fundamental** para implementar operaciones complejas en la base de datos "tienda_ropa". Esta metodología garantiza que, independientemente de los errores de datos o fallas inesperadas del sistema, la base de datos siempre permanecerá en un estado válido, consistente y fiable.