Universidad Nacional del Nordeste

Facultad de Ciencias Exactas y Naturales y Agrimensura

Sistema de Control de Riegos Inteligentes

Equipo 26

Alumnos: - Gálvez Díaz Colodrero, Mateo Nicolás

- Noir Aguilar, Nahuel Kevin

- Ojeda, Juan Agustín

- Sánchez Morales, Benjamín Delfor

Cátedra: Bases de Datos I

Profesor: Darío O. Villegas

Año: 2025

Índice

Capítulo I: Introducción

Tema

En este proyecto vamos a abordar la necesidad de gestionar de manera eficiente la información relacionada con el **riego agrícola**, permitiendo almacenar, analizar y consultar datos vinculados a sensores de humedad, caudal de agua, intervalos de riego y consumo total, con el fin de optimizar recursos hídricos y mejorar la productividad agrícola.

Definición del Problema

La producción agrícola en Corrientes, especialmente de cultivos como arroz, cítricos y yerba mate, entre otros, enfrenta dificultades derivadas de:

- La **variabilidad climática** (lluvias intensas, períodos de sequía, incendios).
- El **uso ineficiente del agua**, particularmente crítico en zonas donde se requiere riego artificial.
- La **falta de digitalización de datos de riego**, que dificulta el seguimiento de consumos, la detección de pérdidas y la programación eficiente de riegos.
- La **ausencia de registros centralizados** de sensores de humedad, caudalímetros y cronogramas de riego, lo que genera un desperdicio de recursos y disminuye la competitividad frente a otras regiones.

El problema central es la falta de un sistema informatizado y confiable que organice y relacione toda la información del riego agrícola, permitiendo a productores y técnicos agrónomos contar con datos precisos para programar riegos, mejorar la eficiencia en el uso del agua y prevenir pérdidas por exceso o déficit hídrico.

A lo largo del desarrollo de este proyecto buscaremos responder a las siguientes incógnitas:

- ¿Cómo puede una base de datos ayudar a registrar y analizar la información sobre riegos en las parcelas de Corrientes?
- ¿Qué beneficios traería digitalizar el control de consumo de agua y su distribución?
- ¿Cómo podría la información almacenada contribuir a la toma de decisiones frente a sequías o exceso de precipitaciones?

Objetivo

Objetivos Generales

Diseñar y modelar una base de datos relacional que soporta un **sistema de control de riegos inteligentes**, capaz de gestionar información sobre parcelas, sensores de humedad y caudal, cronogramas de riego y consumo de agua, con el fin de optimizar recursos y mejorar la eficiencia agrícola local.

Objetivos Específicos

1. Identificar las principales problemáticas en el manejo del agua en la Provincia de Corrientes.
2. Diseñar el modelo entidad-relación (MER) de la base de datos que contemple cultivos, parcelas, sensores, riego, etc.
3. Garantizar que el modelo de datos cumpla con todas las reglas de normalización.
4. Proveer una estructura que facilite la escalabilidad del sistema hacia la inclusión de nuevas tecnologías (IoT, predicciones climáticas).
5. Implementar consultas SQL que permitan analizar indicadores clave: consumo de agua por parcela, historial de riegos, eficiencia hídrica.
6. Entender la consistencia y atomicidad de las transacciones en bases de datos.
7. Implementar transacciones simples y anidadas para garantizar la integridad de los datos.
8. Comprender la diferencia entre procedimientos y funciones almacenadas.
9. Aplicar procedimientos y funciones en la implementación de operaciones CRUD.

10. Conocer los tipos de índices y sus aplicaciones.
11. Evaluar el impacto de los índices en el rendimiento de las consultas.
12. Conocer las técnicas de backup y restore, incluyendo backup en línea.
13. Implementar estrategias de respaldo para asegurar la integridad y recuperación de datos.

Capítulo II: Marco Conceptual

El presente trabajo se enmarca dentro de los avances tecnológicos aplicados a la agricultura, en particular en la gestión eficiente del agua mediante sistemas de riego inteligente. La agricultura es una de las principales actividades económicas de la provincia de Corrientes, y el uso adecuado de los recursos hídricos es clave para sostener la productividad y garantizar el desarrollo regional.

Desarrollo Sustentable

El uso del agua en la agricultura está directamente relacionado con el **desarrollo sustentable**. Un riego excesivo puede dañar los suelos y desperdiciar recursos, mientras que un riego insuficiente puede afectar la producción y calidad de los cultivos.

Por ello, el control inteligente del riego no solo busca mejorar la rentabilidad económica, sino también garantizar el cuidado del medio ambiente y el uso responsable de un recurso vital como el agua.

Innovaciones Tecnológicas en la Agricultura

En los últimos años, la incorporación de tecnologías de información y comunicación **(TICs)** ha permitido transformar la manera en que se producen los alimentos. Conceptos como agricultura de precisión o agricultura inteligente hacen referencia al uso de sensores, dispositivos automáticos y bases de datos que ayudan a los productores a tomar mejores decisiones.

El riego inteligente se enmarca dentro de esta tendencia, ya que utiliza sensores de humedad, caudalímetros y registros digitales para determinar **cuándo, cuánto y cómo regar** cada parcela.

Bases de Datos en el Ámbito Agrícola

La digitalización de la información agrícola es un paso necesario para mejorar la competitividad. Una **base de datos relacional** permite centralizar información de

diferentes fuentes (parcelas, sensores, registros de riego, condiciones ambientales), evitando la dispersión de datos y reduciendo errores.

De este modo, los productores y técnicos agrónomos pueden acceder rápidamente a la información histórica y actualizada, lo que contribuye a la optimización del agua y al ahorro de costos.

Globalización y Desarrollo Regional

En un contexto de globalización, la agricultura debe competir no solo a nivel local, sino también internacional. Para que la producción de Corrientes (arroz, cítricos, yerba mate, entre otros) mantenga su competitividad, resulta indispensable aplicar innovaciones tecnológicas que permitan aumentar la eficiencia y reducir las pérdidas.

Un sistema de control de riegos inteligentes favorece el desarrollo regional al hacer más eficiente la producción, generar empleos vinculados a la tecnología y promover la sustentabilidad en el uso de recursos naturales.

Este trabajo parte de la idea de que los sistemas informáticos aplicados al riego agrícola son una herramienta fundamental para enfrentar los desafíos actuales del sector. La integración de innovaciones tecnológicas, bases de datos relacionales y principios de sustentabilidad permite avanzar hacia una agricultura más eficiente, moderna y comprometida con el desarrollo regional de Corrientes.

Capítulo III: Metodología

El presente trabajo práctico se desarrolló siguiendo una serie de pasos que permitieron organizar la información, diseñar el modelo de base de datos y comprender cómo un sistema informatizado puede aportar soluciones concretas al problema del riego agrícola.

a. Descripción de cómo se realizó el Trabajo Práctico

En primer lugar, se analizó la problemática del riego en la agricultura de Corrientes, identificando factores como el uso excesivo de agua, la falta de registros confiables y la necesidad de digitalizar datos para mejorar la gestión de recursos.

Posteriormente, se definieron los objetivos del proyecto y se elaboró un Modelo Entidad-Relación (MER), representando gráficamente las entidades, sus atributos y relaciones, asegurando que la estructura de datos evita redundancias y garantizara la integridad de la información.

El siguiente paso consistió en la traducción del modelo relacional al modelo físico, es decir, en la elaboración del script SQL con la definición de tablas, claves primarias, claves foráneas y restricciones, una vez terminada la definición del modelo físico, pasamos a elaborar el diccionario de datos.

b. Herramientas (Instrumentos y procedimientos)

Para llevar adelante este trabajo se utilizaron las siguientes herramientas y procedimientos:

- **Revisión bibliográfica y digital**: se consultaron artículos sobre agricultura de precisión y bases de datos aplicadas al agro, disponibles en Internet y en documentos institucionales (ej. INTA).
- **Diagramas conceptuales**: se emplearon herramientas digitales como **ERD Plus** para el diseño del Modelo Entidad-Relación.

- **SQL Server**: se utilizó este sistema gestor de bases de datos para implementar el modelo físico, crear tablas y ejecutar consultas de prueba.
- **Análisis de casos reales**: nos basamos en sensores reales (ej. Gardena 1188) para orientar la estructura de la base de datos a un contexto práctico.

Capítulo IV

Este capítulo presenta los hallazgos del trabajo práctico, organizados a partir del análisis de la problemática planteada y de los objetivos definidos. La información se expondrá de manera objetiva, mostrando los pasos realizados y los resultados obtenidos en el diseño e implementación de un sistema de base de datos orientado al **control de riegos inteligentes**.

Desarrollo del Tema

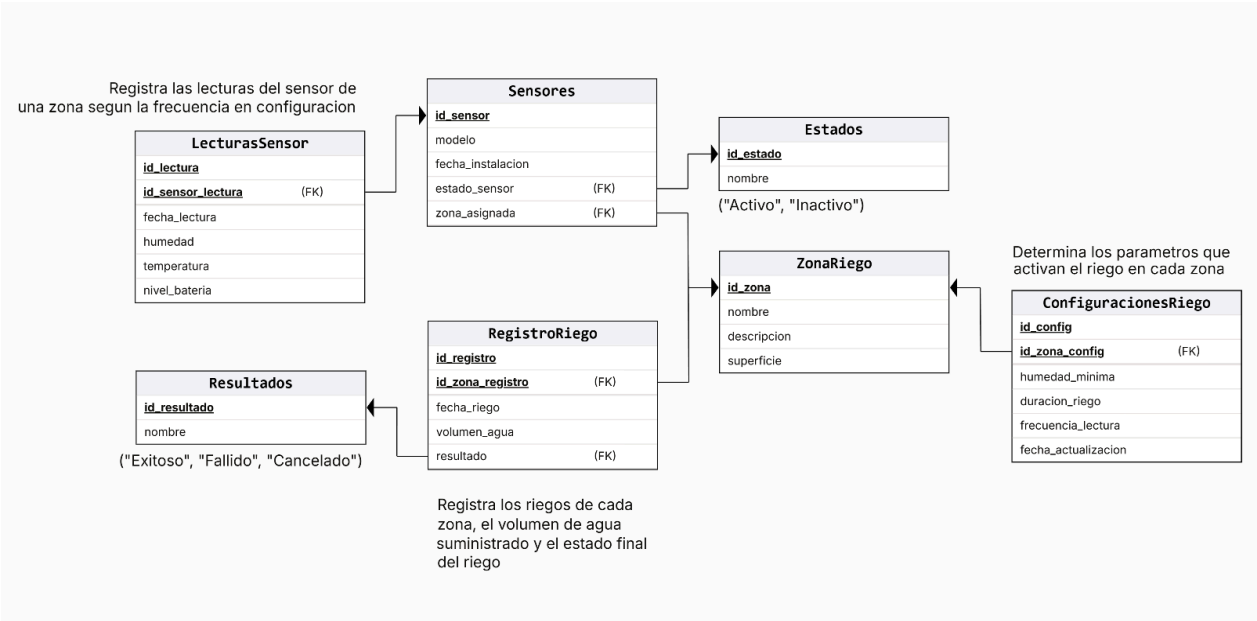
1. Identificación de las entidades principales

A partir del análisis del problema y de la revisión conceptual, se determinaron las entidades más relevantes que forman parte del sistema de riegos:

- **ZonaRiego**: sector de tierra que agrupa los sensores y define áreas independientes del sistema de riego.
- **Sensores**: representa cada dispositivo físico instalado en una zona.
- **LecturasSensor**: se encarga de registrar cada medición de los sensores.
- **ConfiguracionesRiego**: define los parámetros que determinan la activación (o no) del riego automático.
- **RegistroRiego**: se encarga de registrar cada riego automático.

2. Modelo Entidad–Relación (MER)

Se construyó un **MER** que refleja las entidades identificadas y sus relaciones:



Este modelo asegura la normalización hasta la **3FN**, evitando redundancias y garantizando integridad de datos.

Tema 1: Procedimientos y Funciones Almacenadas

Este tema posee dos objetivos principales:

- Comprender la diferencia entre procedimientos y funciones almacenadas.
- Aplicar procedimientos y funciones en la implementación de operaciones CRUD.

Partamos por la base y definamos primero qué es un procedimiento almacenado y qué es una función almacenada.

Procedimiento Almacenado

Un ***procedimiento almacenado*** es un conjunto de sentencias SQL que se pueden guardar en una base de datos, pueden ejecutarse bajo demanda para **realizar tareas de manipulación y validación de datos**, reduciendo la necesidad de escribir código SQL repetitivo para operaciones comunes. Son útiles en la gestión de bases de datos porque promueven la reutilización. Además, permiten mejorar la seguridad y la mantenibilidad de las bases de datos. Entre las **ventajas** de los procedimientos almacenados de SQL se incluyen las siguientes:

- **Reutilización del código**: Una vez creado un procedimiento almacenado, se puede llamar tantas veces como sea necesario, eliminando la redundancia en el código SQL.
- **Rendimiento mejorado**: Los procedimientos almacenados suelen ejecutarse más rápido porque están pre compilados y almacenados en el servidor de la base de datos, lo que reduce la latencia de la red y el tiempo de compilación.
- **Seguridad**: Los procedimientos almacenados pueden mejorar la seguridad de los datos y el control del acceso a datos sensibles, concediendo a los usuarios permiso para ejecutar un procedimiento almacenado sin acceso directo a las tablas.

La **sintaxis** para crear un procedimiento almacenado puede variar ligeramente en función del sistema de base de datos. A continuación se muestra un ejemplo general utilizando la sintaxis de SQL Server:

```
CREATE PROCEDURE NombreProcedimiento
    @Parametro1 INT,
    @Parametro2 VARCHAR(50)
AS
BEGIN
    -- Cuerpo de la consulta
     SELECT * FROM NombreTabla WHERE Columna1 = @Parametro1 AND
Columna2 = @Parametro2;
END;
```

- **CREATE PROCEDURE**: Este comando se utiliza para definir un nuevo procedimiento almacenado.
- **NombreProcedimiento**: El nombre dado al procedimiento almacenado. Debe ser único dentro de la base de datos.
- **@Parametro1, @Parametro2**: Los parámetros son opcionales, permiten que el procedimiento reciba datos de entrada. Cada parámetro se define con un símbolo @ y un tipo de datos (por ejemplo: `INT`, `VARCHAR(50)`).

- **AS BEGIN … END**: Las sentencias SQL dentro de `BEGIN` y `END` forman el cuerpo del procedimiento, donde se ejecuta la lógica principal.

Crear un procedimiento almacenado en SQL Server implica definir el nombre del procedimiento, los parámetros y las sentencias SQL que componen su cuerpo. En SQL Server, el comando `EXEC` o `EXECUTE` llama a un procedimiento almacenado. Por ejemplo:

```
EXEC NombreProcedimiento @Parametro1 = 102, @Parametro2 =´test´;
```

Función Almacenada

Una ***función almacenada***, también conocida como simplemente función, es un conjunto de instrucciones o bloques de código SQL que se almacenan en una base de datos y pueden ser invocados o llamados en diferentes momentos para realizar tareas específicas. Puede recibir cero o varios parámetros y **siempre retorna un valor como resultado**. Normalmente, se utiliza mediante una sentencia `SELECT` o dentro de una expresión.

Existen varios **tipos** de funciones, las **escalares** devuelven un único valor, como un número entero, una cadena de texto o un valor booleano. Las funciones **con valores de tabla** devuelven una tabla completa como resultado. Esto puede ser una tabla de varias instrucciones o una tabla en línea. Las **de agregado** toman un conjunto de valores de una columna y devuelven un único valor de resumen, como el `SUM()` o `AVG()`.

Las funciones dentro de SQL Server tienen las siguientes **características**:

- **Determinismo**: Las funciones pueden ser deterministas (siempre devuelven el mismo resultado para los mismos parámetros de entrada) o no deterministas (pueden devolver resultados diferentes).
- **Uso**: Se pueden usar en sentencias `SELECT`, en cláusulas `WHERE` y en otras partes de una consulta.
- **Reutilización**: Se crean para ejecutar un conjunto de instrucciones SQL repetidamente, lo que ayuda a mantener el código más limpio y fácil de mantener.

La **sintaxis** para crear una función puede variar ligeramente según el sistema de base de datos. A continuación se muestra un ejemplo general utilizando la sintaxis de SQL Server:

```
CREATE FUNCTION NombreFuncion (
    @Parametro1 INT,
    @Parametro2 INT)
RETURNS INT
DETERMINISTIC
BEGIN
    -- Cuerpo de la consulta
    RETURN <valor que corresponda>;
END;
```

- **CREATE FUNCTION**: Este comando se utiliza para definir una nueva función.
- **NombreFuncion**: El nombre dado a la función. Debe ser único dentro de la base de datos.
- **@Parametro1, @Parametro2**: Los parámetros permiten que la función reciba datos de entrada. Cada parámetro se define con un símbolo @ y un tipo de datos (por ejemplo: `INT`, `VARCHAR(50)`).
- **DETERMINISTIC**: Es una palabra clave que especifica que la función que definimos es determinística (si retorna el mismo resultado si se la invoca de nuevo con los mismos valores de entrada).
- **BEGIN … END**: Las sentencias SQL dentro de `BEGIN` y `END` forman el cuerpo de la función, donde se ejecuta la lógica principal.
- **RETURN**: El return debe aparecer como mínimo una vez dentro del cuerpo de la función, ya que todas las funciones devuelven al menos un valor.

Crear una función en SQL Server implica definir el nombre de la función, los parámetros de entrada, el tipo de valor que devuelve la función, el comportamiento de la función y su cuerpo.

En SQL Server, se llama a una función mediante la sentencia `SELECT`. Por ejemplo:

```
SELECT NombreFuncion(2, 4);
```

Ahora resaltando los aspectos más importantes de cada uno:

Aspecto	Procedimiento Almacenado	Función Almacenada
Propósito	Ejecutar operaciones o procesos (puede modificar datos).	Calcular y devolver un valor o conjunto de valores.
Tipo de Retorno	Ninguno o varios (usando parámetros de salida).	Obligatoriamente devuelve un valor.
Uso de sentencias SQL	No puede usarse directamente en un `SELECT`.	Puede usarse dentro de un `SELECT`, `WHERE`, etc.
Permite transacciones (`INSERT`, `UPDATE`, `DELETE`)	Sí	No
Ejemplo de uso	`EXEC InsertarSensor @modelo = 'Gardena 1188' ...;`	`SELECT CalcularPromedioHumedad(3);`

Habiendo sentado ya las bases de este tema pasemos a la práctica, recordemos que las tareas de este tema son:

- Realizar al menos tres procedimientos almacenados que permitan insertar, modificar y borrar registros de alguna de las tablas del proyecto.

- Insertar un lote de datos en las tablas mencionadas (guardar el script) con sentencias `INSERT` y otro lote invocando a los procedimientos creados.
- Realizar `UPDATE` y `DELETE` sobre algunos de los registros insertados en esas tablas invocando a los procedimientos.
- Desarrollar al menos tres funciones almacenadas.
- Comparar la eficiencia de las operaciones directas versus el uso de procedimientos y funciones.

Tarea N°1

Realizar al menos tres procedimientos almacenados que permitan insertar, modificar y borrar registros de alguna de las tablas del proyecto.

Para esta tarea, vamos a desarrollar tres procedimientos de operaciones CRUD para la tabla “ZonaRiego”:

Primer procedimiento - Insertar Registro:

```
CREATE PROCEDURE sp_insertar_zona_riego
    @nombre VARCHAR(15),
    @descripcion VARCHAR(30),
    @superficie FLOAT
AS
BEGIN
    INSERT INTO ZonaRiego (nombre, descripcion, superficie)
    VALUES (@nombre, @descripcion, @superficie);
END;
GO
```

Este procedimiento está definido para tomar tres parámetros de entrada, correspondientes al nombre, la descripción y superficie de la zona de riego, y dentro del cuerpo se encuentra la secuencia de inserción de los valores dentro de la tabla “ZonaRiego”.

Segundo procedimiento - Modificar Registro:

```
CREATE PROCEDURE sp_modificar_zona_riego
    @id_zona INT,
    @nombre VARCHAR(15),
    @descripcion VARCHAR(30),
    @superficie FLOAT
AS
BEGIN
    UPDATE ZonaRiego
    SET nombre = @nombre,
        descripcion = @descripcion,
        superficie = @superficie
    WHERE id_zona = @id_zona;
END;
GO
```

Este procedimiento, similar al anterior, está definido para tomar cuatro parámetros de entrada, correspondientes al identificador de la zona, su nombre, la descripción y superficie, y dentro del cuerpo se encuentra la secuencia de actualización de los valores de la tupla correspondiente dentro de la tabla “ZonaRiego”.

Tercer Procedimiento - Eliminar Registro:

```
CREATE PROCEDURE sp_eliminar_zona_riego
    @id_zona INT
AS
BEGIN
    DELETE FROM ZonaRiego
    WHERE id_zona = @id_zona;
END;
GO
```

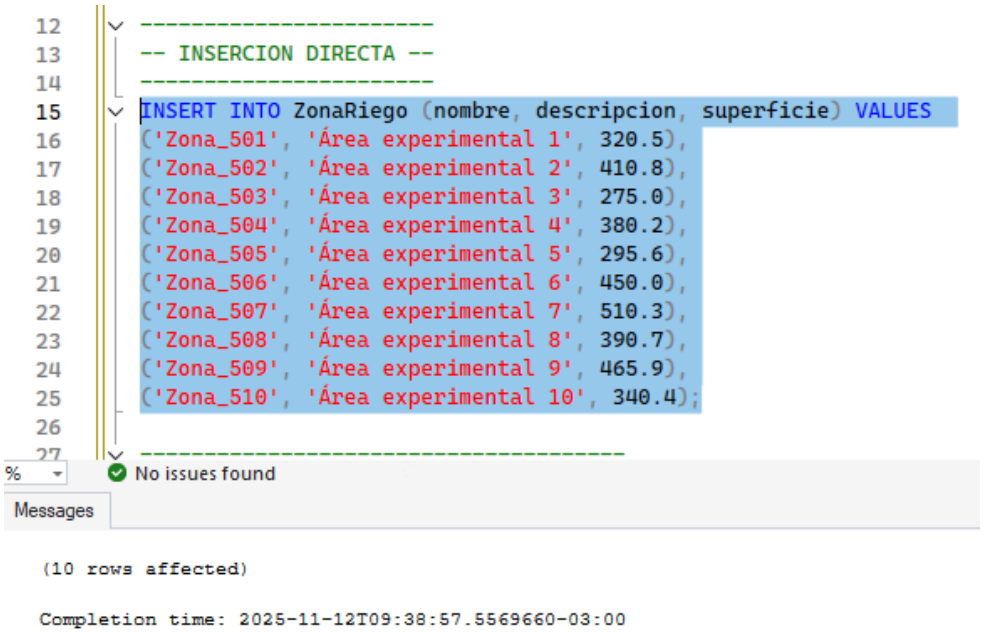
Este último procedimiento está definido para tomar solo un parámetro de entrada, correspondientes al identificador de la zona que se desea eliminar, y dentro del cuerpo se encuentra la secuencia de eliminación de la tupla correspondiente dentro de la tabla “ZonaRiego”.

Tarea N°2

Insertar un lote de datos en las tablas mencionadas (guardar el script) con sentencias `INSERT` y otro lote invocando a los procedimientos creados.

Inserción directa:

```
INSERT INTO ZonaRiego (nombre, descripcion, superficie) VALUES
('Zona_501', 'Área experimental 1', 320.5),
('Zona_502', 'Área experimental 2', 410.8),
('Zona_503', 'Área experimental 3', 275.0),
('Zona_504', 'Área experimental 4', 380.2),
('Zona_505', 'Área experimental 5', 295.6),
('Zona_506', 'Área experimental 6', 450.0),
('Zona_507', 'Área experimental 7', 510.3),
('Zona_508', 'Área experimental 8', 390.7),
('Zona_509', 'Área experimental 9', 465.9),
('Zona_510', 'Área experimental 10', 340.4);
```



Inserción mediante procedimientos:

```
EXEC sp_insertar_zona_riego 'Zona_511', 'Zona agregada mediante SP 1', 410.5;
EXEC sp_insertar_zona_riego 'Zona_512', 'Zona agregada mediante SP 2', 355.0;
EXEC sp_insertar_zona_riego 'Zona_513', 'Zona agregada mediante SP 3', 430.3;
EXEC sp_insertar_zona_riego 'Zona_514', 'Zona agregada mediante SP 4', 298.7;
EXEC sp_insertar_zona_riego 'Zona_515', 'Zona agregada mediante SP 5', 500.0;
EXEC sp_insertar_zona_riego 'Zona_516', 'Zona agregada mediante SP 6', 475.2;
EXEC sp_insertar_zona_riego 'Zona_517', 'Zona agregada mediante SP 7', 382.9;
EXEC sp_insertar_zona_riego 'Zona_518', 'Zona agregada mediante SP 8', 445.4;
```

```
EXEC sp_insertar_zona_riego 'Zona_519', 'Zona agregada mediante SP 9', 410.8;
EXEC sp_insertar_zona_riego 'Zona_520', 'Zona agregada mediante SP 10', 390.1;
```

```
27  ------------------------------------
28  -- INSERCION MEDIANTE PROCEDIMIENTO --
29  ------------------------------------
30  EXEC sp_insertar_zona_riego 'Zona_511', 'Zona agregada mediante SP 1', 410.5;
31  EXEC sp_insertar_zona_riego 'Zona_512', 'Zona agregada mediante SP 2', 355.0;
32  EXEC sp_insertar_zona_riego 'Zona_513', 'Zona agregada mediante SP 3', 430.3;
33  EXEC sp_insertar_zona_riego 'Zona_514', 'Zona agregada mediante SP 4', 298.7;
34  EXEC sp_insertar_zona_riego 'Zona_515', 'Zona agregada mediante SP 5', 500.0;
35  EXEC sp_insertar_zona_riego 'Zona_516', 'Zona agregada mediante SP 6', 475.2;
36  EXEC sp_insertar_zona_riego 'Zona_517', 'Zona agregada mediante SP 7', 382.9;
37  EXEC sp_insertar_zona_riego 'Zona_518', 'Zona agregada mediante SP 8', 445.4;
38  EXEC sp_insertar_zona_riego 'Zona_519', 'Zona agregada mediante SP 9', 410.8;
39  EXEC sp_insertar_zona_riego 'Zona_520', 'Zona agregada mediante SP 10', 390.1;
```

```
00 %   No issues found
Messages

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

Completion time: 2025-11-12T09:41:46.4779317-03:00
```

Ahora vamos a hacer un `SELECT` a la tabla de “ZonaRiego” para ver si efectivamente se insertaron los registros.

```
47    SELECT * FROM ZonaRiego;
```

100 % No issues found

Results | Messages | Client Statistics

	id_zona	nombre	descripcion	superficie
134	134	Zona_134	Área de cultivo número 134	140,68
135	135	Zona_135	Área de cultivo número 135	1089,97
136	136	Zona_501	Área experimental 1	320,5
137	137	Zona_502	Área experimental 2	410,8
138	138	Zona_503	Área experimental 3	275
139	139	Zona_504	Área experimental 4	380,2
140	140	Zona_505	Área experimental 5	295,6
141	141	Zona_506	Área experimental 6	450
142	142	Zona_507	Área experimental 7	510,3
143	143	Zona_508	Área experimental 8	390,7
144	144	Zona_509	Área experimental 9	465,9
145	145	Zona_510	Área experimental 10	340,4
146	146	Zona_511	Zona agregada mediante SP 1	410,5
147	147	Zona_512	Zona agregada mediante SP 2	355
148	148	Zona_513	Zona agregada mediante SP 3	430,3
149	149	Zona_514	Zona agregada mediante SP 4	298,7
150	150	Zona_515	Zona agregada mediante SP 5	500
151	151	Zona_516	Zona agregada mediante SP 6	475,2
152	152	Zona_517	Zona agregada mediante SP 7	382,9
153	153	Zona_518	Zona agregada mediante SP 8	445,4
154	154	Zona_519	Zona agregada mediante SP 9	410,8
155	155	Zona_520	Zona agregada mediante SP 10	390,1

Tarea N°3

Realizar `UPDATE` y `DELETE` sobre algunos de los registros insertados en esas tablas invocando a los procedimientos.

Actualización:

```
EXEC sp_modificar_zona_riego @id_zona = 137, @nombre = 'Zona_Modificada', @descripcion = 'Descripción actualizada', @superficie = 500.0;
```

```
-----------------
-- ACTUALIZACION --
-----------------
SELECT * FROM ZonaRiego
WHERE id_zona = 137;
GO

EXEC sp_modificar_zona_riego @id_zona = 137, @nombre = 'Zona_Modificada', @descripcion = 'Descripción actualizada', @superficie = 500.0;
GO

SELECT * FROM ZonaRiego
WHERE id_zona = 137;
GO
```

100 % — No issues found

Results | Messages

	id_zona	nombre	descripcion	superficie
1	137	Zona_502	Área experimental 2	410,8

	id_zona	nombre	descripcion	superficie
1	137	Zona_Modificada	Descripción actualizada	500

Eliminación:

```
EXEC sp_eliminar_zona_riego @id_zona = 150;
```

```
-----------------
-- ELIMINACION --
-----------------
SELECT * FROM ZonaRiego
WHERE id_zona = 150;
GO

EXEC sp_eliminar_zona_riego @id_zona = 150;
GO

SELECT * FROM ZonaRiego
WHERE id_zona > 145;
GO
```

100 % — No issues found

Results | Messages

	id_zona	nombre	descripcion	superficie
1	150	Zona_515	Zona agregada mediante SP 5	500

	id_zona	nombre	descripcion	superficie
1	146	Zona_511	Zona agregada mediante SP 1	410,5
2	147	Zona_512	Zona agregada mediante SP 2	355
3	148	Zona_513	Zona agregada mediante SP 3	430,3
4	149	Zona_514	Zona agregada mediante SP 4	298,7
5	151	Zona_516	Zona agregada mediante SP 6	475,2
6	152	Zona_517	Zona agregada mediante SP 7	382,9
7	153	Zona_518	Zona agregada mediante SP 8	445,4
8	154	Zona_519	Zona agregada mediante SP 9	410,8
9	155	Zona_520	Zona agregada mediante SP 10	390,1

Tarea N°4

Desarrollar al menos tres funciones almacenadas.

Función N°1 - Cálculo del promedio de humedad de una zona:

```
CREATE FUNCTION fn_promedio_humedad_zona (@id_zona INT)
RETURNS FLOAT
AS
BEGIN
    DECLARE @promedio FLOAT;
    SELECT @promedio = AVG(humedad)
    FROM LecturasSensor L
    JOIN Sensores S ON L.id_sensor_lectura = S.id_sensor
    WHERE S.zona_asignada = @id_zona;
    RETURN @promedio;
END;
GO
```

Función N°2 - Obtención de la cantidad de sensores activos:

```
CREATE FUNCTION fn_cant_sensores_activos ()
RETURNS INT
AS
BEGIN
    DECLARE @cantidad INT;
          SELECT  @cantidad  =  COUNT(*)  FROM  Sensores  WHERE
estado_sensor = 1;
    RETURN @cantidad;
END;
GO
```

Función N°3 - Cálculo de la eficiencia de riego de una zona:

```
CREATE FUNCTION fn_eficiencia_riego_zona (@id_zona INT)
RETURNS FLOAT
AS
BEGIN
    DECLARE @total INT, @exitosos INT;
        SELECT @total = COUNT(*) FROM RegistroRiego WHERE id_zona_registro = @id_zona;
        SELECT @exitosos = COUNT(*) FROM RegistroRiego WHERE id_zona_registro = @id_zona AND resultado = 1;
     RETURN CASE WHEN @total = 0 THEN 0 ELSE CAST(@exitosos AS FLOAT)/@total * 100 END;
END;
GO
```

Tarea N°5

Comparar la eficiencia de las operaciones directas versus el uso de procedimientos y funciones.

Para esta tarea, usaremos los comandos `SET STATISTICS TIME ON` y `SET STATISTICS IO ON`, que nos permite ver el tiempo necesario para analizar, compilar y ejecutar cada declaración de la consulta (`STATISTICS TIME`) y ver tiempo de actividad en disco generado por la consulta (`STATISTICS IO`).

```
SET STATISTICS TIME ON;         -- Este comando muestra el tiempo necesario para analizar, compilar y ejecutar cada declaracion de la consulta
SET STATISTICS IO ON;           -- Este comando muestra el tiempo de actividad en disco generado por la consulta
GO

-- Para testear vamos a usar los comandos de la Tarea 2
-------------------------
-- INSERCION DIRECTA --
-------------------------
INSERT INTO ZonaRiego (nombre, descripcion, superficie) VALUES
('Zona_501', 'Área experimental 1', 320.5),
('Zona_502', 'Área experimental 2', 410.8),
('Zona_503', 'Área experimental 3', 275.0),
('Zona_504', 'Área experimental 4', 380.2),
('Zona_505', 'Área experimental 5', 295.6),
('Zona_506', 'Área experimental 6', 450.0),
('Zona_507', 'Área experimental 7', 510.3),
('Zona_508', 'Área experimental 8', 390.7),
('Zona_509', 'Área experimental 9', 465.9),
('Zona_510', 'Área experimental 10', 340.4);
GO

-------------------------------------
-- INSERCION MEDIANTE PROCEDIMIENTO --
-------------------------------------
EXEC dbo.sp_insertar_zona_riego 'Zona_511', 'Zona agregada mediante SP 1', 410.5;
EXEC dbo.sp_insertar_zona_riego 'Zona_512', 'Zona agregada mediante SP 2', 355.0;
EXEC dbo.sp_insertar_zona_riego 'Zona_513', 'Zona agregada mediante SP 3', 430.3;
EXEC dbo.sp_insertar_zona_riego 'Zona_514', 'Zona agregada mediante SP 4', 298.7;
EXEC dbo.sp_insertar_zona_riego 'Zona_515', 'Zona agregada mediante SP 5', 500.0;
EXEC dbo.sp_insertar_zona_riego 'Zona_516', 'Zona agregada mediante SP 6', 475.2;
EXEC dbo.sp_insertar_zona_riego 'Zona_517', 'Zona agregada mediante SP 7', 382.9;
EXEC dbo.sp_insertar_zona_riego 'Zona_518', 'Zona agregada mediante SP 8', 445.4;
EXEC dbo.sp_insertar_zona_riego 'Zona_519', 'Zona agregada mediante SP 9', 410.8;
EXEC dbo.sp_insertar_zona_riego 'Zona_520', 'Zona agregada mediante SP 10', 390.1;
GO
```

```
(10 rows affected)
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'ZonaRiego'. Scan count 0, logical reads 2, physical reads (

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(1 row affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 1 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'ZonaRiego'. Scan count 0, logical reads 2, physical reads (

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(1 row affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'ZonaRiego'. Scan count 0, logical reads 2, physical reads (

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(1 row affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'ZonaRiego'. Scan count 0, logical reads 2, physical reads (

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(1 row affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'ZonaRiego'. Scan count 0, logical reads 2, physical reads (
```

En el caso de la inserción directa vs. la inserción con procedimientos, vemos que la directa es más rápida, pero al ser directa no posee control de errores ni validaciones,

en cambio el procedimiento almacenado es más lento (en este caso), pero se le pueden agregar verificaciones para mejorar la seguridad, además de ser reutilizable.

```sql
-- Y ahora para las funciones, usando la de humedad promedio
SELECT dbo.fn_promedio_humedad_zona(1) AS Promedio_Humedad;

SELECT
    AVG(humedad) AS Promedio_Humedad
FROM LecturasSensor AS L
INNER JOIN Sensores AS S ON L.id_sensor_lectura = S.id_sensor
WHERE S.zona_asignada = 1;
GO
```

```
(1 row affected)
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead reads (
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead reads
Table 'LecturasSensor'. Scan count 1, logical reads 27, physical reads 0, page server reads 0, read-ahead
Table 'Sensores'. Scan count 1, logical reads 2, physical reads 0, page server reads 0, read-ahead reads (

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 2 ms.

(1 row affected)
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead reads (
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead reads
Table 'LecturasSensor'. Scan count 1, logical reads 27, physical reads 0, page server reads 0, read-ahead
Table 'Sensores'. Scan count 1, logical reads 2, physical reads 0, page server reads 0, read-ahead reads (

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 1 ms.
```

Y en el caso de la consulta directa vs. la función almacenada, vemos que la consulta directa es más rápida a comparación de la función, pero la ventaja de la función radica en poder usarla de manera repetitiva y sencilla, además de poder llamarla dentro de otras consultas, mejorando la legibilidad.

Tema 2: Optimización de Consultas a través de Índices

Tipos de Índices en SQL Server y sus Aplicaciones

Los índices son estructuras adicionales que aceleran la búsqueda de datos dentro de las tablas, de forma similar al índice de un libro. Su uso adecuado mejora drásticamente el rendimiento de consultas, especialmente en bases grandes como tu sistema de riego. A continuación, se explican todos los tipos de índices relevantes, con ejemplos y su aplicación práctica.

Clustered Index

El **índice clustered**, o índice agrupado, organiza físicamente los datos en disco siguiendo el orden de la columna indexada. Dado que la tabla se ordena a partir de dicho índice, sólo puede existir uno por tabla. Es especialmente útil en columnas que se utilizan para ordenar, filtrar por rangos o representar el orden natural de los datos, como fechas o códigos correlativos. Su uso resulta fundamental en tablas de gran tamaño, donde minimizar lecturas lógicas se vuelve crítico.

Nonclustered Index

El **índice nonclustered**, en cambio, no altera el orden físico de los datos. Opera como una estructura adicional que contiene claves y punteros hacia las filas reales de la tabla. Este tipo de índice es adecuado para consultas por igualdad o filtrados frecuentes sobre columnas específicas, así como para acelerar operaciones JOIN. Dado su carácter secundario, una tabla puede contener múltiples índices de este tipo.

Dentro de esta categoría también existen los índices **nonclustered con columnas incluidas**, que agregan a la estructura indexada columnas adicionales para evitar accesos a la tabla base durante un SELECT. Esta estrategia elimina la necesidad de realizar key lookups y es muy efectiva para consultas de lectura intensiva, tales como reportes o consultas analíticas que devuelven varias columnas.

Unique Index

Los **índices únicos** ofrecen además una garantía de integridad, evitando la existencia de valores duplicados en columnas críticas como correos electrónicos, DNIs o códigos de identificación. Proveen mejoras en el rendimiento similares a los índices tradicionales, sumadas al beneficio de validar la unicidad de los datos.

Índices Compuestos

En cuanto a los **índices compuestos**, estos permiten indexar dos o más columnas simultáneamente. Son especialmente convenientes en consultas que aplican filtros combinados o en operaciones JOIN que vinculan múltiples atributos entre tablas.

Filtered Indexes

Por otra parte, los **índices filtrados** constituyen una alternativa focalizada, ya que almacenan solo las filas que cumplen una condición determinada. Esto los vuelve ideales para tablas extensas donde solo un subconjunto reducido de datos es consultado con frecuencia, por ejemplo, sensores activos.

Columnstore Index

Finalmente, el **columnstore index** presenta una arquitectura orientada a columnas y está diseñado para escenarios de análisis masivo de datos. Su elevada capacidad de compresión y su eficiencia en consultas agregadas lo posicionan como la opción preferida en aplicaciones de inteligencia de negocios o procesamiento analítico de grandes volúmenes.

Carga Masiva de Datos

Para simular un entorno de producción con grandes volúmenes de datos, se generó un millón de registros utilizando un script automatizado:

```
USE sistema_riego;
GO

DECLARE @minSensor INT, @maxSensor INT;
SELECT @minSensor = MIN(id_sensor), @maxSensor = MAX(id_sensor)
FROM
Sensores;
```

```
;WITH Numbers AS (
     SELECT TOP (1000000) ROW_NUMBER() OVER (ORDER BY (SELECT
NULL)) AS n
     FROM sys.objects AS a
     CROSS JOIN sys.objects AS b
     CROSS JOIN sys.objects AS c
)
INSERT INTO LecturasSensor (fecha_lectura, humedad, temperatura,
nivel_bateria, id_sensor_lectura)
SELECT
     DATEADD(DAY, -ABS(CHECKSUM(NEWID())) % 730, GETDATE()),
     CAST(ROUND(RAND(CHECKSUM(NEWID())) * 100, 2) AS FLOAT),
     CAST(ROUND(RAND(CHECKSUM(NEWID())) * 35 + 5, 2) AS FLOAT),
     ABS(CHECKSUM(NEWID())) % 100,
     FLOOR(RAND(CHECKSUM(NEWID())) * (@maxSensor - @minSensor +
1)) + @minSensor
FROM Numbers;
```

Pruebas, consultas y planes de ejecución

Se evaluaron tres escenarios: sin índices, con un índice clustered en fecha_lectura y con un índice nonclustered que incluye columnas adicionales. El objetivo fue comparar tiempos, lecturas lógicas y tipo de operación utilizado por SQL Server.

Escenario 1: Consulta sin índices

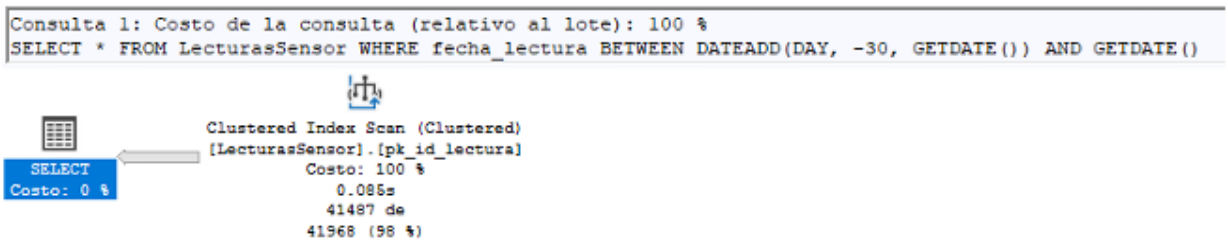
En este escenario la consulta realiza un Table Scan, recorriendo toda la tabla debido a que no existe ningún índice que permita filtrar por fecha.

```
SET STATISTICS TIME ON;
SET STATISTICS IO ON;
```

```
SELECT *
FROM LecturasSensor
WHERE fecha_lectura BETWEEN DATEADD(DAY, -30, GETDATE()) AND
GETDATE();
```

Resultados:

- Tiempo de CPU: 78 ms
- Tiempo total: 271 ms
- Lecturas lógicas: 5001
- Tipo de operación: Table Scan



Escenario 2: Índice clustered en fecha_lectura

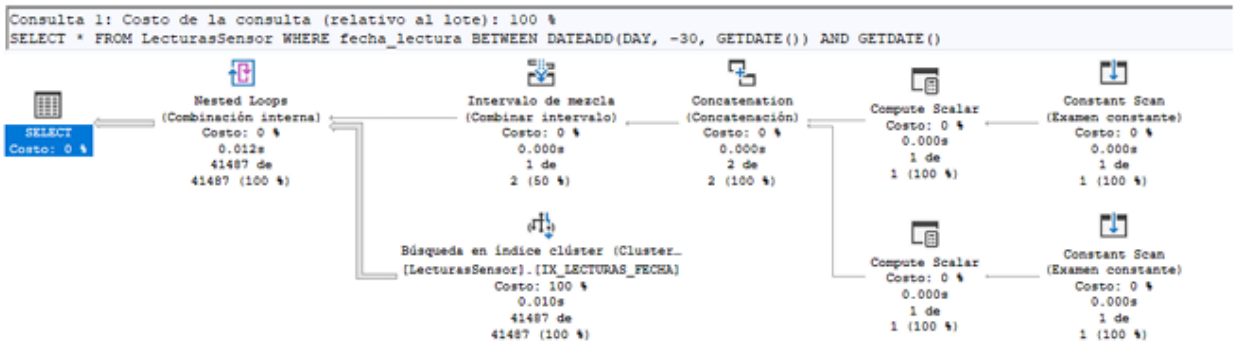
El índice clustered ordena físicamente la tabla usando la columna fecha_lectura. Para crearlo fue necesario convertir la clave primaria en NONCLUSTERED.

```
ALTER TABLE LecturasSensor
DROP CONSTRAINT pk_id_lectura;
ALTER TABLE LecturasSensor
ADD CONSTRAINT pk_id_lectura
PRIMARY KEY NONCLUSTERED (id_lectura, id_sensor_lectura);
CREATE CLUSTERED INDEX IX_LECTURAS_FECHA
ON LecturasSensor (fecha_lectura);
```

Resultados:

- Tiempo de CPU: 16 ms
- Tiempo total: 288 ms

- Lecturas lógicas: 251
- Tipo de operación: Clustered Index Seek / Range Scan



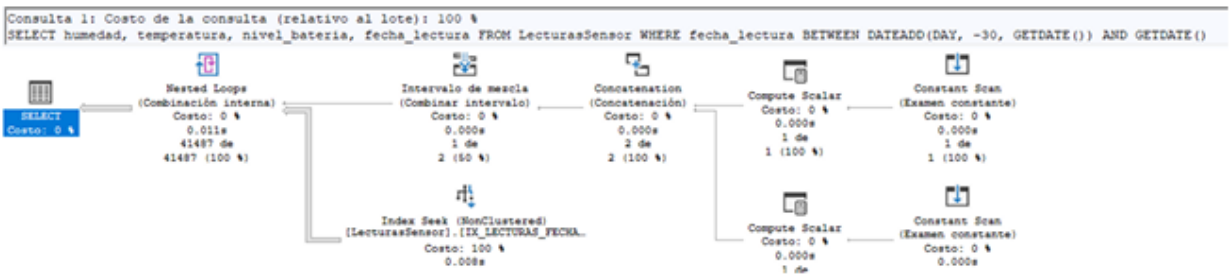
Escenario 2: Índice nonclustered con columnas incluidas

Este índice optimiza aún más la consulta porque permite que SQL Server resuelva todo el SELECT directamente desde el índice, sin acceder a la tabla base, eliminando los Key Lookups.

```
CREATE NONCLUSTERED INDEX IX_LECTURAS_FECHA_INCL
ON LecturasSensor (fecha_lectura)
INCLUDE (humedad, temperatura, nivel_bateria);
```

Resultados:

- Tiempo de CPU: 0 ms
- Tiempo total: 189 ms
- Lecturas lógicas: 196
- Tipo de operación: Nonclustered Index Seek (Range Scan).



Tema 3: 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 explícitas	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 instrucción **BEGIN TRAN** 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.

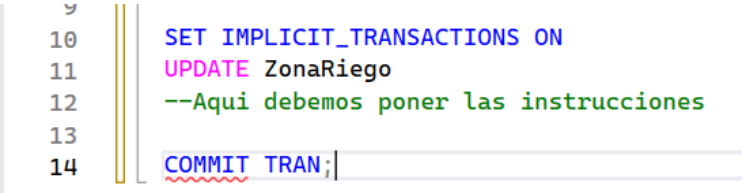
Cómo definir una transacción implícita

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**.



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.

Cómo 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, completamos 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;
```

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 comandos BEGIN TRAN 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 variable @@TRANCOUNT 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 revertirse, 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.

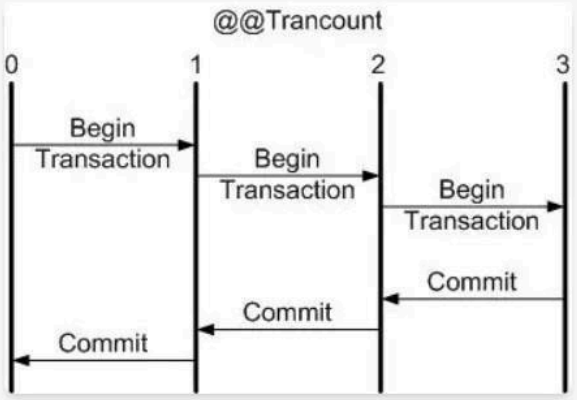


Figura 1: Un COMMIT siempre equilibra un BEGIN TRANSACTION reduciendo el recuento de transacciones en uno.

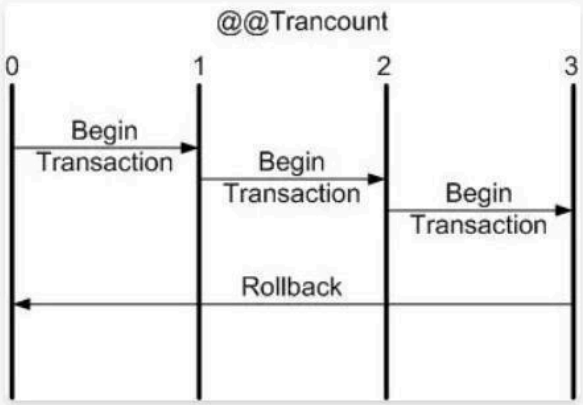


Figura 2: Un único ROLLBACK siempre revierte toda la transacción.

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 en 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 revirtieron 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 más registros en otra tabla. Actualizar los datos solamente si toda la operación es completada con éxito.

Instrucción:

```sql
BEGIN TRY;
    BEGIN TRAN;
        INSERT INTO RegistroRiego(volumen_agua,id_zona_registro,resultado)
        VALUES (288.70,1,1);

        INSERT INTO LecturasSensor(humedad,temperatura,nivel_bateria,id_sensor_lectura)
        VALUES(70.5,30.33,80,3);

        UPDATE ConfiguracionesRiego
            SET duracion_riego = 30
        WHERE id_config = 5;

    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

```
(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

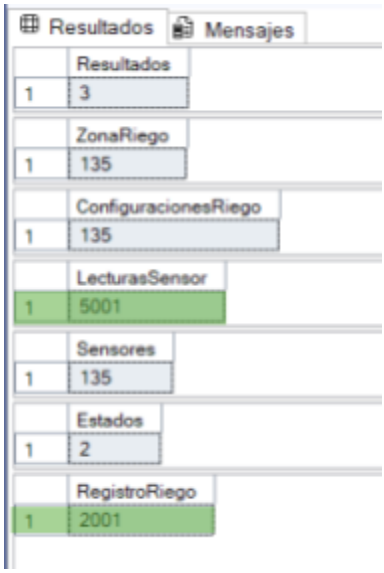
	LecturasSensor
1	5000

	Sensores
1	135

	Estados
1	2

	RegistroRiego
1	2000

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



Y la tabla que vamos a actualizar será **ConfiguracionesRiego** más precisamente 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:

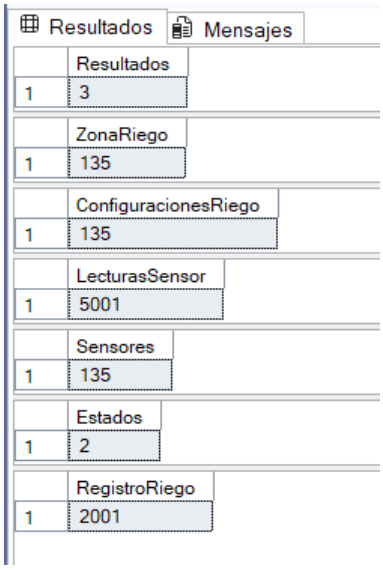
```
(1 fila afectada)

(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:

```sql
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 revirtieron los cambios.';
END CATCH;
GO
```

No se encontraron problemas.

Mensajes

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

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

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

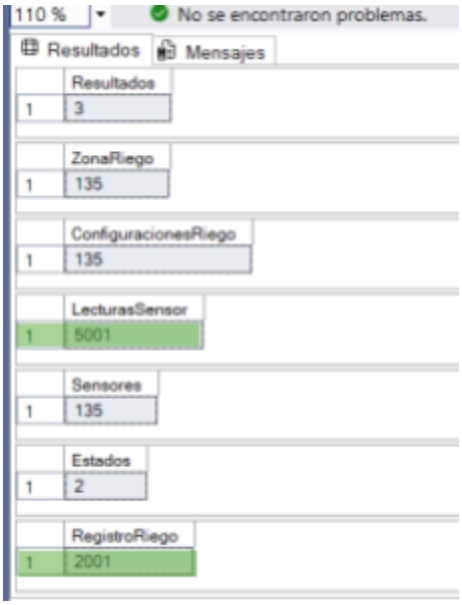
```sql
--TABLAS
SELECT * FROM Resultados;
SELECT * FROM ZonaRiego;
```

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:



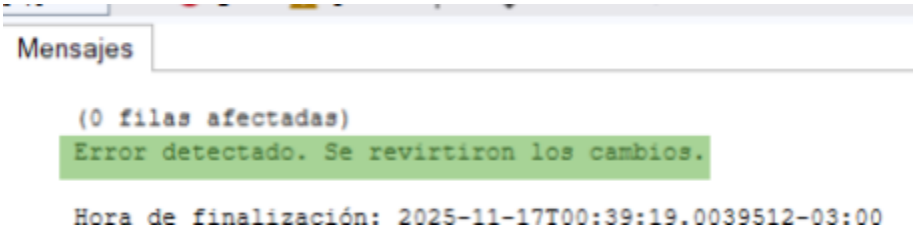
Y que no se realizó ninguna actualización en la tabla **ConfiguracionesRiego** en la columna de **duracion_riego** para el registro con id 4:

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
				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. Como las conclusiones no corresponden al desarrollo, las mismas estarán en su apartado correspondiente.

Tema 4: Backup y Restore. Backup en Línea

En esta parte abordaremos el proceso completo de backup y restore en SQL Server, utilizando el modelo de recuperación FULL, el cual permite realizar backups en línea, es decir, mientras la base se encuentra operativa.

El trabajo consiste en generar una secuencia de respaldos (FULL → LOG1 → LOG2), realizar inserciones en una tabla de referencia y recuperar la base en distintos puntos del tiempo para verificar la correcta integridad de los datos.

Este enfoque garantiza la continuidad operativa y demuestra cómo SQL Server permite restaurar estados anteriores sin pérdida de información, siempre que se respete el orden de la cadena de backup.

Recordemos los objetivos de este tema:

- Conocer los distintos tipos de backup y su uso dentro del modelo de recuperación FULL.
- Implementar una estrategia de respaldo basada en una cadena de restauración (FULL + LOGs).
- Verificar la consistencia de los datos luego de cada restauración.
- Comprender el funcionamiento del backup en línea y su importancia operativa.

Modelo de Recuperación FULL

SQL Server permite definir el **modelo de recuperación** de una base para determinar cómo se gestionan los cambios y qué tipo de recuperaciones son posibles. Los modelos más comunes son: SIMPLE, FULL y BULK_LOGGED.

El modelo FULL registra todas las operaciones en el log de transacciones, lo que permite:

- Hacer backup del log.
- Recuperar la base hasta un punto específico en el tiempo.
- Restaurar estados intermedios mediante secuencias FULL → LOG1 → LOG2 → LOGn.
- Realizar backups en línea sin interrumpir la operación de usuarios.

Este modelo se habilita con:

```
ALTER DATABASE sistema_riego
SET RECOVERY FULL;
```

En este trabajo se utiliza FULL porque es el único que permite restauraciones con múltiples logs, recreando estados intermedios sin pérdida de datos.

Log de transacciones

El Transaction Log o archivo de log funciona como un registro continuo de:

- inserciones,
- actualizaciones,
- eliminaciones,
- creación de objetos,
- transacciones confirmadas y no confirmadas.

Cada operación genera un número interno llamado LSN (Log Sequence Number). Este valor ordena cronológicamente todos los cambios de la base. El proceso de restauración exige que los backups se apliquen respetando el orden de LSN. De lo contrario puede surgir un error como nos ocurrió durante el intento de este proyecto que surgió el mensaje:

"The log in this backup set begins at LSN… which is too recent…"

que indica que se está intentando restaurar un log que no continúa la secuencia.

Backup en línea

El backup en línea es un respaldo que se ejecuta mientras la base está siendo utilizada por usuarios o aplicaciones, sin necesidad de detenerla ni ponerla en modo exclusivo.

- La base sigue operativa
- Los cambios continúan registrándose
- El log de transacciones asegura consistencia

SQL Server garantiza que el backup sea coherente porque:

1. Congela páginas de datos momentáneamente para capturarlas.
2. Utiliza el log para completar las transacciones en curso.

3. Crea un snapshot interno del estado del archivo en el instante del backup.

El backup en línea sólo es posible si el modelo de recuperación es FULL o BULK_LOGGED.

Comandos utilizados en un backup en línea

Backup Full (en línea):

```
BACKUP DATABASE sistema_riego
TO DISK = 'C:\Backups\sistema_riego_FULL.bak'
WITH INIT, NAME = 'Backup FULL';
```

- Realiza copia completa de los archivos .mdf y .ldf.
- No interrumpe el uso de la base.

Backup del Log (en línea)

```
BACKUP LOG sistema_riego
TO DISK = 'C:\Backups\sistema_riego_LOG1.trn'
WITH INIT, NAME = 'Backup LOG 1';
```

El backup de log registra únicamente:

- Cambios desde el anterior backup de log o full.
- Los LSN que continuarán la secuencia de restauración.

Restauración con NORECOVERY y RECOVERY

Cuando se restaura una base, SQL Server debe saber si:

- va a seguir recibiendo logs, o
- ya se terminó la secuencia.

NORECOVERY: Mantiene la base en estado RESTORING, permitiendo aplicar backups de log posteriores.

```
RESTORE DATABASE sistema_riego
FROM DISK='FULL.bak'
WITH NORECOVERY;
```

Mientras está en RESTORING:

- No se puede consultar la base
- No se puede modificar
- No se puede cambiar a MULTI_USER
- Solo se puede seguir aplicando backups

RECOVERY: Cierra la secuencia de restauración y deja la base ONLINE.

```
RESTORE LOG sistema_riego
FROM DISK='LOG1.trn'
WITH RECOVERY;
```

Una vez que se usa RECOVERY:

- La restauración se da por finalizada
- No se pueden aplicar más logs (se rompería la cadena)

Secuencia completa del proceso de Backup / Restore

Backup

1. FULL
2. Generación de nuevos datos
3. LOG1
4. Generación de más datos
5. LOG2

Restore

- Estado 1: Solo FULL
- Estado 2: FULL + LOG1

- Estado 3: FULL + LOG1 + LOG2

Errores típicos en la cadena de restauración

1) Usar RECOVERY en el momento incorrecto

   Cierra la secuencia e impide aplicar logs posteriores.

2) Restaurar un FULL que no corresponde al LOG

   Genera error de LSN no coincidente.

3) Intentar restaurar con la base ONLINE

   Da error: "database is in use".

4) Querer cambiar MULTI_USER mientras está en RESTORING

   Da error: “ALTER DATABASE is not permitted while… restoring”

¿Por qué es útil el backup en línea?

En entornos reales:

- No se puede detener la base para respaldarla.
- Los usuarios deben operar mientras se realiza el backup.
- Debe garantizarse la consistencia aunque existan transacciones abiertas.

El proceso FULL + LOG permite:

- Mantener un historial detallado de cambios.
- Restaurar estados intermedios según necesidad.
- Proteger la integridad de los datos ante fallas.

Creación de tablas y carga de datos de prueba

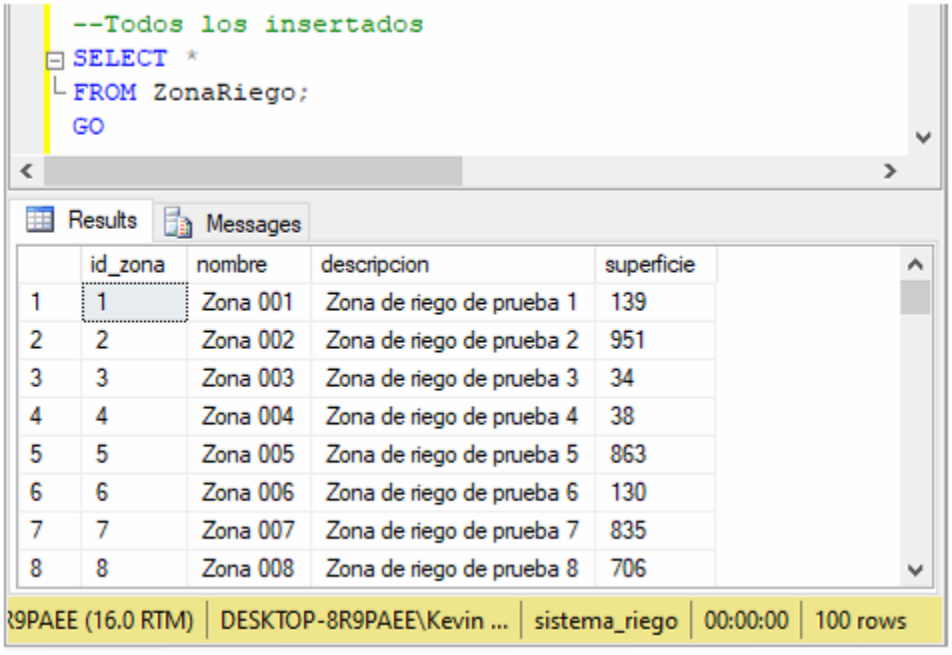
Para realizar las pruebas se creó la tabla **ZonaRiego** y un procedimiento almacenado **GenerarZonasRiegoPrueba**, que permite generar N registros automáticos:

```sql
CREATE OR ALTER PROCEDURE GenerarZonasRiegoPrueba
    @Cantidad INT = 100
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @i INT = 1;

    WHILE @i <= @Cantidad
    BEGIN
        INSERT INTO ZonaRiego (nombre, descripcion, superficie)
        VALUES (
            CONCAT('Zona ', RIGHT('000' + CAST(@i AS varchar(3)), 3)),
            CONCAT('Zona de riego de prueba ', @i),
            CAST(10 + (ABS(CHECKSUM(NEWID())) % 991) AS float)
        );

        SET @i = @i + 1;
    END;
END;
```

Mediante la iteración desde i = 1 hasta la cantidad necesaria inserta los registros con nombre y descripción iterativos y superficies generadas de manera aleatoria.

Con el siguiente código se generó 100 registros para la base inicial:

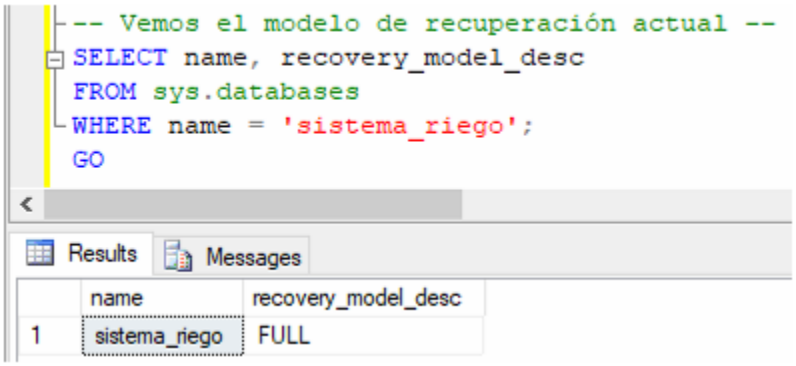
```sql
EXEC GenerarZonasRiegoPrueba @Cantidad = 100;
```



Se verificó los 100 registros ingresados con la zona, descripción y superficie.

Configuración del modelo de recuperación

Primero verificamos que el modelo de recuperación esté en FULL.

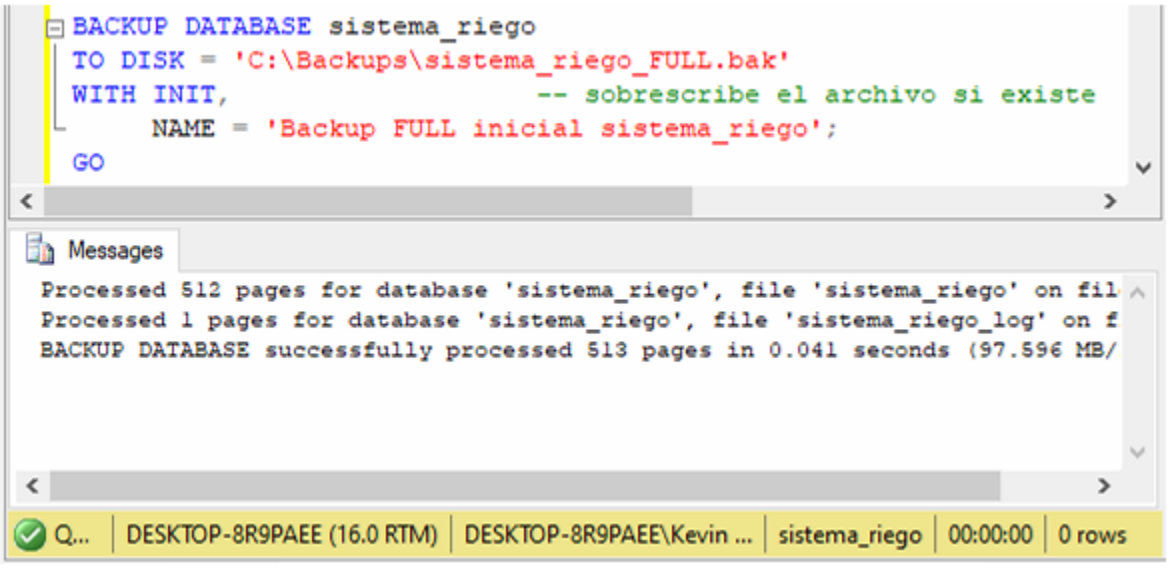


Sino lo cambiamos con:

```
ALTER DATABASE sistema_riego
SET RECOVERY FULL;
```

Realización del Backup FULL

Realizamos el backup full mediante el siguiente código:



Y vemos que se realiza de forma exitosa.

Primeros 10 inserts y verificación

Mediante el procedimiento GenerarZonasRiegoPrueba insertamos 10 registros:

```
EXEC GenerarZonasRiegoPrueba @Cantidad = 10;
GO
```

Y verificamos:

```
SELECT COUNT(*) AS CantZonaRiego
FROM ZonaRiego;
GO
```

Results | Messages

	CantZonaRiego
1	110

Q... | DESKTOP-8R9PAEE (16.0 RTM) | DESKTOP-8R9PAEE\Kevin ... | sistema_riego | 00:00:00 | 1 rows

Backup del archivo de LOG (LOG1)

Realizamos el backup LOG1 junto con la hora del backup para su registro:

```
BACKUP LOG sistema_riego
TO DISK = 'C:\Backups\sistema_riego_LOG1.trn'
WITH INIT,
     NAME = 'Backup LOG 1 - luego de primeros 10 inserts en ZonaRiego';
GO

SELECT SYSDATETIME() AS HoraBackupLog1; --Guardamos la hora del backup
GO
```

Results | Messages

	HoraBackupLog1
1	2025-11-17 11:54:27.7309108

Segundos 10 inserts y verificación

Mediante el procedimiento GenerarZonasRiegoPrueba insertamos otros 10 registros:

```
EXEC GenerarZonasRiegoPrueba @Cantidad = 10;
GO
```

Y verificamos:

```
SELECT COUNT(*) AS CantZonaRiego
FROM ZonaRiego;
GO
```

Results | Messages

	CantZonaRiego
1	120

Segundo backup de LOG (LOG2)

Realizamos el backup LOG2 junto con la hora del backup para su registro:

```
BACKUP LOG sistema_riego
TO DISK = 'C:\Backups\sistema_riego_LOG2.trn'
WITH INIT,
     NAME = 'Backup LOG 2 - luego de otros 10 inserts en ZonaRiego';
GO

SELECT SYSDATETIME() AS HoraBackupLog1; --Guardamos la hora del backup
GO
```

Results | Messages

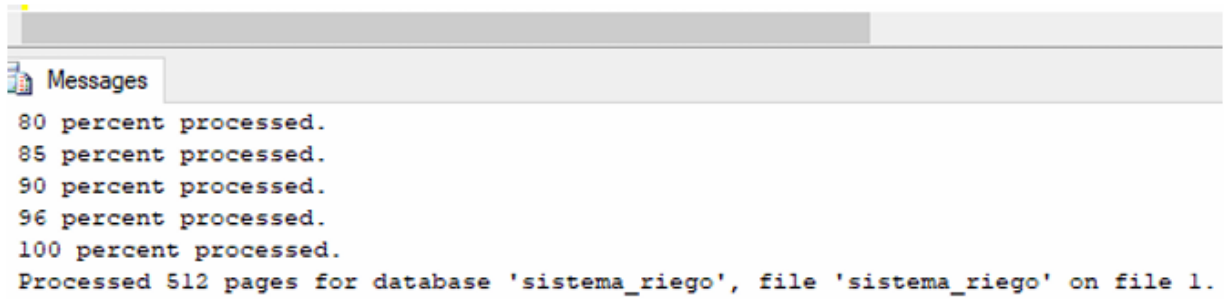
HoraBackupLog1
2025-11-17 11:57:47.6365938

Restauración del FULL

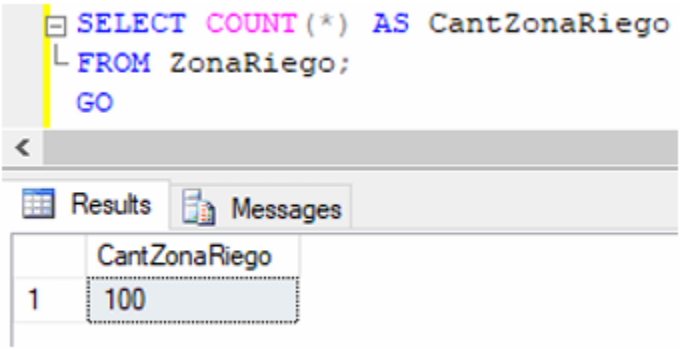
Cambiamos a Master (USE Master) y ejecutamos:

```
ALTER DATABASE sistema_riego
SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
GO
```

```
RESTORE DATABASE sistema_riego
FROM DISK = 'C:\Backups\sistema_riego_FULL.bak'
WITH RECOVERY,
REPLACE,
STATS = 5;
GO

ALTER DATABASE sistema_riego
SET MULTI_USER;
GO
```



Resultado esperado: 100 registros. Cambiamos a sistema_riego (USE) y verificamos:



Obteniendo un resultado exitoso.

Restauración FULL + LOG1

```
USE master;
```

```
GO

ALTER DATABASE sistema_riego
SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
GO

 -- Restauramos FULL (sin recuperar todavía) --
RESTORE DATABASE sistema_riego
FROM DISK = 'C:\Backups\sistema_riego_FULL.bak'
WITH NORECOVERY,
     REPLACE,
     STATS = 5;
GO

 -- Aplicamos LOG1 y RECOVERY --
RESTORE LOG sistema_riego
FROM DISK = 'C:\Backups\sistema_riego_LOG1.trn'
WITH RECOVERY,                -- La base queda ONLINE
     STATS = 5;
GO

ALTER DATABASE sistema_riego
SET MULTI_USER;
GO

USE sistema_riego;
GO
```

Realizamos el backup de manera secuencial, empezando por el base (FULL) y luego el LOG1. Para ello, como vimos antes, respetamos la cadena ejecutando el full con NORECOVERY y RECOVERY en LOG1.

Resultado esperado: 110 registros. Verificamos:

```
SELECT COUNT(*) AS CantZonaRiego
FROM ZonaRiego;
GO
```

Results | Messages

CantZonaRiego
110

Restauración FULL + LOG1 + LOG2

Siguiendo con la restauración anterior, para restaurar el LOG2 lo hacemos de manera secuencial

```
ALTER DATABASE sistema_riego
SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
GO

RESTORE DATABASE sistema_riego
FROM DISK = 'C:\Backups\sistema_riego_FULL.bak'
WITH NORECOVERY,
     REPLACE,
     STATS = 5;
GO

RESTORE LOG sistema_riego
FROM DISK = 'C:\Backups\sistema_riego_LOG1.trn'
WITH NORECOVERY,
     STATS = 5;
GO

RESTORE LOG sistema_riego
FROM DISK = 'C:\Backups\sistema_riego_LOG2.trn'
WITH RECOVERY,
     STATS = 5;
GO
```

Realizamos el backup, empezando por el base (FULL), luego LOG1 Y finalmente LOG2. Para ello, como vimos antes, respetamos la cadena ejecutando el full con NORECOVERY , NORECOVERY en LOG1 y RECOVERY en LOG2.

Resultado esperado: 120 registros. Verificamos:

```
SELECT COUNT(*) AS CantZonaRiego
FROM ZonaRiego;
GO
```

Results | Messages

	CantZonaRiego
1	120

Capítulo V: Conclusiones

Respecto al primer tema, los **procedimientos** son ideales para operaciones CRUD y lógica de negocio, las **funciones** son mejores para cálculos y reportes dentro de consultas, el **uso combinado** mejora la modularidad, seguridad y mantenibilidad del sistema y las pruebas, en nuestro caso, mostraron una **diferencia mínima** en rendimiento, lo que justifica su uso en sistemas productivos.

Respecto al segundo tema, las pruebas realizadas demuestran claramente el **impacto de los índices en el rendimiento**. El Table Scan inicial fue el más costoso en tiempo y lecturas. El índice clustered redujo drásticamente las lecturas, aunque incrementó el tiempo total debido al costo físico del índice. Finalmente, el índice nonclustered con columnas incluidas ofreció el mejor rendimiento global, minimizando el costo de lectura y acelerando la ejecución.

Respecto al tercer tema, durante las pruebas se comprobó que **la transacción mantiene la atomicidad**: si 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.

Respecto al cuarto tema, el modelo de recuperación **FULL es esencial para permitir backups en línea** y restauraciones detalladas mediante archivos de log, la estrategia de respaldo utilizada demostró ser consistente, ya que el estado inicial (100 registros) fue restaurado correctamente desde el FULL y las restauraciones intermedias utilizando backups de LOG permitieron recuperar estados exactos después de los primeros 10 y 20 inserts. Se comprobó que **SQL Server aplica estrictamente la cadena de restauración**: si se intenta restaurar un log que no corresponde al LSN adecuado, el motor devuelve un error, protegiendo la integridad del sistema. El proceso realizado garantiza que en un entorno real se podría recuperar la base de datos sin pérdida de información, incluso si ocurre un fallo entre inserciones.

En conclusión, el diseño de un sistema de base de datos para el **control de riegos inteligentes** demostró ser una solución efectiva para organizar y centralizar la

información agrícola. El modelo relacional implementado permitió registrar parcelas, sensores y consumos de agua, generando consultas que facilitan la toma de decisiones. Se comprobó que la digitalización de los registros de riego contribuye a un uso más eficiente del agua, mejora la productividad y ofrece una base sólida para incorporar nuevas tecnologías en el futuro.

Capítulo VI: Bibliografía

- Instituto Nacional de Tecnología Agropecuaria – INTA. 2021. *Sistemas de riego y agricultura de precisión en la provincia de Corrientes*. Publicación Técnica INTA. https://www.argentina.gob.ar/informacion-agroclimatica
- BBVA. 2024. ¿Qué es la agricultura de precisión? La gestión digital del campo. ¿Qué es la agricultura de precisión? La gestión digital del campo
- Allan Ouko. 2025. *SQL Stored Procedure: Automate and Optimize Queries*. Learn the basics of SQL stored procedures and how to implement them in different databases, including MySQL and SQL Server. SQL Stored Procedure: Automate and Optimize Queries | DataCamp
- GeeksForGeeks. 2025. *SQL Stored Procedures*. SQL Stored Procedures - GeeksforGeeks
- Microsoft Learn. 2025. *FUNCIÓN CREATE (Transact-SQL)*. FUNCIÓN CREATE (Transact-SQL) - SQL Server | Microsoft Learn
- Robert Sheldon. 2022. Working with MySQL stored functions. Working with MySQL stored functions - Simple Talk
- Allan Ouko. 2025. Dominar los índices de SQL Server: Aumentar el rendimiento de la base de datos. Índice SQL Server: La clave para consultas más rápidas | DataCamp
- Oluseye Jeremiah. 2025. Comprender las Transacciones SQL: Guía completa. Comprender las Transacciones SQL: Guía completa | DataCamp
- Microsoft Learn. 2025. BACKUP (Transact-SQL). BACKUP (Transact-SQL) - SQL Server | Microsoft Learn
- Microsoft Learn. 2025. Instrucciones RESTORE (Transact-SQL). RESTORE (Transact-SQL) - SQL Server | Microsoft Learn