TEMA 3 : Manejo de Transacciones y Transacciones Anidadas

Una transacción es una unidad lógica de trabajo en la que se agrupan una o varias operaciones de base de datos (como INSERT, UPDATE, DELETE, incluso SELECT en ciertos casos) que deben ejecutarse como un todo.En SQL Server, cualquier modificación de datos puede estar contenida dentro de una transacción explícita o implícita.El objetivo de agruparlas es asegurar que o bien todas las operaciones se efectúan satisfactoriamente, o bien ninguna se aplique evitando que queden datos en un estado intermedio inconsistente.

Propiedades claves ACID
Para que una transacción cumpla su propósito correctamente, debe satisfacer las propiedades conocidas como ACID:

• **Atomicidad**: La operación es atómica; o se hacen todas las acciones del bloque o ninguna.
• **Consistencia**: La base de datos pasa de un estado válido a otro, respetando todas las reglas de integridad (restricciones, relaciones, índices, etc.).
• **Aislamiento**: Las operaciones de una transacción no deben interferir con otras transacciones concurrentes de forma que causen resultados incorrectos.
• **Durabilidad**: Una vez que la transacción se confirma (COMMIT), los cambios persisten incluso en caso de fallos del sistema.

Modo de definición en SQL Server
Según la documentación de Microsoft y recursos relacionados, en SQL Server las transacciones pueden darse en distintos modos:
• **Modo Autocommit:** Es el modo por defecto en SQL Server. Cada instrucción individual DML (INSERT, UPDATE, DELETE) se trata como una transacción completa si no se especifica lo contrario.
• **Modo Explícito:** El desarrollador usa BEGIN TRANSACTION, luego ejecuta varias instrucciones, y finalmente emplea COMMIT o ROLLBACK.
• **Modo Implícito:** Cuando se habilita este modo (SET IMPLICIT_TRANSACTIONS ON), cada sentencia modifica los datos inicia automáticamente una transacción, que debe finalizarse explícitamente.

Ventajas del uso de transacciones
1. *Atomicidad y Manejo de Errores*: Permiten agrupar operaciones en un solo bloque lógico. Si alguna operación falla, se puede usar ROLLBACK para revertir todo el bloque al estado inicial, evitando cambios parciales. Esto simplifica la lógica de error en el código.
2. *Consistencia de Datos*: Garantizan que la base de datos pase de un estado válido a otro, respetando las
restricciones, la integridad referencial y las reglas de negocio.
3. *Aislamiento de Operaciones:* Evitan que los cambios no confirmados de una transacción interfieran o sean vistos incorrectamente por otras transacciones concurrentes. Esto reduce problemas como lecturas "sucias","no repetibles" o "fantasma".
4. *Durabilidad de los cambios confirmados*: Una vez que una transacción hace COMMIT, los cambios son permanentes incluso ante fallas del sistema o reinicios. Esto da confianza en que los datos no “se pierdan” tras un error crítico.

5. *Mejora de la integridad en operaciones complejas*: En procesos que involucran múltiples tablas, pasos o sistemas, las transacciones facilitan que todos esos pasos se consideren como uno solo: o se guardan todos o ninguno. Esto es especialmente útil en escenarios de negocio “complicados”.
6. *Facilitación del manejo de errores y recuperación:* Si ocurre un error durante la ejecución de un conjunto de operaciones dentro de una transacción, se puede usar ROLLBACK para retroceder al estado anterior y asegurar que los datos no queden “a medias”. Esto simplifica la lógica de error en el código.

Desventajas de las transacciones
1. *Sobrecarga de rendimiento:* Mantener las características ACID, los locks, los logs de transacción, validación y recuperación añade coste en tiempo-procesamiento y recursos.
2. *Contención y bloqueo*: Si la transacción abarca muchas operaciones o dura mucho tiempo, puede bloquear recursos durante su ejecución, lo que reduce concurrentes y provoca esperas.
3. *Problemas en sistemas de alta concurrencia o distribuidos*: En ambientes con muchas transacciones paralelas o bases de datos distribuidas, la simple transacción plana puede no escalar bien sin diseño adicional.
4. *Duración limitada y dificultad en operaciones prolongadas*: Transacciones muy largas pueden generar bloqueos, retener conexiones o generar fallos más costosos al hacer rollback.
5. *Trade-offs en aislamiento vs rendimiento:* Escoger un nivel alto de aislamiento mejora integridad pero reduce rendimiento; escoger uno bajo mejora rendimiento pero incrementa riesgo de anomalías (lecturas sucias, no repetibles, etc.).

¿Cómo funciona?
1º Inicia con BEGIN TRANSACTION (o la sintaxis equivalente del motor de BD) marca el inicio de la transacción.
2º Ejecuta varias operaciones SQL (por ejemplo: actualizar una tabla, insertar en otra, borrar en una tercera,etc.).
3º COMMIT si todo ha ido correctamente, confirmar los cambios: todas las operaciones pasan a estado “permanente”.
4º ROLLBACK si ha ocurrido un error (una restricción violada, falta de espacio, problema de red, etc.), entonces deshacer todos los cambios realizados por la transacción, dejando la base de datos como estaba al inicio de la transacción.

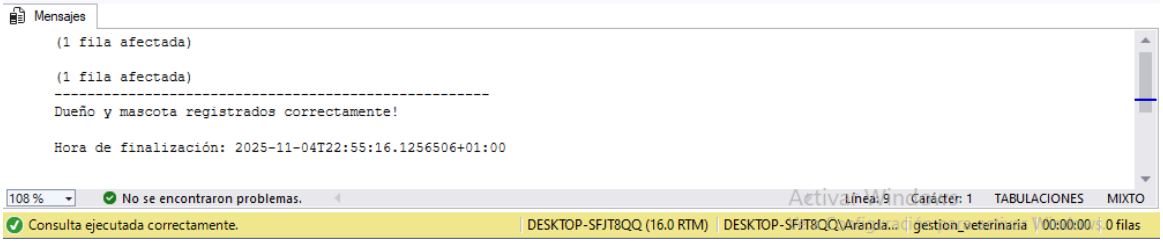
CASO PRÁCTICO 1: Registrar una nueva mascota y su dueño

Supongamos que llega un cliente nuevo con su mascota.Primero se intenta registrar al dueño en la tabla Duenio; si la inserción tiene éxito, se obtiene el ID generado (@id_dueno) con SCOPE_IDENTITY(). Luego se inserta la mascota en la tabla Mascota, vinculándola con ese dueño mediante id_dueno.Todo esto ocurre dentro de una única transacción, lo que significa que ambas operaciones deben completarse correctamente para que se confirme (COMMIT TRANSACTION). Si ocurre un error en cualquiera de las dos inserciones por ejemplo, si el dni, teléfono o email del dueño ya existen y violan una restricción UNIQUE, o si el id_raza no existe y genera un error de clave foránea el bloque CATCH se activa. En ese caso, se ejecuta un ROLLBACK TRANSACTION, que revierte todas las operaciones

realizadas dentro de la transacción, eliminando también al dueño recién insertado. Esto garantiza que no queden registros huérfanos ni inconsistentes (por ejemplo, una mascota sin dueño asociado).

Transacción exitosa

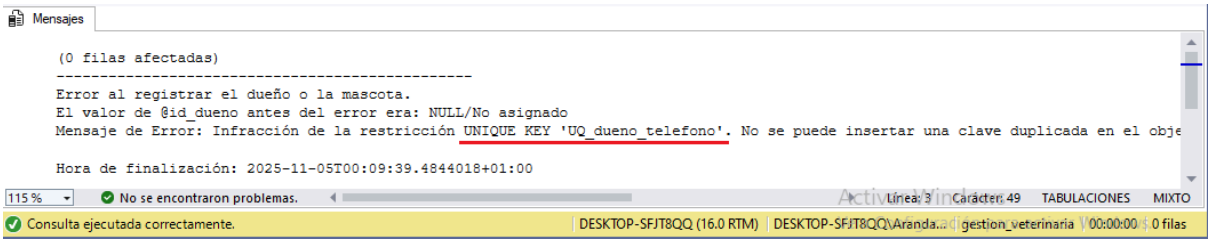
```sql
DECLARE @id_duenio INT; -- Se declara una variable que guarda el ID del dueño recién insertado.

BEGIN TRANSACTION;  --Inicio de la transaccion
BEGIN TRY
    -- Insertar Dueño
    INSERT INTO duenio (nombre_duenio, apellido_duenio, dni_duenio, telefono_duenio, email_duenio, direccion_duenio)
    VALUES ('Ana', 'López', '59321213', 3794935410, 'ana6699L@gmail.com', 'San Martín 123');

    SET @id_duenio = SCOPE_IDENTITY(); -- Se asigna el ID del Dueño recién insertado

    -- Insertar Mascota
    INSERT INTO mascota (nombre_mascota, fecha_nacimiento, peso_mascota, condicion_mascota, id_duenio, id_raza)
    VALUES ('Toby', '2020-06-15', 5.8, 'Sano', @id_duenio, 2);

    -- Si ambas inserciones se ejecutan sin errores, la transacción se confirma.
    COMMIT TRANSACTION;
    PRINT '-------------------------------------------------------';
    PRINT 'Dueño y mascota registrados correctamente!';
END TRY
BEGIN CATCH
    -- El ROLLBACK revierte toda la transaccion
    ROLLBACK TRANSACTION;
    PRINT '-------------------------------------------------------';
    PRINT 'Error al registrar el dueño o la mascota.';
    -- Si falló antes de asignar el dueño, mostrará "NULL/No asignado".
    PRINT 'El valor de @id_dueno antes del error era: ' + ISNULL(CAST(@id_duenio AS VARCHAR(50)), 'NULL/No asignado');
    PRINT 'Mensaje de Error: ' + ERROR_MESSAGE();  -- Muestra el mensaje de error SQL

END CATCH;
GO
```



La imagen muestra la correcta inserción del registro de una mascota y su respectivo dueño

Transacción fallida

```sql
DECLARE @id_duenio INT; -- Se declara una variable que guarda el ID del dueño recién insertado.

BEGIN TRANSACTION;  --Inicio de la transaccion
BEGIN TRY
    -- Insertar Dueño
    INSERT INTO duenio (nombre_duenio, apellido_duenio, dni_duenio, telefono_duenio, email_duenio, direccion_duenio)
    VALUES ('Juan', 'Perez', '43948978', 3794935410, 'juannPerez@gmail.com', 'San Martín 123');

    SET @id_duenio = SCOPE_IDENTITY(); -- Se asigna el ID del Dueño recién insertado

    -- Insertar Mascota
    INSERT INTO mascota (nombre_mascota, fecha_nacimiento, peso_mascota, condicion_mascota, id_duenio, id_raza)
    VALUES ('Yacki', '2020-02-5', 6.8, 'Sano', @id_duenio, 2);

    -- Si ambas inserciones se ejecutan sin errores, la transacción se confirma.
    COMMIT TRANSACTION;
    PRINT '-------------------------------------------------------';
    PRINT 'Dueño y mascota registrados correctamente!';
END TRY
BEGIN CATCH
    -- El ROLLBACK revierte toda la transaccion
    ROLLBACK TRANSACTION;
    PRINT '-------------------------------------------------------';
    PRINT 'Error al registrar el dueño o la mascota.';
    -- Si falló antes de asignar el dueño, mostrará "NULL/No asignado".
    PRINT 'El valor de @id_dueno antes del error era: ' + ISNULL(CAST(@id_duenio AS VARCHAR(50)), 'NULL/No asignado');
    PRINT 'Mensaje de Error: ' + ERROR_MESSAGE();  -- Muestra el mensaje de error SQL

END CATCH;
GO
```



En este caso, la transacción falló porque el valor ingresado en el campo telefono_dueno viola una restricción UNIQUE definida en la tabla Duenio.Esto significa que el número de teléfono ya existe en otro registro y no puede repetirse. Al detectarse esta violación, SQL Server genera un error, el control pasa al bloque CATCH y se ejecuta el ROLLBACK, revirtiendo tanto la inserción del dueño como la de la mascota para mantener la integridad de los datos.

Transacciones animadas

Una transacción anidada es una transacción que se inicia dentro del contexto de otra transacción ya activa.

Ventajas de las transacciones anidadas

1. *Modularización y Mantenimiento:* Permiten dividir un gran proceso transaccional en sub-transacciones,
haciendo el diseño más manejable y mejorando la legibilidad y el mantenimiento del código.
2. *Recuperación más Fina (Parcial):* En algunos modelos (aunque no es el comportamiento puro de SQL Server), si una sub-transacción falla, la transacción padre puede decidir revertir solo esa parte sin abortar todo el proceso principal.
3. *Reutilización y Encapsulamiento*: Los componentes que ejecutan sub-transacciones pueden diseñarse, probarse y reutilizarse más fácilmente, ya que están encapsulados dentro de su propia lógica transaccional hija.
4. *Mejor modelado de procesos largos o compuestos*: En escenarios complejos (por ejemplo workflows de negocio extensos, procesos distribuidos), las transacciones anidadas permiten estructurar mejor la lógica sin violar el principio de que la operación global sea atómica.

Desventajas de las transacciones anidadas

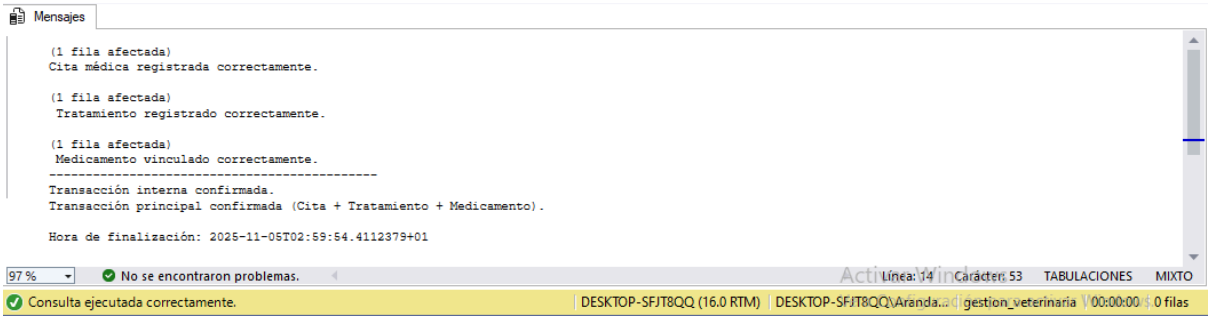
1. *Complejidad Operacional:* Introducen complejidad en el diseño y seguimiento, pues se debe manejar correctamente la jerarquía padre/hija, los commit y rollback.
2. *Soporte Limitado y Riesgo de Falsas Expectativas:* No todos los SGBD las implementan de forma completa. Esto puede llevar al desarrollador a tener falsas expectativas sobre el comportamiento de COMMIT y ROLLBACK de las sub-transacciones.
3. *Impacto en el Rendimiento:* La gestión de múltiples niveles de transacción implica una sobrecarga adicional (más locks, más control de estado y mayor uso de logs).
4. *Difícil depuración y testing:* Las transacciones anidadas pueden hacer que el flujo de control sea más difícil de seguir y verificar: por ejemplo, ¿qué sucede cuando falla una sub-transacción?, ¿se revierte sólo ella?, ¿cómo se comunica al padre?, etc. Además, en entornos de prueba puede ser más difícil replicar el comportamiento real del sistema en producción.

¿Cómo funciona en SQL Server?
Cada vez que haces BEGIN TRANSACTION incrementa un contador @@TRANCOUNT. Cuando haces COMMIT TRANSACTION, este decrementa @@TRANCOUNT. Solo cuando @@TRANCOUNT llega a 0 se considera que la transacción realmente se confirma. Si haces ROLLBACK TRANSACTION en cualquier nivel (externo o interno) se revierte todo: la transacción exterior e interior. En SQL Server se dice que existe “apariencia” de transacciones anidadas, pero en esencia es un bloque único con contador de niveles

CASO PRÁCTICO 2: Registrar una cita + tratamiento + medicamento

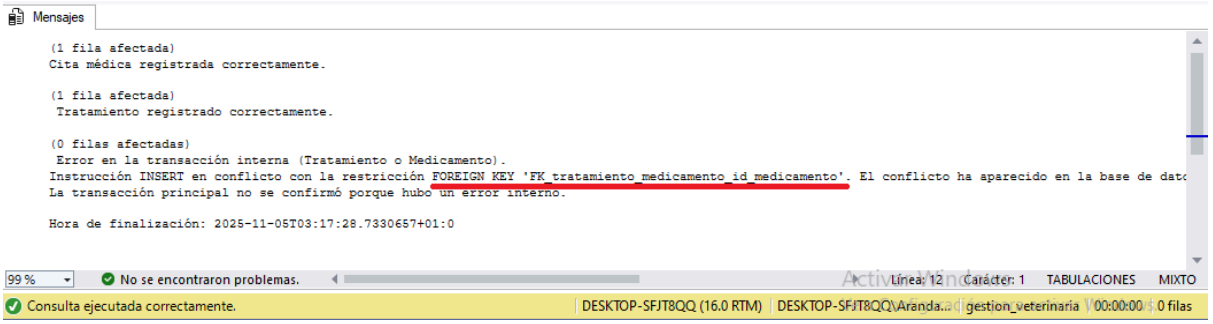
Transacción anidada exitosa

```sql
DECLARE @id_cita INT;        -- Variable para guardar el ID generado de la cita médica
DECLARE @id_tratamiento INT; -- Variable para guardar el ID generado del tratamiento

-- NIVEL 1: Transacción Principal (Registro de la Cita)
BEGIN TRANSACTION CitaPrincipal;  -- Inicia la transacción principal. Todo lo que ocurra adentro depende de ella.
BEGIN TRY   -- Inicia un bloque de control de errores TRY/CATCH
    --  Registrar una cita médica
    INSERT INTO citas_medica (fecha_citaMedica, observaciones_citaMedica, usuario, motivo_visita, id_mascota, id_veterinario)
    VALUES (GETDATE(), ' dolor de cabeza', SYSTEM_USER, 'Dolor persistente', 7, 1);

    SET @id_cita = SCOPE_IDENTITY();     -- Guarda el ID recién generado de la cita
    PRINT 'Cita médica registrada correctamente.';

    -- NIVEL 2: Transacción anidada (Registro de Tratamiento y Medicamento)
    BEGIN TRANSACTION TratamientoAnidado;  -- Abre una nueva transacción dentro de la principal (incrementa @@TRANCOUNT)
    BEGIN TRY
        --   Registrar el tratamiento vinculado a la cita
        INSERT INTO tratamiento (nombre_tratamiento, inicio_tratamiento, fin_tratamiento, id_citaMedica)
        VALUES ('Tratamiento con analgésicos', '2025-11-03', '2025-12-03', @id_cita);

        SET @id_tratamiento = SCOPE_IDENTITY(); --Guarda el ID generado del tratamiento
        PRINT ' Tratamiento registrado correctamente.';

        -- Registrar un tratamiento_medicamento, Forzamos un error (medicamento inexistente)
        INSERT INTO tratamiento_medicamento (id_medicamento, id_tratamiento)
        VALUES (1, @id_tratamiento);

        PRINT ' Medicamento vinculado correctamente.';
        PRINT '-------------------------------------------';

        -- Si todo salió bien dentro del bloque anidado:
        COMMIT TRANSACTION TratamientoAnidado;  -- Cierra la transacción interna
        PRINT 'Transacción interna confirmada.';
    END TRY
    BEGIN CATCH
        -- Si ocurre un error dentro de la transacción anidada:
        PRINT ' Error en la transacción interna (Tratamiento o Medicamento).';
        PRINT ERROR_MESSAGE();  -- Muestra el mensaje de error SQL
        ROLLBACK TRANSACTION;   -- Este ROLLBACK deshace TODO (no solo el bloque anidado), porque SQL Server no permite revertir
    END CATCH;

     -- Si todo salió bien hasta acá, se confirma la transacción principal
    IF @@TRANCOUNT > 0        -- Verifica si aún queda una transacción abierta
    BEGIN
        COMMIT TRANSACTION CitaPrincipal;   -- Confirma todo (Cita + Tratamiento + Medicamento)
        PRINT 'Transacción principal confirmada (Cita + Tratamiento + Medicamento).';
    END
    ELSE
    BEGIN -- Si hubo error en la interna, @@TRANCOUNT ya es 0
        PRINT 'La transacción principal no se confirmó porque hubo un error interno.';
    END
END TRY
BEGIN CATCH
    -- Si ocurre un error fuera del bloque interno (en la cita)
    PRINT 'Error en la transacción principal.';
    PRINT ERROR_MESSAGE();    -- Muestra el error SQL exacto
    ROLLBACK TRANSACTION;   -- Revierte todo lo hecho
    PRINT ' Se revirtió toda la transacción.';
END CATCH;
GO
```



Transacción anidada fallida

```sql
DECLARE @id_cita INT;        -- Variable para guardar el ID generado de la cita médica
DECLARE @id_tratamiento INT; -- Variable para guardar el ID generado del tratamiento

-- NIVEL 1: Transacción Principal (Registro de la Cita)
BEGIN TRANSACTION CitaPrincipal;  -- Inicia la transacción principal. Todo lo que ocurra adentro depende de ella.
BEGIN TRY   -- Inicia un bloque de control de errores TRY/CATCH
    --  Registrar una cita médica
    INSERT INTO citas_medica (fecha_citaMedica, observaciones_citaMedica, usuario, motivo_visita, id_mascota, id_veterinario)
    VALUES ('2025-11-03', ' dolor de cabeza', 'admin', 'Dolor persistente', 7, 1);

    SET @id_cita = SCOPE_IDENTITY();     -- Guarda el ID recién generado de la cita
    PRINT 'Cita médica registrada correctamente.';

    -- NIVEL 2: Transacción anidada (Registro de Tratamiento y Medicamento)
    BEGIN TRANSACTION TratamientoAnidado;  -- Abre una nueva transacción dentro de la principal (incrementa @@TRANCOUNT)
    BEGIN TRY
        --   Registrar el tratamiento vinculado a la cita
        INSERT INTO tratamiento (nombre_tratamiento, inicio_tratamiento, fin_tratamiento, id_citaMedica)
        VALUES ('Tratamiento con analgésicos', '2025-11-03', '2025-12-03', @id_cita);

        SET @id_tratamiento = SCOPE_IDENTITY(); --Guarda el ID generado del tratamiento
        PRINT ' Tratamiento registrado correctamente.';

        -- Registrar un tratamiento_medicamento, Forzamos un error (medicamento inexistente)
        INSERT INTO tratamiento_medicamento (id_medicamento, id_tratamiento)
        VALUES (999, @id_tratamiento);

        PRINT ' Medicamento vinculado correctamente.';
         PRINT '-----------------------------------------------';

        -- Si todo salió bien dentro del bloque anidado:
        COMMIT TRANSACTION TratamientoAnidado;  -- Cierra la transacción interna
        PRINT 'Transacción interna confirmada.';
    END TRY
    BEGIN CATCH
        -- Si ocurre un error dentro de la transacción anidada:
        PRINT ' Error en la transacción interna (Tratamiento o Medicamento).';
        PRINT ERROR_MESSAGE();  -- Muestra el mensaje de error SQL
        ROLLBACK TRANSACTION;   -- Este ROLLBACK deshace TODO (no solo el bloque anidado), porque SQL Server no permite revertir
    END CATCH;

    -- Si todo salió bien hasta acá, se confirma la transacción principal
    IF @@TRANCOUNT > 0   -- Verifica si aún queda una transacción abierta
    BEGIN
        COMMIT TRANSACTION CitaPrincipal;  -- Confirma todo (Cita + Tratamiento + Medicamento)
        PRINT 'Transacción principal confirmada (Cita + Tratamiento + Medicamento).';
    END
    ELSE
    BEGIN-- Si hubo error en la interna, @@TRANCOUNT ya es 0
        PRINT 'La transacción principal no se confirmó porque hubo un error interno.';
    END
END TRY
BEGIN CATCH
    -- Si ocurre un error fuera del bloque interno (en la cita)
    PRINT 'Error en la transacción principal.';
    PRINT ERROR_MESSAGE();    -- Muestra el error SQL exacto
    ROLLBACK TRANSACTION;   -- Revierte todo lo hecho
    PRINT 'Se revirtió toda la transacción.';
END CATCH;
GO
```



La transacción falla porque, aunque la cita médica y el tratamiento se insertan correctamente, al intentar registrar el medicamento en la tabla Tratamiento_Medicamento se usa un id_medicamento que no existe en la tabla Medicamento. Esto viola la restricción de clave foránea, genera un error en la transacción interna y provoca que toda la transacción tanto la interna como la principal sea revertida completamente para mantener la integridad de los datos.

Conclusión

En conclusión, el uso de transacciones y transacciones anidadas es fundamental para asegurar la coherencia y confiabilidad de los datos en sistemas donde múltiples operaciones deben ejecutarse de forma coordinada. Las transacciones simples permiten que un bloque de operaciones funcione como una unidad indivisible: si todo sale bien se confirma, y si ocurre un error se revierte por completo, evitando información parcial o corrupta. Las transacciones anidadas amplían este concepto permitiendo dividir un proceso grande en subprocesos independientes, facilitando la claridad lógica, el control por etapas y la detección de errores dentro de cada parte del flujo. No obstante, su uso requiere conocimiento del comportamiento real del motor de base de datos: en SQL Server, los commits internos no confirman nada definitivamente y los rollbacks internos pueden revertir toda la transacción externa, lo que significa que deben implementarse con cuidado,con manejo estructurado de errores y una planificación clara. Bien utilizadas, las transacciones y las transacciones anidadas aportan robustez, orden y confiabilidad al manejo de datos en aplicaciones complejas.