Tema 2: Manejo de Transacciones y Transacciones Anidadas

Este tema posee objetivos principales para el aprendizaje:

- Entender la consistencia y atomicidad de las transacciones en base de datos
- Implementar transacciones simples y anidadas para garantizar la integridad de los datos

Transacciones

Una transacción es una unidad única de trabajo que agrupa una o varias instrucciones en una base de datos. Las transacciones pueden consistir en una sola operación de lectura, escritura, eliminación o actualización, o una combinación de estas. Si una transacción tiene éxito, todas las modificaciones de los datos realizadas durante la transacción se confirman y se convierten en una parte permanente de la base de datos. Si una transacción encuentra errores y debe cancelarse o revertirse, se borran todas las modificaciones de los datos.

Su objetivo es asegurar la integridad de los datos, cumpliendo las propiedades **ACID**:

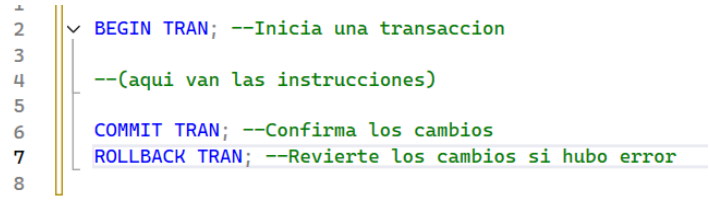
- **Atomicidad:** Todas las operaciones incluidas en la transacción se realizan correctamente. De lo contrario, todas las operaciones se cancelan en el punto de fallo y se revierten todas las operaciones anteriores.
- **Consistencia**: Esta propiedad garantiza que todos los datos serán consistentes después de que se complete una transacción de acuerdo con las reglas, restricciones, cascadas y desencadenantes definidos.
- **Aislamiento**: Todas las transacciones están aisladas de otras transacciones.
- **Duradero**: La modificación de las transacciones confirmadas se vuelve persistente en la base de datos.

Modos de transacciones en SQL Server:

SQL Server puede operar en 3 modos de transacción diferentes, que son los siguientes:

Modos de transacciones	Descripción
Transacciones de confirmación automática	Cada instrucción individual es una transacción.
Transacciones explicitas	Cada transacción se inicia explícitamente con la instrucción **BEGIN TRANSACTION** y finaliza explícitamente con una instrucción **COMMIT** o **ROLLBACK**.
Transacciones implícitas	Una nueva transacción se inicia implícitamente cuando se completa la transacción anterior, pero cada transacción se completa explícitamente con una instrucción **COMMIT** o **ROLLBACK**.

Comandos básicos para manejar transacciones:



```
2  BEGIN TRAN; --Inicia una transaccion
3
4  --(aqui van las instrucciones)
5
6  COMMIT TRAN; --Confirma los cambios
7  ROLLBACK TRAN; --Revierte los cambios si hubo error
8
```

Antes de que comience el procesamiento, la **BEGIN TRAN** instrucción notifica a SQL Server que trate todas las acciones siguientes como una sola transacción.

Si se produce algún error en alguna de las instrucciones agrupadas, todos los cambios deben cancelarse. El proceso de revertir cambios se denomina **ROLLBACK** en SQL Server. Si todas las instrucciones de una misma transacción son correctas, todos los cambios se registran conjuntamente en la base de datos. En SQL Server, decimos que estos cambios se confirman con un **COMMIT** en la base de datos.

Como definir una transacción implícita

En SQL Server, una transacción implícita es un modo de operación donde SQL Server inicia automáticamente una nueva transacción cuando se ejecuta la siguiente instrucción que modifica datos (**INSERT, UPDATE, DELETE, MERGE**) o una sentencia que requiera transacción (por ejemplo, **ALTER o CREATE**).

Esta modalidad es útil cuando se desea trabajar con transacciones sin tener que escribir explícitamente **BEGIN TRAN**, aunque sí es necesario confirmar o revertir los cambios manualmente con **COMMIT** o **ROLLBACK**.

Las transacciones implícitas se activan por sesión o conexión, por lo que, si la conexión se cierra antes de un **COMMIT**, SQL Server hará **ROLLBACK** automático para evitar dejar la base en un estado inconsistente.

Para definir una transacción implícita, debemos habilitar la opción **IMPLICIT_TRANSACTIONS.**

```
10  SET IMPLICIT_TRANSACTIONS ON
11  UPDATE ZonaRiego
12  --Aqui debemos poner las instrucciones
13
14  COMMIT TRAN;
```

COMMIT TRANSACTION hace que los cambios realizados dentro de la transacción sean permanentes.

Si no se ejecuta un **COMMIT**, se recomienda usar **ROLLBACK TRANSACTION** explícito; sin embargo, si la sesión finaliza inesperadamente, SQL Server revertirá los cambios automáticamente.

Como definir una transacción explícita

Tras definir una transacción explícita mediante el comando **BEGIN TRANSACTION**, los recursos relacionados adquieren un bloqueo según el nivel de aislamiento de la transacción. Por este motivo, utilizar la transacción más corta posible ayudará a reducir los problemas de bloqueo.

Como se indicó en la sección anterior, la instrucción **COMMIT TRAN** aplica los cambios a la base de datos, y estos se vuelven permanentes. Ahora, completemos la transacción abierta con una instrucción **COMMIT TRAN.**

Por otro lado, la instrucción **ROLLBACK TRANSACTION** permite deshacer todas las modificaciones de datos aplicadas por la transacción.

```
BEGIN TRAN;

    UPDATE CUENTA
    SET saldo = saldo - 1000
    WHERE cuenta_id = 4;

COMMIT TRAN;
ROLLBACK TRAN;
```

Ejemplo de transacción explicita

Transacciones Anidadas

SQL Server permite anidar transacciones. Esto significa que una nueva transacción puede comenzar, aunque la anterior no haya finalizado. Transact-SQL permite anidar operaciones de transacción mediante **BEGIN TRAN** comandos anidados. **@@TRANCOUNT** Se puede consultar la variable automática para determinar el nivel de anidamiento: 0 indica que no hay anidamiento, 1 indica un nivel de anidamiento, y así sucesivamente.

Una operación **COMMIT** emitida contra cualquier transacción, excepto la más externa, no guarda ningún cambio en el disco; simplemente decrementa la **@@TRANCOUNT** variable automática. **ROLLBACK** En cambio, una operación funciona independientemente del nivel en el que se emita, pero revierte todas las transacciones, sin importar el nivel de anidamiento. Aunque parezca contradictorio, hay una razón de peso para ello. Si una transacción anidada **COMMIT** escribiera cambios de forma permanente en el disco, una **ROLLBACK** transacción externa no podría revertirlos, ya que quedarían registrados permanentemente.

Al iniciar explícitamente una transacción, el **@@TRANCOUNT** contador automático de variables aumenta de 0 a 1; al ejecutar **COMMIT**, el contador disminuye en uno; y al ejecutar **ROLLBACK**, el contador se reduce a 0. Como se observa, el comportamiento de **COMMIT** y **ROLLBACK** no es simétrico. Si se anidan transacciones, **COMMIT** siempre disminuye el nivel de anidamiento en 1, como se ilustra en la *Figura 1*. El comando **ROLLBACK**, por otro lado, revierte toda la transacción, como se ilustra en la *Figura 2.* Esta asimetría entre **COMMIT** y **ROLLBACK** es clave para gestionar los errores en transacciones anidadas.

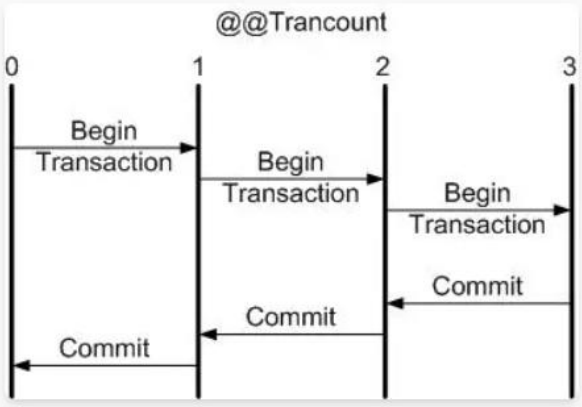


Figura 1: Un COMMIT siempre equilibra un BEGIN TRANSACTION reduciendo el recuento de transacciones en uno.

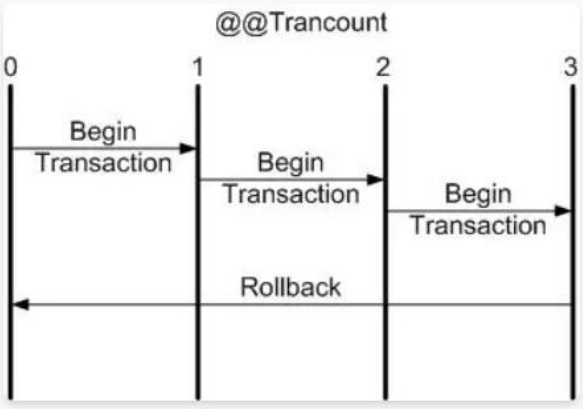


Figura 2: Un único ROLLBACK siempre revierte toda la transacción.

Ejemplo:

```
1   BEGIN TRAN; --externa
2
3       --instruccion 1
4
5       BEGIN TRAN; --interna(anidada)
6
7           --instruccion 2
8
9       COMMIT TRAN; --interna
10
11
12  COMMIT TRAN; --externa(confirma todo)
13  -- Si algo no va bien, entonces usar solo: ROLLBACK TRAN;
14
```

Manejo de errores un T-SQL

Los errores son inevitables al trabajar con sistemas de bases de datos, y T-SQL no es la excepción. El manejo eficiente de errores en T-SQL ha evolucionado con el tiempo y es crucial para garantizar el correcto funcionamiento y la integridad de los datos dentro de una base de datos. Esto implica **detectar errores durante la ejecución de instrucciones o lotes T-SQL**, impedir efectos no deseados y **definir acciones correctivas o alternativas** de manera controlada.

Tradicionalmente, el manejo de errores en SQL Server dependía del análisis del valor devuelto por funciones como @@ERROR inmediatamente después de cada operación, lo que hacía el proceso **poco elegante, repetitivo y propenso a omisiones**. Con el tiempo, SQL Server incorporó mecanismos más robustos, especialmente orientados a entornos transaccionales donde es necesario asegurar que **una falla parcial no afecte la consistencia general**.

El enfoque moderno y recomendado para el manejo de errores en T-SQL utiliza **bloques TRY…CATCH**, similares a los de lenguajes estructurados de programación como C#, Java o Python. Dentro del bloque TRY se coloca el código susceptible de

producir un error, mientras que en el bloque CATCH se definen las acciones a ejecutar si dicho error ocurre, tales como registrar información de auditoría, revertir transacciones, mostrar mensajes personalizados o finalizar un proceso de manera controlada.

Dentro del bloque CATCH es posible acceder a funciones del sistema que devuelven información sobre el error ocurrido, tales como:

- ERROR_NUMBER() → número identificador del error
- ERROR_MESSAGE() → descripción del error
- ERROR_LINE() → línea donde ocurrió
- ERROR_PROCEDURE() → procedimiento almacenado (si aplica)
- ERROR_STATE() y ERROR_SEVERITY() → información interna del error

De esta forma, el manejo de errores se vuelve **centralizado, legible, reutilizable y seguro**, lo que contribuye a la estabilidad de las operaciones críticas, como procesos de carga, transacciones financieras, actualizaciones masivas o lógica de negocio implementada con stored procedures.

```
BEGIN TRY;
    BEGIN TRAN;

        --instruccion 1

    COMMIT TRAN;
END TRY
BEGIN CATCH
    ROLLBACK TRAN;
    --Podemos enviar un mensaje de error
    PRINT 'Error detectado. Se revirtiron los cambios.';
    --Si quisieramos mas informacion, podriamos agregar estas funciones
    SELECT
        ERROR_NUMBER()    AS NumeroError,
        ERROR_MESSAGE()   AS DescripcionError,
        ERROR_LINE()      AS LineaDelError,
        ERROR_PROCEDURE() AS Procedimiento,
        ERROR_STATE()     AS Estado,
        ERROR_SEVERITY()  AS Severidad;
END CATCH;
```

Tarea N° 1: Escribir el código T-SQL que permita definir una transacción consistente en: **Insertar** un registro en alguna tabla, luego otro registro en otra tabla y por último la **actualización** de uno o mas registros en otra tabla. Actualizar los datos solamente si toda la operación es completada con éxito.

Instrucción:

```
6
7   BEGIN TRY;
8       BEGIN TRAN;
9           INSERT INTO RegistroRiego(volumen_agua,id_zona_registro,resultado)
10          VALUES (288.70,1,1);
11
12          INSERT INTO LecturasSensor(humedad,temperatura,nivel_bateria,id_sensor_lectura)
13          VALUES(70.5,30.33,80,3);
14
15          UPDATE ConfiguracionesRiego
16              SET duracion_riego = 30
17          WHERE id_config = 5;
18
19      COMMIT TRAN;
20      PRINT 'Operacion completada con exito';
21  END TRY
22  BEGIN CATCH
23      ROLLBACK TRAN;
24      PRINT 'Error detectado. Se revirtieron los cambios.';
25  END CATCH;
26  GO
```

No se encontraron problemas.

Mensajes

```
(1 fila afectada)

(1 fila afectada)

(1 fila afectada)
Operacion completada con exito

Hora de finalización: 2025-11-17T00:05:49.6264165-03:00
```

Antes de hacer algún cambio podemos ver la cantidad de registros de cada tabla:

Resultados | Mensajes

	Resultados
1	3

	ZonaRiego
1	135

	ConfiguracionesRiego
1	135

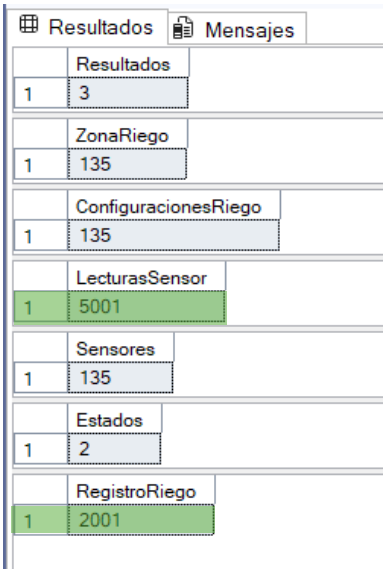
	LecturasSensor
1	5000

	Sensores
1	135

	Estados
1	2

	RegistroRiego
1	2000

Después de insertar nuevos registros a las tablas de **RegistroRiego** y **LecturasSensor** podemos ver como aumento sus registros:



Resultados | Mensajes

	Resultados
1	3

	ZonaRiego
1	135

	ConfiguracionesRiego
1	135

	LecturasSensor
1	5001

	Sensores
1	135

	Estados
1	2

	RegistroRiego
1	2001

Y la tabla que vamos a actualizar será **ConfiguracionesRiego** más preciso la columna de **duración_riego** que actualmente es de 21 minutos debería actualizarse a 30 minutos donde el **id_config** es igual a 5:

Antes de la actualización:

Resultados | Mensajes

	id_config	humedad_minima	duracion_riego	frecuencia_lectura	fecha_actualizacion	id_zona_cor
1	1	42,39	46	14	2025-11-16	42
2	2	34,27	36	18	2025-11-16	37
3	3	32,38	42	11	2025-11-16	123
4	4	44,58	26	9	2025-11-16	102
5	5	34,14	21	15	2025-11-16	21
6	6	21,63	46	5	2025-11-16	89

Después de la actualización:

Resultados | Mensajes

	id_config	humedad_minima	duracion_riego	frecuencia_lectura	fecha_actualizacion	id_zona_config
1	1	42,39	46	14	2025-11-16	42
2	2	34,27	36	18	2025-11-16	37
3	3	32,38	42	11	2025-11-16	123
4	4	44,58	26	9	2025-11-16	102
5	5	34,14	30	15	2025-11-16	21
6	6	21,63	46	5	2025-11-16	89
7	7	22,38	26	15	2025-11-16	18

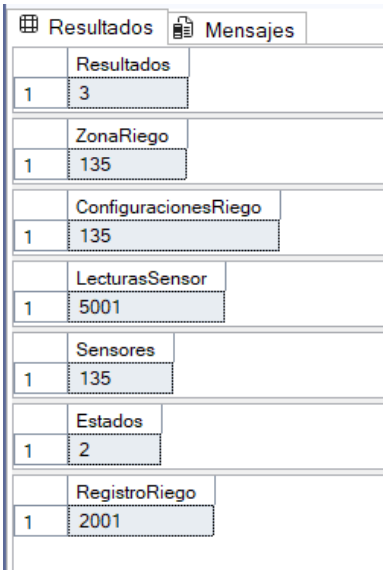
También podemos ver el mensaje que la transacción fue un éxito:

```
(1 fila afectada)

(1 fila afectada)

(1 fila afectada)
Operacion completada con exito

Hora de finalización: 2025-11-17T00:05:49.6264165-03:00
```

Tarea N° 2: Sobre el código escrito anteriormente provocar intencionalmente un error luego del **insert** y verificar que los datos queden consistentes (No se debería realizar ningún **insert**).

Para este caso intentaremos insertar nuevos registros a las tablas de **RegistroRiego** y **LecturasSensor** e intentaremos una nueva actualización a la tabla **ConfiguracionesRiego**, pero no se debería realizar ningún cambio ya que provocaremos intencionalmente un error.

Tener en cuenta los registros actualizados en el ejercicio anterior:



Instrucción:

```
BEGIN TRY;
    BEGIN TRAN;
        INSERT INTO RegistroRiego(volumen_agua,id_zona_registro,resultado)
        VALUES (288.70,1,99);--Aqui deberia provocarse el error

        INSERT INTO LecturasSensor(humedad,temperatura,nivel_bateria,id_sensor_lectura)
        VALUES(80.5,30.33,80,3);

        UPDATE ConfiguracionesRiego
            SET duracion_riego = 20
        WHERE id_config = 4;

    COMMIT TRAN;
    PRINT 'Operacion completada con exito';
END TRY
BEGIN CATCH
    ROLLBACK TRAN;
    PRINT 'Error detectado. Se revirtiron los cambios.';
END CATCH;
GO
```

% No se encontraron problemas.

Mensajes

```
(0 filas afectadas)
Error detectado. Se revirtiron los cambios.

Hora de finalización: 2025-11-17T01:13:14.5625772-03:00
```

En esta instrucción podemos observar que en la tabla **RegistroRiego** mas preciso en la columna **resultado** estamos agregando el valor **99** y es este valor que provocara el error ya que en la tabla **Resultados** este valor no existe:

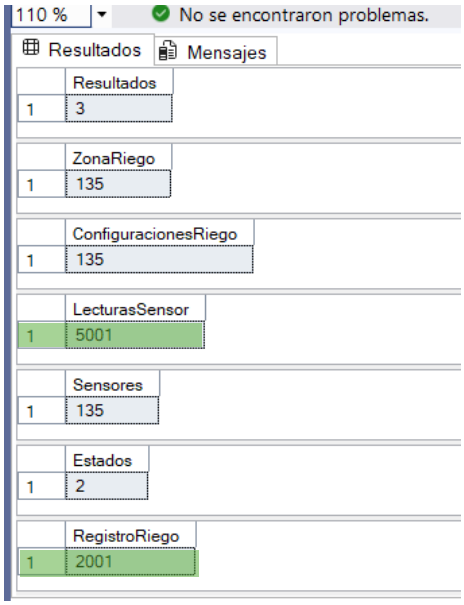
```
--TABLAS
SELECT * FROM Resultados;
SELECT * FROM ZonaRiego;
```

110 % 3 0

Resultados Mensajes

	id_resultado	nombre
1	1	Exitoso
2	2	Fallido
3	3	Cancelado

También podemos observar que no se insertó ningún registro nuevo:



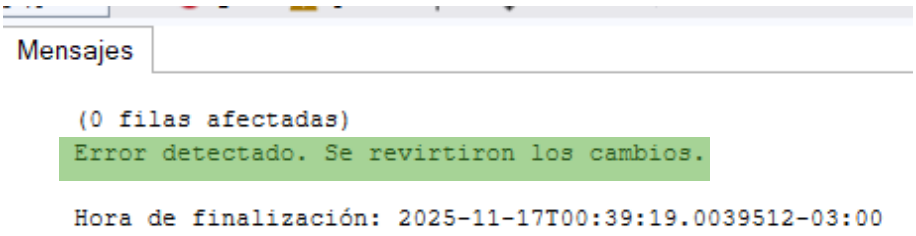
Y que no se realizo ninguna actualizacion en la tabla **ConfiguracionesRiego** en la columna de **duracion_riego** para el registro con id **4**:

Resultados Mensajes

	id_config	humedad_minima	duracion_riego	frecuencia_lectura	fecha_actualizacion	id_zona_config
1	1	42,39	46	14	2025-11-16	42
2	2	34,27	36	18	2025-11-16	37
3	3	32,38	42	11	2025-11-16	123
4	4	44,58	26	9	2025-11-16	102
5	5	34,14	30	15	2025-11-16	21
6	6	22,38	26	5	2025-11-16	89
7	7	22,38	26	15	2025-11-16	18

Haga clic para seleccionar toda la fila

También podemos observar en mensaje de error capturado por la sentencia:



Tarea N° 3: Expresar las conclusiones en base a las pruebas realizadas.

Durante las pruebas se comprobó que la transacción mantiene la atomicidad:

sí una instrucción falla, ninguna operación se ejecuta, garantizando la consistencia de los datos. El uso de **TRY**...**CATCH** permitió capturar el error y aplicar **ROLLBACK** de forma segura. Esto asegura que el sistema de riego inteligente no registre datos incompletos o corruptos, manteniendo su integridad y confiabilidad.