FaCENA - UNNE

Facultad de Ciencias Exactas y Naturales y Agrimensura

UNIVERSIDAD NACIONAL DEL NORDESTE

Proyecto de estudio: Sistema gestor de mantenimiento

Cátedra: Bases de Datos I

Integrantes:

- Jorge Ramón Montiel Nuñez
- Francisco Daniel Segovia
- Alexis Toledo Exequiel
- Renata Villalba Ruiz Diaz

Profesores:

-Lic. Dario O. VILLEGAS

-Lic. Walter O. VALLEJOS

-Exp. Juan Jose Cuzziol

-Lic. Numa Badaracco

Grupo: 29

Año: 2025

CAPÍTULO I: INTRODUCCIÓN

Caso de estudio

En este proyecto se desarrollará una base de datos orientada a la gestión del mantenimiento de maquinaria, con especial énfasis en la organización de tareas por parte de grupos técnicos. El sistema debe contemplar actividades clave como el registro de máquinas, la planificación de revisiones, la ejecución de reparaciones, la administración de repuestos y la asignación del personal técnico.

A través de esta base de datos se busca no solo almacenar la información operativa, sino también gestionarla de forma eficiente, asegurando trazabilidad de las intervenciones realizadas y control sobre los repuestos y reparaciones. El objetivo es mejorar la planificación, reducir errores administrativos y optimizar la toma de decisiones sobre el mantenimiento preventivo y correctivo.

Definición o planteamiento del problema

Actualmente, la gestión de mantenimiento de maquinaria presenta diversas limitaciones debido a que la información se maneja en registros dispersos y, en muchos casos, de forma manual. Esto provoca una serie de inconvenientes, entre los que se destacan la falta de control en los costos asociados a reparaciones y repuestos, la dificultad para organizar los grupos de técnicos y asignarles tareas específicas, y la ausencia de un historial centralizado de revisiones y diagnósticos de cada máquina. A esto se suman los riesgos de pérdida de información vinculada a la disponibilidad y ubicación de las instalaciones, así como la escasa trazabilidad en los procesos de reparación, lo cual repercute de manera directa en la eficiencia operativa. Todas estas deficiencias impactan negativamente en la productividad de la empresa, generando gastos innecesarios y limitando la capacidad de planificación.

Alcance

- Registro de máquinas con información de modelo, matrícula, marca e instalación correspondiente.

- Gestión de revisiones de maquinaria: fechas, diagnósticos asociados, grupos técnicos responsables y vinculación con reparaciones posteriores.

- Administración de reparaciones: fechas de inicio y fin, repuestos utilizados y grupos asignados (de ser necesaria una reparación).

- Registro de repuestos vinculados a cada máquina al realizar una reparación, incluyendo descripción.

- Control de grupos de técnicos: cantidad de integrantes y distribución de tareas.

- Gestión de técnicos: datos personales, grupo asignado y disponibilidad.

- Registro de instalaciones con datos de contacto y teléfonos asociados.
Límites

Límites

- No se incluye registro avanzado de rentabilidad o presupuestos de mantenimiento.

- No se contemplan métricas de desempeño individual de técnicos ni equipos de trabajo.

- No se contempla la gestión de proveedores externos de repuestos o servicios.

- No se implementa un sistema automatizado de compras para reposición de stock de repuestos.

Objetivo del Trabajo Práctico

Objetivos Generales

- Analizar las necesidades de gestión de mantenimiento de maquinaria, con foco en el control de costos y la organización técnica.

- Diseñar una base de datos que integre la información de máquinas, técnicos, repuestos, revisiones, diagnósticos y reparaciones.

- Brindar una solución que permita mejorar la eficiencia operativa y la trazabilidad de las intervenciones realizadas.

- Cumplimentar la investigación de los temas obligatorios proporcionados.

Objetivos Específicos

- Diseñar un modelo de datos que represente de manera clara las relaciones entre las entidades principales: máquinas, instalaciones, revisiones, diagnósticos, reparaciones, repuestos, técnicos y grupos.

- Implementar un sistema que permita registrar y consultar repuestos y reparaciones.

- Evaluar la eficacia del sistema en la reducción de errores administrativos y en la mejora de la planificación del mantenimiento.

- Aplicar técnicas de bases de datos relacionales para garantizar integridad, consistencia y seguridad de la información.

- Facilitar la trazabilidad histórica de cada máquina, contemplando diagnósticos, revisiones y reparaciones realizadas.

- Mejorar la asignación de tareas a grupos de técnicos, optimizando la distribución de recursos humanos.

CAPÍTULO II: MARCO CONCEPTUAL O REFERENCIAL

TEMA 1 “Manejo de transacciones y transacciones anidadas”

Transacciones en bases de datos

Una transacción en una Base de Datos (DB) es una unidad lógica de trabajo compuesta por una o más operaciones que deben ejecutarse completamente o no ejecutarse en absoluto, asegurando que la DB permanezca siempre en un estado consistente y los cambios sean permanentes una vez confirmados.

Propiedades ACID de una transacción

- **Atomicidad (Atomicity):** La transacción se trata como una unidad indivisible. Todas sus operaciones deben ejecutarse por completo o ninguna de ellas. Es la regla del "todo o nada".
- **Consistencia (Consistency):** Una transacción debe llevar la base de datos de un **estado válido a otro estado válido**. Mantiene las reglas y restricciones de la base de datos intactas.
- **Aislamiento (Isolation):** Las transacciones concurrentes no deben interferir entre sí. Las modificaciones de una transacción (T1) solo son visibles para otras transacciones (T2) después de que T1 haya realizado el `COMMIT` (confirmación).
- **Durabilidad (Durability):** Una vez que una transacción ha sido confirmada (`COMMIT`), sus cambios son permanentes. Las actualizaciones persisten y sobreviven a cualquier fallo o caída del sistema.

Estados de una Transacción

Las operaciones mencionadas hacen que las transacciones pasen por una serie de estados, denominados estados de una transacción, y que se ilustran en la figura:

- **Estado Activa**: Una transacción se inicia en el estado ACTIVA, donde realiza sus operaciones de LEER y ESCRIBIR.
- **Estado Parcialmente Confirmada**: Al completar estas operaciones, pasa a PARCIALMENTE CONFIRMADA. En este estado, los subsistemas de Control de Concurrencia y de Recuperación realizan verificaciones para:
    1. Asegurar que no haya interferido con otras transacciones.
    2. Registrar los cambios en la bitácora (`log`) del sistema para garantizar la durabilidad.
- **Estado Confirmada**: Si ambas verificaciones son exitosas, la transacción pasa a CONFIRMADA y sus cambios se hacen permanentes.
- **Estado Fallida**: Si la transacción falla (por una verificación fallida o si se aborta mientras estaba ACTIVA), pasa al estado FALLIDA. En este punto, debe ser cancelada (anulada o revertida) para deshacer sus efectos.
- **Estado Terminada**: Finalmente, la transacción pasa a TERMINADA, abandonando el sistema. Las transacciones fallidas pueden ser reiniciadas por el sistema o por el usuario como transacciones nuevas.

Operaciones de una Transacción

Para fines de recuperación es necesario saber cuándo se inicia, termina y confirma o aborta una transacción. Así, el gestor de concurrencia del SGBD debe controlar las siguientes operaciones:

- **BEGIN_TRANSACTION**: Indica el inicio de la transacción.
- **READ o WRITE**: Son las operaciones de acceso a datos (lectura o modificación) que se ejecutan dentro de la transacción.
- **END_TRANSACTION**: Señala el final de las operaciones de acceso. En este punto, el sistema verifica si los cambios pueden aplicarse de forma permanente o si deben desecharse.
- **COMMIT_TRANSACTION**: Indica el éxito. Los cambios realizados por la transacción se almacenan de forma permanente en la base de datos.
- **ROLLBACK o ABORT**: Indican un fallo. Se deshacen todos los cambios realizados para revertir la base de datos al estado anterior.

Operaciones de Transacciones en SQL Server

- **BEGIN TRANSACTION**: Esta instrucción se utiliza para marcar el punto de inicio de una transacción explícita en SQL Server.
- **COMMIT TRANSACTION**: Esta instrucción se utiliza para confirmar (hacer permanentes) los cambios realizados durante la transacción actual.
- **ROLLBACK TRANSACTION**: Esta instrucción se utiliza para deshacer (revertir) todos los cambios realizados durante la transacción actual.

- **SAVE TRANSACTION**: Esta instrucción se utiliza para crear un punto de guardado (**Savepoint**) dentro de una transacción, permitiendo deshacer cambios hasta ese punto específico sin afectar a toda la transacción.

Transacciones Explícitas e Implícitas

Una transacción se inicia automáticamente al ejecutar una sentencia de **SQL** (DML o DDL) de forma interactiva o por un programa cuando no hay una transacción ya en curso. Toda transacción finaliza con **`COMMIT`** (confirmar) o **`ROLLBACK`** (revertir). Estas operaciones pueden ser explícitas (ejecutadas directamente por el código) o implícitas (realizadas automáticamente por el sistema). Por defecto, el sistema aplica un `COMMIT` implícito si la transacción fue exitosa o un `ROLLBACK` implícito si se detectó algún fallo.

Transacciones Anidadas

Podemos anidar varias transacciones. Cuando anidamos varias transacciones, la instrucción `COMMIT` afectará a la última transacción abierta, pero `ROLLBACK` afectará a todas las transacciones abiertas. Un hecho a tener en cuenta es que si hacemos `ROLLBACK` de la transacción superior, se desharán también los cambios de todas las transacciones internas, aunque hayamos realizado `COMMIT en alguna` de ellas.

En SQL Server, las transacciones anidadas se gestionan con la variable `@@TRANCOUNT` (contador de transacciones). `BEGIN TRAN` aumenta `@@TRANCOUNT`, y `COMMIT TRAN` lo disminuye, pero los cambios sólo se vuelven permanentes cuando `@@TRANCOUNT` llega a cero (la transacción más externa se confirma). Si se ejecuta un `ROLLBACK` (en cualquier nivel de anidamiento), se deshacen todos los cambios realizados desde la primera transacción externa, y la variable `@@TRANCOUNT` se reinicia a cero. Es decir, un *rollback* interno deshace toda la transacción principal, ignorando cualquier *commit* parcial que se haya realizado en los niveles inferiores. Se pueden usar *SAVEPOINTS* para marcar puntos específicos dentro de una transacción anidada, permitiendo deshacer hasta esos puntos sin afectar a toda la transacción.

TEMA 2 “Procedimientos y funciones almacenadas”

Procedimiento

Un procedimiento almacenado (stored procedure en inglés) es un programa (o procedimiento) almacenado físicamente en una base de datos. Su implementación varía de un gestor de bases de datos a otro. La ventaja
de un procedimiento almacenado es que al ser ejecutado, en respuesta a una petición de usuario, es ejecutado directamente en el motor de bases de datos, el cual usualmente corre en un servidor separado. Como

tal, posee acceso directo a los datos que necesita manipular y sólo necesita enviar sus resultados de regreso al usuario, deshaciéndose de la sobrecarga resultante de comunicar grandes cantidades de datos salientes
y entrantes.

Ventajas de usar procedimientos almacenados

Tráfico de red reducido entre el cliente y el servidor

Los comandos de un procedimiento se ejecutan en un único lote de código. Esto puede
reducir significativamente el tráfico de red entre el servidor y el cliente porque
únicamente se envía a través de la red la llamada que va a ejecutar el procedimiento.

Mayor seguridad

Los usuarios y aplicaciones pueden operar sobre los objetos de la base a través del procedimiento sin tener permisos directos. El procedimiento controla las operaciones y protege los objetos, evitando otorgar
permisos en cada nivel y simplificando la seguridad.

Reutilización del código

Las operaciones repetitivas pueden encapsularse en procedimientos, evitando escribir el mismo código, reduciendo inconsistencias y permitiendo que cualquier usuario autorizado ejecute esa lógica.
Mantenimiento más sencillo

Al centralizar la lógica en procedimientos, los cambios se aplican solo en la base de datos. Las aplicaciones cliente no necesitan modificarse ni conocer cambios en diseños o procesos internos.

Rendimiento mejorado

De forma predeterminada, un procedimiento se compila la primera vez que se ejecuta y
crea un plan de ejecución que vuelve a usarse en posteriores ejecuciones. Como el procesador de consultas no tiene que crear un nuevo plan, normalmente necesita menos tiempo para procesar el procedimiento

Sintaxis Básica de un Procedimiento Almacenado

```
.CREATE PROCEDURE NombreDelProcedimiento
    @Parametro1 TipoDato,
    @Parametro2 TipoDato,
    @Resultado TipoDato OUTPUT
AS
BEGIN
    -- Instrucciones SQL (Se pone un ejemplo)
    SELECT *
```

```
    FROM Tabla
    WHERE Columna1 = @Parametro1
      AND Columna2 = @Parametro2;

    -- Asignación al parámetro de salida
    SET @Resultado = (SELECT COUNT(*)
                      FROM Tabla
                      WHERE Columna1 = @Parametro1
                        AND Columna2 = @Parametro2);
END;
```

-Pueden tener parámetros de entrada y salida (`IN` y `OUT`), permitiendo enviar valores hacia y desde el procedimiento.
- En el cuerpo, los procedimientos pueden ejecutar una amplia gama de operaciones, incluidas consultas de modificación de datos (`INSERT`, `UPDATE`, `DELETE`).
- Generalmente, la devolucion de valores se hace a través de parámetros de salida y no con una instrucción RETURN como en las funciones. Los procedimientos no tienen un tipo de retorno fijo.
- Los procedimientos pueden contener estructuras de control como `IF`, `WHILE` y `TRYCATCH`.

Cambiar un procedimiento (ALTER PROCEDURE)

Se utiliza la instrucción ALTER PROCEDURE NombreProcedimiento para modificar un procedimiento almacenado existente. La sintaxis es similar a la de CREATE PROCEDURE, pero se utiliza ALTER en lugar de CREATE.

Eliminar un procedimiento (DROP PROCEDURE)

Se utiliza la instrucción DROP PROCEDURE NombreProcedimiento para eliminar un procedimiento almacenado existente de la base de datos.

Funciones Almacenadas

Una Función Almacenada es un bloque de código SQL precompilado y almacenado en el servidor de la base de datos, diseñado principalmente para el propósito de reutilización de código y la ejecución de tareas
específicas. Siempre acepta parámetros de entrada y está obligada a devolver un único valor (escalar o una tabla).

Sintaxis Básica de una Función Almacenada

Definición de una Función Escalar

```
CREATE FUNCTION NombreDeLaFuncion
    (@Parametro1 TipoDato,
     @Parametro2 TipoDato)
```

```
RETURNS TipoDeRetorno -- Obligatorio: debe especificar el tipo de dato
que devuelve (ej. INT, VARCHAR, DECIMAL)
AS
BEGIN
    DECLARE @Resultado TipoDeRetorno;

    -- Lógica de cálculo o consulta
    SET @Resultado = (
        SELECT COUNT(*)
        FROM Tabla
        WHERE Columna1 = @Parametro1
          AND Columna2 = @Parametro2
    );

    RETURN @Resultado; -- Obligatorio: debe finalizar con la sentencia
RETURN
END;
```

Definición de una Función que Devuelve una Tabla

```
CREATE FUNCTION NombreDeFuncionTabla
    (@Parametro1 TipoDato)
RETURNS TABLE -- Indica que el resultado es una tabla
AS
RETURN
(
    -- La lógica debe ser una única sentencia SELECT
    SELECT ColumnaA, ColumnaB, ColumnaC
    FROM OtraTabla
    WHERE ColumnaFiltro = @Parametro1
);
```

Cambiar una Función (ALTER FUNCTION)

Se utiliza la instrucción ALTER FUNCTION NombreDeLaFuncion para modificar una función almacenada existente. La sintaxis es la misma que la de CREATE FUNCTION, pero se utiliza ALTER en lugar de CREATE.

Eliminar una Función (DROP FUNCTION)

Se utiliza la instrucción DROP FUNCTION NombreDeLaFuncion para eliminar una función almacenada existente de la base de datos.

Diferencia entre Procedimientos y Funciones Almacenadas

La distinción clave con los procedimientos almacenados radica en que la función no puede modificar objetos de la base de datos ni manejar transacciones, lo que limita su uso a operaciones de cálculo y recuperación
de datos. Además, su valor de retorno único permite que sea utilizada directamente dentro de expresiones SQL (como en SELECT o WHERE), algo que los procedimientos no pueden hacer al no devolver un valor simple,
requiriendo una sentencia de ejecución (EXECUTE) separada.

TEMA 3 “Optimización de consultas a través de índices”

Índices

Los índices son estructuras especializadas asociadas a una tabla o vista que permiten mejorar significativamente el rendimiento de las consultas. Funcionan generando claves a partir de una o varias columnas, las cuales se organizan de manera eficiente para acelerar la localización de los datos.

Al igual que el índice de un libro evita revisar cada página para encontrar un tema específico, un índice en la base de datos evita recorrer toda la tabla. Sin un índice, el motor debe realizar un Table Scan, examinando fila por fila hasta encontrar la información solicitada. En cambio, con un índice adecuado, se lleva a cabo un Index Seek, limitando la búsqueda únicamente a las filas relacionadas con los valores clave, reduciendo el tiempo de ejecución y optimizando el uso de recursos.

Tipos de índices

Dependiendo del propósito y de la estructura física requerida, los motores de base de datos ofrecen distintos tipos de índices. Entre los más relevantes para SQL Server se encuentran:

- **Clustered (agrupado):** reorganiza físicamente la tabla siguiendo el orden de la columna clave. Solo puede existir uno por tabla, ya que determina cómo se almacenan las filas. Es ideal para consultas por rango o columnas usadas en filtros frecuentes.
- **Nonclustered (no agrupado):** estructura separada de la tabla que almacena las claves del índice y un puntero hacia cada fila de datos. Permite acelerar consultas específicas sin modificar el orden físico de la tabla.
- **Hash:** utilizado en sistemas in-memory; permite búsquedas exactas con alta eficiencia.
- **Índice no agrupado optimizado para memoria:** consume memoria según la cantidad de filas y las columnas clave utilizadas.
- **Unique:** asegura que no existan valores duplicados en la columna o conjunto de columnas indexadas. Puede aplicarse tanto a índices agrupados como no agrupados.
- **Columnstore:** almacena los datos por columnas y permite una compresión elevada, orientado a cargas analíticas y consultas sobre grandes volúmenes.

- **Índice con columnas incluidas:** amplía el nivel hoja de un índice no agrupado para cubrir consultas completas sin necesidad de volver a la tabla base.
- **Índice sobre columnas calculadas:** basado en expresiones derivadas de columnas existentes, útil cuando una consulta utiliza cálculos repetitivos.
- **Filtered:** indexa únicamente un subconjunto de filas según una condición, reduciendo espacio y mejorando la eficiencia en consultas filtradas.
- **Spatial, XML y Full-text:** orientados a la búsqueda y manipulación de datos espaciales, documentos XML y texto no estructurado.

¿Cuándo conviene crear un índice?

Si bien los índices mejoran el rendimiento de lectura, también introducen un costo adicional en las operaciones de escritura (INSERT, UPDATE, DELETE), ya que cada modificación debe actualizar la estructura del índice. Por ello, su creación debe ser planificada estratégicamente.

Es recomendable crear un índice cuando:

- La columna se utiliza frecuentemente en cláusulas **WHERE**.
- La columna participa en **ORDER BY**.
- Se realizan consultas por rango (**BETWEEN**, <, >).
- La tabla contiene un volumen considerable de registros.
- Las consultas se repiten sobre las mismas columnas.

En cambio, no se aconseja crear índices en:

- Columnas con **baja cardinalidad** (pocos valores distintos).
- Tablas **pequeñas**.
- Columnas que se **modifican constantemente**, ya que el costo de mantenimiento puede superar los beneficios.

Cómo usa los índices el optimizador de consultas

Los índices bien diseñados pueden reducir significativamente las operaciones de E/S y el uso de recursos, mejorando el rendimiento de instrucciones **SELECT**, **UPDATE**, **DELETE** o **MERGE**.

Cuando una consulta se ejecuta, el optimizador evalúa todos los métodos posibles para recuperar los datos y elige el más eficiente. Entre las estrategias disponibles se encuentran:

Recorrido de tabla (Table Scan)

El optimizador lee todas las filas de la tabla y filtra las que cumplen con la condición.

Es costoso en términos de E/S, aunque puede ser eficiente si el porcentaje de filas devuelto es muy alto.

Búsqueda mediante índices (Index Seek / Index Scan)

Cuando la tabla posee índices adecuados, el optimizador puede:

- Buscar directamente el valor en la estructura del índice.
- Saltar a la ubicación física donde se encuentran las filas necesarias.
- Extraer únicamente las filas relevantes, reduciendo drasticamente el tiempo de ejecución.

Los índices suelen ser más rápidos porque contienen pocas columnas y están ordenados, lo que facilita el acceso secuencial o la búsqueda directa.

TEMA 4 “Manipulación de datos JSON”

¿Qué es JSON?

JSON es un formato de texto ampliamente utilizado para el intercambio de datos en aplicaciones web y móviles. También se emplea para almacenar información no estructurada en archivos de registro y bases NoSQL como Azure Cosmos DB. Numerosos servicios REST devuelven o reciben datos en formato JSON; entre ellos, varios servicios de Azure, como Azure Search, Azure Storage y Azure Cosmos DB. Además, JSON es el formato principal para el intercambio de datos entre páginas web y servidores mediante llamadas AJAX.

Las funciones JSON incorporadas en SQL Server a partir de la versión 2016 permiten integrar conceptos NoSQL dentro de una base de datos relacional. Con estas funciones es posible combinar columnas tradicionales con columnas que almacenan documentos JSON, transformar JSON en estructuras relacionales y generar JSON a partir de datos relacionales.

Estructura de datos en formato JSON

JSON funciona mediante la representación de datos de forma jerárquica, utilizando pares clave-valor para almacenar información. Los datos JSON se colocan entre llaves ({}), y cada par clave-valor se separa con una coma (,). Por ejemplo, el siguiente JSON representa la información de contacto de una persona:

```
{
  "nombre": "María González",
  "edad": 28,
  "ciudad": "Buenos Aires",
  "telefono": "01145678932",
```

```
  "correo": "maria.gonzalez@ejemplo.com"
}
```

En este caso, "**nombre**", "**edad**", "**ciudad**", "**telefono**" y "**correo**" son las claves, mientras que "**María González**", **28**, "**Buenos Aires**", "**01145678932**" y "**maria.gonzales@ejemplo.com**" son los valores correspondientes.

Uso del formato JSON en SQL Server

SQL Server no tiene un tipo de dato exclusivo llamado JSON, pero permite almacenar JSON en columnas NVARCHAR y procesarlo mediante funciones nativas, algunas de estas funciones nativas son:

- Crear objetos JSON
  El JSON se guarda en columnas NVARCHAR:

```
CREATE TABLE Tabla (
    id INT IDENTITY PRIMARY KEY,
    json NVARCHAR(tamaño)
);
```

- Validar JSON

  SQL Server permite verificar si el contenido es un JSON válido:

```
SELECT ISJSON(json) AS EsValido
FROM Tabla;
```

- Extraer valores

  Devuelve un valor simple (texto, número, booleano) desde una ruta JSON. Si el valor es un objeto o arreglo, devuelve NULL.

```
SELECT JSON_VALUE(json, '$.nombre') AS Nombre
FROM Tabla;
```

- Modificar contenido
  Se usa para obtener porciones completas del JSON, como objetos anidados o arreglos. Devuelve un JSON válido.

```
SELECT JSON_QUERY(json, '$.direccion') AS DireccionCompleta
```

```
FROM Tabla;
```

- Convertir arrays JSON en filas

  Interpreta contenido JSON (arreglo u objeto) y lo convierte en una tabla de filas y columnas. Permite expandir arreglos y crear resultados tabulares.

```
SELECT *
FROM OPENJSON('[
  {"id":1, "nombre":"Juan"},
  {"id":2, "nombre":"Ana"}
]');
```

- Actualizar valores o agregar propiedades

```
JSON_MODIFY ( Expresion, Ruta, Nuevo valor)
```

CAPÍTULO III: METODOLOGÍA SEGUIDA

a. Descripción de cómo se realizó el Trabajo Práctico.

La definición previa objeto de investigación del trabajo práctico se realizó mediante la evaluación de varias ideas propuestas por cada integrante. Posteriormente se profundizó en las incumbencias de este objeto de estudio con sugerencias de ideas y sugerencias de cada uno. Luego se dividió un tema de investigación por cada integrante, solicitando ayuda en caso de ser necesario.

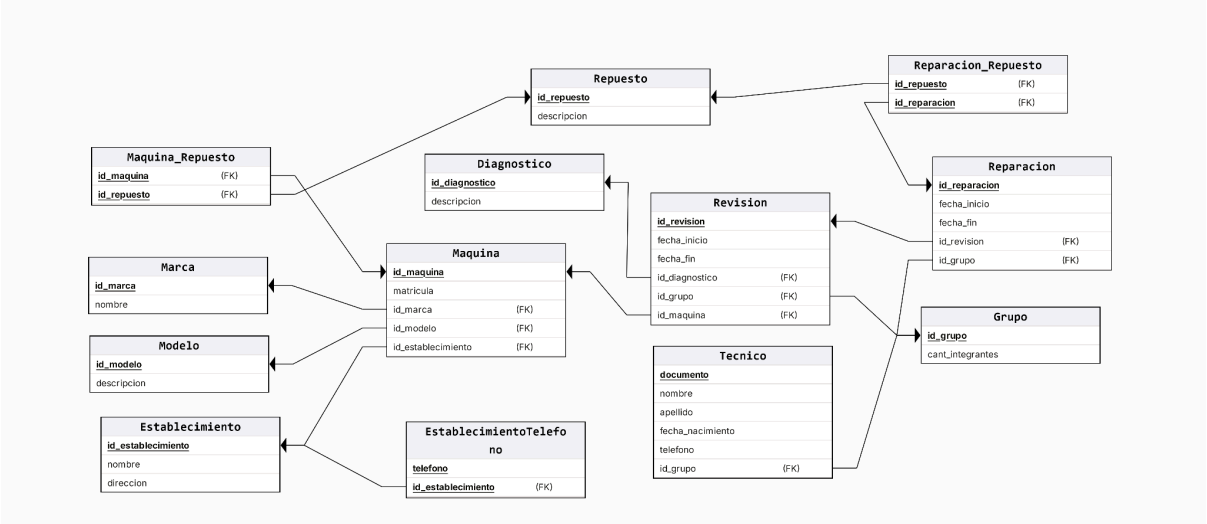
b. Herramientas (Instrumentos y procedimientos)

Las principales herramientas e instrumentos utilizados durante el desarrollo del trabajo fueron:
ERDPlus: Utilizado para la creación del diagrama relacional de la base de datos.
Microsoft SQL Server: Fue el sistema gestor de bases de datos utilizado para desplegar la base de datos del proyecto y realizar las pruebas respectivas.
Github: Utilizado para guardar y obtener la información obtenida junto a los scripts referidos a cada tema.
Microsoft Visual Studio: Fue utilizado en gran parte para la elaboración de los scripts por su facilidad para conectarse al repositorio Github de forma gráfica.
Google Docs: Para la unificación de la información y resultados obtenidos por cada uno en un único documento final.

La información se obtuvo en gran parte de consultas de material bibliográfico y recursos confiables, priorizando fuentes académicas.

CAPÍTULO IV: DESARROLLO DEL TEMA / PRESENTACIÓN DE RESULTADOS

Diagrama relacional



Diccionario de datos

Tabla: Marca			
Campo	Tipo	Nulo	Descripción
id_marca	INT IDENTITY	NO	Identificador de la marca (PK)
nombre	VARCHAR(100)	NO	Nombre de la marca

Tabla: Modelo			
Campo	Tipo	Nulo	Descripción
id_modelo	INT IDENTITY	NO	Identificador del modelo (PK)
descripcion	VARCHAR(100)	NO	Descripción del modelo

Tabla: Establecimiento			
Campo	Tipo	Nulo	Descripción
id_establecimiento	INT IDENTITY	NO	Identificador del establecimiento (PK)
nombre	VARCHAR(100)	NO	Nombre del establecimiento

direccion	VARCHAR(150)	NO	Dirección del establecimiento

Tabla: Maquina			
Campo	Tipo	Nulo	Descripción
id_maquina	INT IDENTITY	NO	Identificador de la máquina (PK)
matricula	CHAR(10)	NO	Matrícula única
id_modelo	INT	NO	FK al modelo
id_marca	INT	NO	FK a la marca
id_establecimiento	INT	SÍ	FK al establecimiento (opcional)

Tabla: EstablecimientoTelefono			
Campo	Tipo	Nulo	Descripción
telefono	VARCHAR(15)	NO	Teléfono del establecimiento
id_establecimiento	INT	NO	FK al establecimiento

Tabla: Repuesto			
Campo	Tipo	Nulo	Descripción
id_repuesto	INT IDENTITY	NO	Identificador del repuesto (PK)
descripcion	VARCHAR(150)	NO	Descripción del repuesto

Tabla: Maquina_Repuesto			
Campo	Tipo	Nulo	Descripción
id_maquina	INT	NO	FK a máquina
id_repuesto	INT	NO	FK a repuesto

Tabla: Diagnostico			
Campo	Tipo	Nulo	Descripción

id_diagnostico	INT IDENTITY	NO	Identificador (PK)
descripcion	VARCHAR(200)	NO	Descripción del diagnóstico

Tabla: Grupo			
Campo	Tipo	Nulo	Descripción
id_grupo	INT IDENTITY	NO	Identificador del grupo (PK)
cant_integrantes	INT	NO	Cantidad de integrantes del grupo

Tabla: Revision			
Campo	Tipo	Nulo	Descripción
id_revision	INT IDENTITY	NO	Identificador de la revisión (PK)
fecha_inicio	DATE	NO	Fecha de inicio
fecha_fin	DATE	SÍ	Fecha de fin
id_diagnostico	INT	SÍ	FK al diagnóstico
id_grupo	INT	NO	FK al grupo
id_maquina	INT	NO	FK a la máquina

Tabla: Reparacion			
Campo	Tipo	Nulo	Descripción
id_reparacion	INT IDENTITY	NO	Identificador de la reparación (PK)
fecha_inicio	DATE	NO	Fecha de inicio
fecha_fin	DATE	SÍ	Fecha de fin
id_revision	INT	NO	FK a la revisión
id_grupo	INT	NO	FK al grupo

Tabla: Tecnico			
Campo	Tipo	Nulo	Descripción
documento	INT	NO	DNI del técnico (PK)
nombre	VARCHAR(50)	NO	Nombre
apellido	VARCHAR(50)	NO	Apellido
fecha_nacimiento	DATE	NO	Fecha de nacimiento
telefono	VARCHAR(15)	SÍ	Teléfono

id_grupo	INT	NO	Grupo al que pertenece

Tabla: Reparacion_Repuesto			
Campo	Tipo	Nulo	Descripción
id_repuesto	INT	NO	FK a repuesto
id_reparacion	INT	NO	FK a reparación

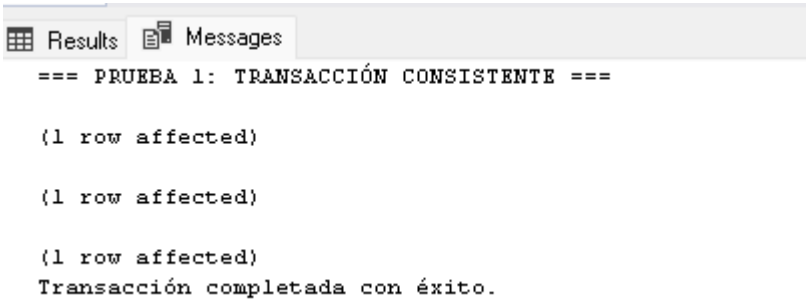
Desarrollo del tema 1: Transacciones y Transacciones Anidadas

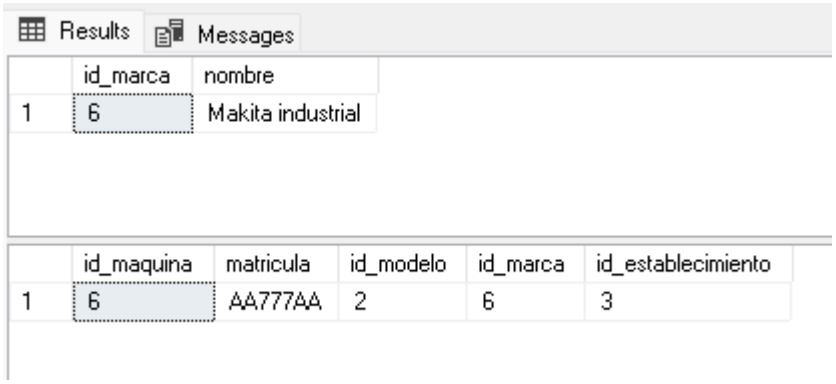
Se realizó un código de transacción simple para agregar registros en las tablas Marca y Maquina. Luego de que se agrega el registro de la tabla Maquina, su campo “nombre” se modifica. Si no ocurre ningún error, los cambios deben guardarse.

```
-- 1. TRANSACCIÓN CONSISTENTE

PRINT('=== PRUEBA 1: TRANSACCIÓN CONSISTENTE ===');
GO

BEGIN TRY
    BEGIN TRANSACTION;

    -- 1. Insertar un nuevo registro en la tabla Marca
    INSERT INTO Marca (nombre)
    VALUES ('Komatsu');

    DECLARE @idMarca INT = SCOPE_IDENTITY();

    -- 2. Insertar un registro en la tabla Maquina asociado a la nueva
marca
    INSERT INTO Maquina (matricula, id_modelo,
id_marca,id_establecimiento)
    VALUES ('AA777AA', '2', @idMarca,3);

    -- 3. Actualizar el nombre de la marca recientemente insertada
    UPDATE Marca
    SET nombre = 'Komatsu Industrial'
    WHERE id_marca = @idMarca;

    COMMIT TRANSACTION;
```

```
    PRINT('Transacción completada con éxito.');
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT('Error detectado. Se realizó ROLLBACK.');
    PRINT('Mensaje de error: ' + ERROR_MESSAGE());
END CATCH;
GO

-- Verificación de resultados
SELECT * FROM Marca WHERE nombre LIKE 'Komatsu%';
SELECT * FROM Maquina WHERE matricula = 'AA777AA';
GO
```

La transacción se completa exitosamente y los cambios se guardan



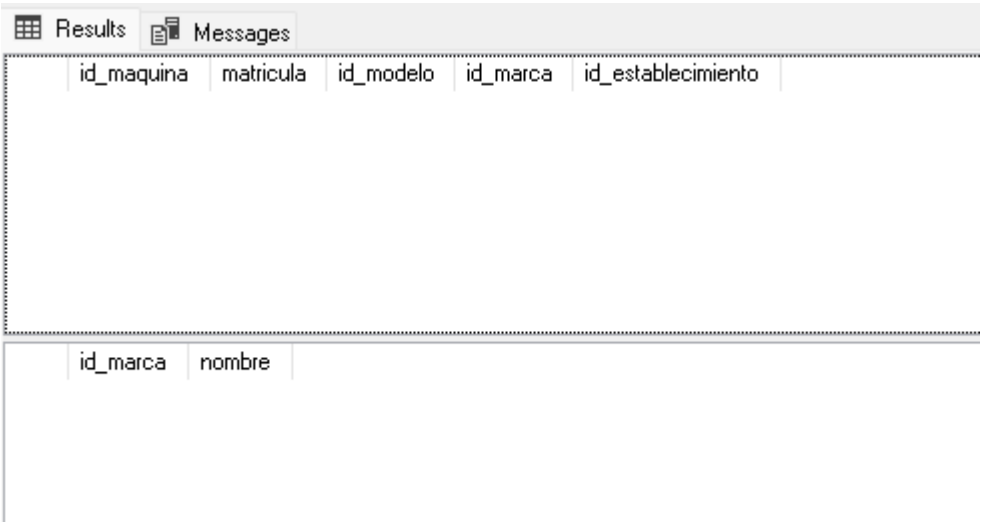


Posteriormente se creó otra prueba para comprobar qué pasa si una de las operaciones da un error, en el caso del código propuesto, se intenta hacer la inserción de una máquina con su campo id_modelo nulo, esto debería provocar un error que haga que consecuentemente se ejecute un ROLLBACK de la transacción principal (ninguno de los cambios se guardará)

```
-- 2. ERROR INTENCIONAL Y VERIFICACIÓN DE CONSISTENCIA

PRINT('=== PRUEBA 2: ERROR INTENCIONAL ===');

BEGIN TRY
```

```
    BEGIN TRANSACTION;

    -- 1. Insertar una nueva marca
    INSERT INTO Marca (nombre)
    VALUES ('Makita');

    DECLARE @idMarca2 INT = SCOPE_IDENTITY();

    -- 2. Provocar un error intencional: valor NULL en un campo
obligatorio (id_modelo)
    INSERT INTO Maquina (matricula, id_modelo, id_marca)
    VALUES ('BB888BB', NULL, @idMarca2);

    -- 3. Actualización que no se ejecutará si ocurre error
    UPDATE Marca
    SET nombre = 'BOBCAT'
    WHERE id_marca = @idMarca2;

    COMMIT TRANSACTION;
    PRINT('Transacción completada con éxito.');
    END TRY
    BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT('Error intencional detectado. Se realizó ROLLBACK.');
    PRINT('Mensaje de error: ' + ERROR_MESSAGE());
    END CATCH;
    GO

    -- Verificación: los registros no deben existir
    SELECT * FROM Maquina WHERE matricula = 'BB888BB';
    SELECT * FROM Marca WHERE nombre LIKE 'BOBCAT%';
```

```
=== PRUEBA 2: ERROR INTENCIONAL ===

(1 row affected)

(0 rows affected)
Error intencional detectado. Se realizó ROLLBACK.
```



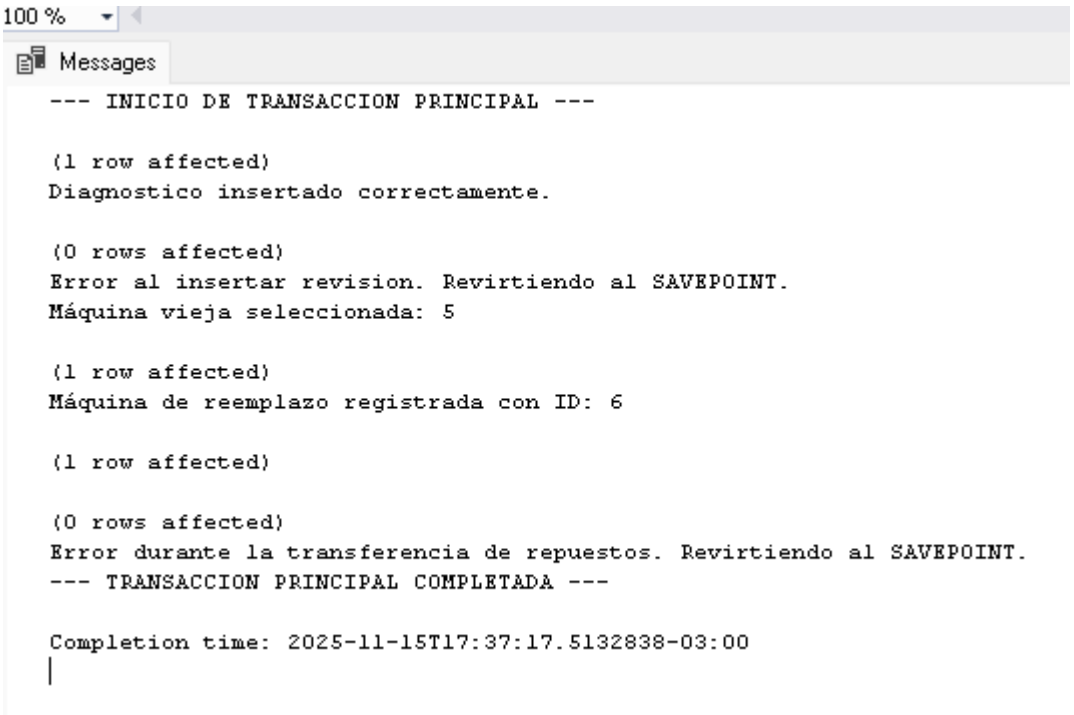
El código provocó que se ejecute un ROLLBACK de la transacción principal. No se agregó ningún registro en las tablas Maquina ni Marca.

Para probar el funcionamiento las transacciones anidadas, se creó el siguiente script que tiene una transacción principal para insertar un registro en la tabla Diagnostico y luego realizar una inserción en la tabla Revision por medio de una transacción anidada, previamente creando un SAVEPOINT. Posteriormente se captura el ID de la última máquina registrada, se inserta una nueva máquina y se ejecuta otra transacción para transferir los repuestos de la máquina vieja a la nueva. Se crea un SAVEPOINT antes de esta última transacción que contiene un error intencional. Por último se ejecuta COMMIT de la transacción principal o bien ROLLBACK en caso de que haya ocurrido un error y no fuese posible volver a un SAVEPOINT.

```
-- 3. TRANSACCIONES ANIDADAS CON EL USO DE SAVEPOINTS

BEGIN TRY
    BEGIN TRANSACTION TransGeneral;
    PRINT('--- INICIO DE TRANSACCIÓN PRINCIPAL ---');

    ------------------------------------------------------------
    -- 1. Insertar diagnostico
    ------------------------------------------------------------
    INSERT INTO Diagnostico (descripcion)
    VALUES ('Desgaste severo en componentes hidraulicos');

    DECLARE @idDiagnostico INT = SCOPE_IDENTITY();
    PRINT('Diagnostico insertado correctamente.');



    ------------------------------------------------------------
    -- 2. Transacción anidada 1 (Inserción de Revision)
    ------------------------------------------------------------
    SAVE TRANSACTION InsercionRevision;
```

```
    BEGIN TRY
        INSERT INTO Revision (fecha_inicio, fecha_fin, id_diagnostico,
id_grupo)
        VALUES (GETDATE(), GETDATE(), @idDiagnostico, 1);

        DECLARE @idRevision INT = SCOPE_IDENTITY();
        PRINT('Revisión creada exitosamente.');
    END TRY
    BEGIN CATCH
        PRINT('Error al insertar revision. Revirtiendo al SAVEPOINT.');
        ROLLBACK TRANSACTION InsercionRevision;
    END CATCH;


    ------------------------------------------------------------
    -- 3. Obtener máquina vieja (la última registrada)
    ------------------------------------------------------------
    DECLARE @idMaquinaVieja INT;

    SELECT @idMaquinaVieja = MAX(id_maquina)
    FROM Maquina;

    PRINT('Máquina vieja seleccionada: ' + CAST(@idMaquinaVieja AS
VARCHAR(10)));


    ------------------------------------------------------------
    -- 4. Insertar máquina nueva
    ------------------------------------------------------------
    INSERT INTO Maquina (matricula, id_modelo, id_marca)
    VALUES ('RPL-999', 1, 2);

    DECLARE @idNuevaMaquina INT = SCOPE_IDENTITY();
    PRINT('Máquina de reemplazo registrada con ID: ' +
CAST(@idNuevaMaquina AS VARCHAR(10)));


    ------------------------------------------------------------
    -- 5. Transacción anidada 2 (Transferencia de repuestos)
    ------------------------------------------------------------
    SAVE TRANSACTION TransferenciaRepuestos;

    BEGIN TRY
        INSERT INTO Maquina_Repuesto (id_maquina, id_repuesto)
        SELECT @idNuevaMaquina, id_repuesto
```

```
        FROM Maquina_Repuesto
        WHERE id_maquina = @idMaquinaVieja;

        -- Error intencional
        INSERT INTO Repuesto (descripcion) VALUES (NULL);

        DELETE FROM Maquina_Repuesto
        WHERE id_maquina = @idMaquinaVieja;

        PRINT('Repuestos transferidos correctamente.');
    END TRY
    BEGIN CATCH
        PRINT('Error durante la transferencia de repuestos. Revirtiendo
al SAVEPOINT.');
        ROLLBACK TRANSACTION TransferenciaRepuestos;
    END CATCH;


    ------------------------------------------------------------
    -- 6. Commit general
    ------------------------------------------------------------
    COMMIT TRANSACTION TransGeneral;
    PRINT('--- TRANSACCION PRINCIPAL COMPLETADA ---');
END TRY


------------------------------------------------------------
-- CATCH GENERAL
------------------------------------------------------------
BEGIN CATCH
    PRINT('Error general. Revirtiendo transaccion principal.');
    IF @@TRANCOUNT > 0
        ROLLBACK TRANSACTION TransGeneral;

    PRINT(ERROR_MESSAGE());
END CATCH;
```



Se observa claramente que las transacciones anidadas han revertido los cambios a los SAVEPOINT, los cambios contenido dentro de estas transacciones no deberían guardarse. Para comprobar qué cambios se guardaron y cuales no, se formuló el siguiente script:

```
--VERIFICACIÓN DE RESULTADOS DE LAS TRANSACCIONES ANIDADAS
-----------------------------------------------------------------
-- 1. Verificar último Diagnóstico insertado
-----------------------------------------------------------------
PRINT '--- Diagnóstico ---';

SELECT TOP 5 *
FROM Diagnostico
ORDER BY id_diagnostico DESC;



-----------------------------------------------------------------
-- 2. Verificar si se insertó la Revisión (puede haber rollback)
-----------------------------------------------------------------
PRINT '--- Revisión ---';

SELECT TOP 5 *
FROM Revision
ORDER BY id_revision DESC;



-----------------------------------------------------------------
-- 3. Verificar máquinas (la vieja y la nueva)
```

```
-----------------------------------------------------------------
PRINT '--- Máquinas ---';

SELECT TOP 5 *
FROM Maquina
ORDER BY id_maquina DESC;



-----------------------------------------------------------------
-- 4. Verificar repuestos asignados a la máquina nueva
-----------------------------------------------------------------
PRINT '--- Repuestos asociados a la nueva máquina ---';

DECLARE @ultimaMaquina INT;
SELECT @ultimaMaquina = MAX(id_maquina) FROM Maquina;

SELECT *
FROM Maquina_Repuesto
WHERE id_maquina = @ultimaMaquina;



-----------------------------------------------------------------
-- 5. Verificar si se insertó el repuesto inválido (NULL)
--    Debería NO existir por el rollback del SAVEPOINT
-----------------------------------------------------------------
PRINT '--- Repuestos (verificar que NO se insertó NULL) ---';

SELECT TOP 5 *
FROM Repuesto
ORDER BY id_repuesto DESC;
```

00 %

Results | Messages

	id_diagnostico	descripcion
1	6	Desgaste severo en componentes hidraulicos
2	5	Falla eléctrica intermitente
3	4	Ruidos anómalos en motor
4	3	Pérdida de potencia
5	2	Desgaste de orugas

	id_revision	fecha_inicio	fecha_fin	id_diagnostico	id_grupo	id_maquina
1	5	2024-05-20	NULL	5	5	5
2	4	2024-04-02	NULL	4	4	4
3	3	2024-03-11	2024-03-15	3	3	3
4	2	2024-02-05	2024-02-06	2	2	2
5	1	2024-01-10	2024-01-12	1	1	1

	id_maquina	matricula	id_modelo	id_marca	id_establecimiento
1	6	RPL-999	1	2	NULL
2	5	MAQ0005	5	5	5
3	4	MAQ0004	4	2	4
4	3	MAQ0003	3	4	3
5	2	MAQ0002	2	3	2

	id_maquina	id_repuesto

	id_repuesto	descripcion
1	5	Radiador reforzado
2	4	Bomba hidráulica
3	3	Juego de orugas
4	2	Batería industrial
5	1	Filtro de aceite

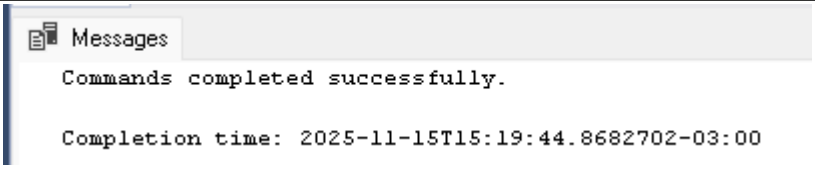
De los resultados del script se puede observar que el diagnóstico y la máquina se guardaron. Sin embargo la revisión no se guardó, ni se asignaron los repuestos de la máquina vieja a la máquina nueva. Tampoco se registró el repuesto que se intentó insertar con descripción nula.

Desarrollo del tema 2: Procedimientos y Funciones Almacenadas

Se crearon tres procedimientos, el primero permite la agregar una reparación y actualizar automáticamente la tabla Reparacion_Repuesto. El segundo procedimiento permite actualizar la fecha de finalización de una reparación. El tercer procedimiento permite eliminar una reparación y su relación con Repuesto eliminado el registro de Reparacion_Repuesto correspondiente.

```
--Procedimiento para agregar una reparación con repuesto (el id_repuesto
se pasa por parámetro y actualizar la tabla Reparacion_Repuesto)

CREATE PROCEDURE AgregarReparacionConRepuesto
    @fecha_inicio DATE,
    @fecha_fin DATE = NULL,
    @id_revision INT,
    @id_grupo INT,
    @id_repuesto INT
AS
BEGIN

    DECLARE @NuevoIdReparacion INT;

    INSERT INTO Reparacion (
        fecha_inicio,
        fecha_fin,
        id_revision,
        id_grupo
    )
    VALUES (
        @fecha_inicio,
        @fecha_fin,
        @id_revision,
        @id_grupo
    );

    SET @NuevoIdReparacion = SCOPE_IDENTITY();

    INSERT INTO Reparacion_Repuesto (
        id_repuesto,
        id_reparacion
    )
    VALUES (
        @id_repuesto,
        @NuevoIdReparacion
    );
```

```sql
END
GO


-- Procedimiento para actualizar la fecha de fin de una reparacion
CREATE PROCEDURE setFechaFinReparacion
    @id_reparacion INT,
    @fecha_fin DATE
AS
BEGIN
    SET NOCOUNT ON;

    UPDATE Reparacion
    SET fecha_fin = @fecha_fin
    WHERE id_reparacion = @id_reparacion;
END
GO

--Procedimiento para eliminar una reparacion junto con su relacion a un
repuesto asociado
CREATE PROCEDURE EliminarReparacionConRepuesto
    @id_reparacion INT
AS
BEGIN
    SET NOCOUNT ON;

    -- Primero se elimina la asiacion de repuestos a la reparación
    DELETE FROM Reparacion_Repuesto
    WHERE id_reparacion = @id_reparacion;

    -- Luego se elimna la reparación
    DELETE FROM Reparacion
    WHERE id_reparacion = @id_reparacion;
END
GO
```



Se realizó un lote de datos para insertar registros en la tabla reparación y su relación en la tabla intermedia Reparacion_Repuesto directamente con instrucciones de INSERT y otro lote datos haciéndolo mediante llamadas al procedimiento AgregarReparacionConRepuesto creado previamente. También se incluyeron contadores de tiempo para registrar el tiempo de ejecución de cada lote.

```
-------------------------------------------------------------------------
-
-- INSERTANDO DATOS DE PRUEBA EN LA TABLA REPARACION Y
REPARACION_REPUESTO DE FORMA MANUAL
-------------------------------------------------------------------------
-
PRINT '--- INICIO: Inserción Manual---';
DECLARE @NuevoIdReparacion_Manual INT;

BEGIN TRANSACTION;

INSERT INTO Reparacion (fecha_inicio, id_revision, id_grupo) VALUES
('2025-11-14', 1, 1);
SET @NuevoIdReparacion_Manual = SCOPE_IDENTITY();
INSERT INTO Reparacion_Repuesto (id_repuesto, id_reparacion) VALUES (1,
@NuevoIdReparacion_Manual);

INSERT INTO Reparacion (fecha_inicio, id_revision, id_grupo) VALUES
('2025-11-15', 2, 2);
SET @NuevoIdReparacion_Manual = SCOPE_IDENTITY();
INSERT INTO Reparacion_Repuesto (id_repuesto, id_reparacion) VALUES (2,
@NuevoIdReparacion_Manual);

INSERT INTO Reparacion (fecha_inicio, id_revision, id_grupo) VALUES
('2025-11-16', 3, 3);
SET @NuevoIdReparacion_Manual = SCOPE_IDENTITY();
INSERT INTO Reparacion_Repuesto (id_repuesto, id_reparacion) VALUES (3,
@NuevoIdReparacion_Manual);

INSERT INTO Reparacion (fecha_inicio, id_revision, id_grupo) VALUES
('2025-11-17', 4, 4);
SET @NuevoIdReparacion_Manual = SCOPE_IDENTITY();
INSERT INTO Reparacion_Repuesto (id_repuesto, id_reparacion) VALUES (4,
@NuevoIdReparacion_Manual);

INSERT INTO Reparacion (fecha_inicio, id_revision, id_grupo) VALUES
('2025-11-18', 5, 5);
SET @NuevoIdReparacion_Manual = SCOPE_IDENTITY();
INSERT INTO Reparacion_Repuesto (id_repuesto, id_reparacion) VALUES (5,
@NuevoIdReparacion_Manual);

INSERT INTO Reparacion (fecha_inicio, id_revision, id_grupo) VALUES
('2025-11-19', 1, 2);
SET @NuevoIdReparacion_Manual = SCOPE_IDENTITY();
INSERT INTO Reparacion_Repuesto (id_repuesto, id_reparacion) VALUES (1,
@NuevoIdReparacion_Manual);
```

```sql
INSERT INTO Reparacion (fecha_inicio, id_revision, id_grupo) VALUES
('2025-11-20', 2, 3);
SET @NuevoIdReparacion_Manual = SCOPE_IDENTITY();
INSERT INTO Reparacion_Repuesto (id_repuesto, id_reparacion) VALUES (2,
@NuevoIdReparacion_Manual);

INSERT INTO Reparacion (fecha_inicio, id_revision, id_grupo) VALUES
('2025-11-21', 3, 4);
SET @NuevoIdReparacion_Manual = SCOPE_IDENTITY();
INSERT INTO Reparacion_Repuesto (id_repuesto, id_reparacion) VALUES (3,
@NuevoIdReparacion_Manual);

INSERT INTO Reparacion (fecha_inicio, id_revision, id_grupo) VALUES
('2025-11-22', 4, 5);
SET @NuevoIdReparacion_Manual = SCOPE_IDENTITY();
INSERT INTO Reparacion_Repuesto (id_repuesto, id_reparacion) VALUES (4,
@NuevoIdReparacion_Manual);

INSERT INTO Reparacion (fecha_inicio, id_revision, id_grupo) VALUES
('2025-11-23', 5, 1);
SET @NuevoIdReparacion_Manual = SCOPE_IDENTITY();
INSERT INTO Reparacion_Repuesto (id_repuesto, id_reparacion) VALUES (5,
@NuevoIdReparacion_Manual);

COMMIT TRANSACTION;
PRINT '--- FIN: Inserción Manual ---';

PRINT '--- INICIO: INFORME DE REPARACIONES (INSERCIÓN MANUAL) ---';
SELECT
    R.id_reparacion,
    R.fecha_inicio,
    R.id_revision,
    R.id_grupo,
    RR.id_repuesto
FROM
    Reparacion AS R
INNER JOIN
    Reparacion_Repuesto AS RR ON R.id_reparacion = RR.id_reparacion
WHERE
    R.fecha_inicio BETWEEN '2025-11-14' AND '2025-11-23' -- Filtrar por
el rango de fechas de la inserción manual
ORDER BY
    R.id_reparacion;
PRINT '--- FIN: INFORME DE REPARACIONES (INSERCIÓN MANUAL) ---';
```

```sql
-----------------------------------------------------------------------
-
-- INSERTANDO DATOS DE PRUEBA EN LA TABLA REPARACION Y
REPARACION_REPUESTO LLMANDO A PROCEDURE AgregarReparacionConRepuesto
-----------------------------------------------------------------------
-
PRINT '--- INICIO: Inserción con procedimiento almacenado ---';

EXEC AgregarReparacionConRepuesto '2025-12-01', NULL, 1, 1, 5;

EXEC AgregarReparacionConRepuesto '2025-12-02', NULL, 2, 2, 4;

EXEC AgregarReparacionConRepuesto '2025-12-03', NULL, 3, 3, 3;

EXEC AgregarReparacionConRepuesto '2025-12-04', NULL, 4, 4, 2;

EXEC AgregarReparacionConRepuesto '2025-12-05', NULL, 5, 5, 1;

EXEC AgregarReparacionConRepuesto '2025-12-06', NULL, 1, 2, 5;

EXEC AgregarReparacionConRepuesto '2025-12-07', NULL, 2, 3, 4;

EXEC AgregarReparacionConRepuesto '2025-12-08', NULL, 3, 4, 3;

EXEC AgregarReparacionConRepuesto '2025-12-09', NULL, 4, 5, 2;

EXEC AgregarReparacionConRepuesto '2025-12-10', NULL, 5, 1, 1;
PRINT '--- FIN: Inserción con Stored Procedure ---';

PRINT '--- INICIO: INFORME DE REPARACIONES (INSERCIÓN CON PROCEDURE)
---';
SELECT
    R.id_reparacion,
    R.fecha_inicio,
    R.id_revision,
    R.id_grupo,
    RR.id_repuesto
FROM
    Reparacion AS R
INNER JOIN
    Reparacion_Repuesto AS RR ON R.id_reparacion = RR.id_reparacion
WHERE
    R.fecha_inicio BETWEEN '2025-12-01' AND '2025-12-10' -- Filtrar por
el rango de fechas de la inserción con procedure
ORDER BY
    R.id_reparacion;
```

```
PRINT '--- FIN: INFORME DE REPARACIONES (INSERCIÓN CON PROCEDURE) ---';
```

Los registros de ambos métodos se insertaron con éxito:

Results | Messages

	id_reparacion	fecha_inicio	id_revision	id_grupo	id_repuesto
1	6	2025-11-14	1	1	1
2	7	2025-11-15	2	2	2
3	8	2025-11-16	3	3	3
4	9	2025-11-17	4	4	4
5	10	2025-11-18	5	5	5
6	11	2025-11-19	1	2	1
7	12	2025-11-20	2	3	2
8	13	2025-11-21	3	4	3
9	14	2025-11-22	4	5	4

	id_reparacion	fecha_inicio	id_revision	id_grupo	id_repuesto
1	16	2025-12-01	1	1	5
2	17	2025-12-02	2	2	4
3	18	2025-12-03	3	3	3
4	19	2025-12-04	4	4	2
5	20	2025-12-05	5	5	1
6	21	2025-12-06	1	2	5
7	22	2025-12-07	2	3	4
8	23	2025-12-08	3	4	3
9	24	2025-12-09	4	5	2

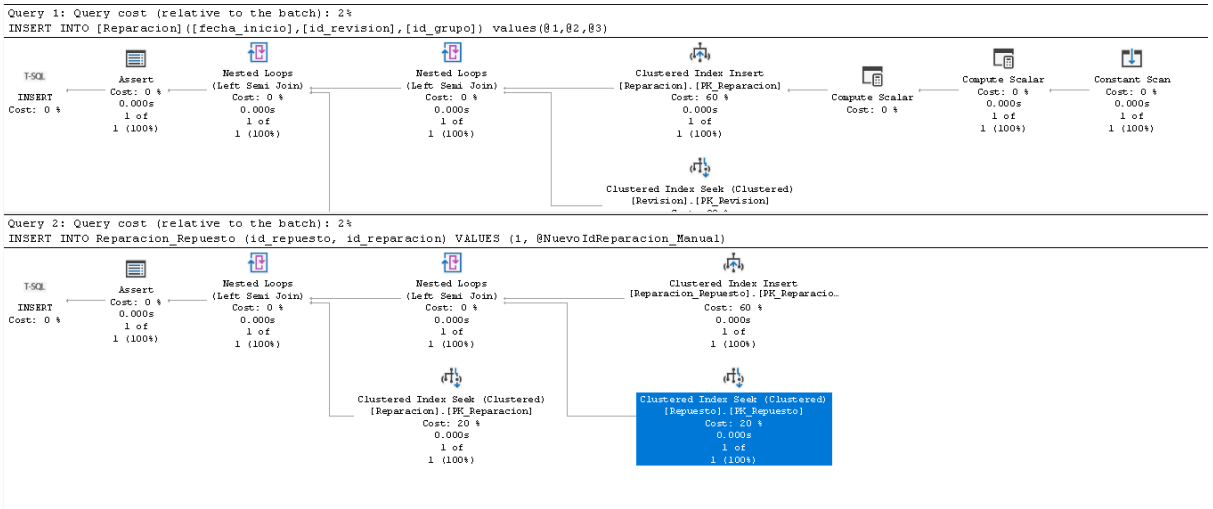
Tiempo de ejecución del lote de INSERTS manual:

```
(1 row affected)

(1 row affected)
--- FIN: Inserción Manual ---
TIEMPO TOTAL INSERT MANUAL (microsegundos): 997
```

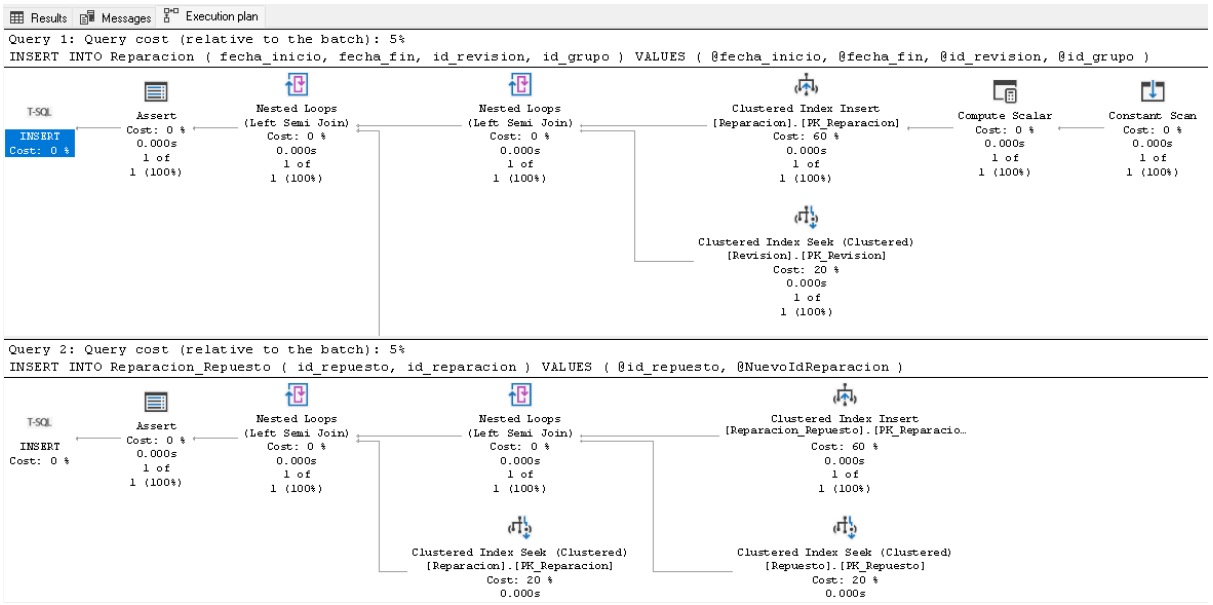
Tiempo de ejecución del lote llamando al procedimiento AgregarReparacionConRepuesto:

```
(1 row affected)
--- FIN: Inserción con Stored Procedure ---
TIEMPO TOTAL INSERT CON PROCEDURE (microsegundos): 2993
--- INICIO: INFORME DE REPARACIONES (INSERCIÓN CON PROCEDURE) ---
```

Plan de ejecución para el lote de INSERT (repetido N veces)



Plan de ejecución para el procedimiento (repetido N veces)

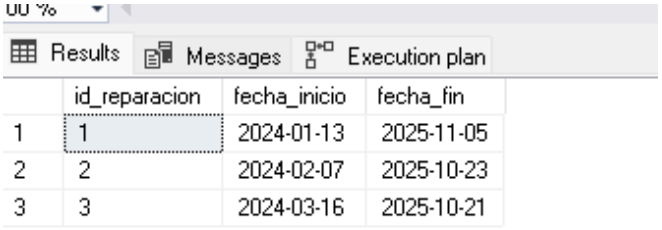


Se ejecutó el procedimiento setFechaFinReparacion y eliminarReparacionConRepuesto en un lote de reparaciones:

```
-- Ejecución del procedimiento de actualización
EXEC setFechaFinReparacion @id_reparacion = 1, @fecha_fin = '2025-11-5';
EXEC setFechaFinReparacion @id_reparacion = 2, @fecha_fin =
'2025-10-23';
EXEC setFechaFinReparacion @id_reparacion = 3, @fecha_fin =
'2025-10-21';
-- Verificación del cambio
SELECT
    id_reparacion,
    fecha_inicio,
    fecha_fin
FROM Reparacion
WHERE id_reparacion IN(1,2,3);
```

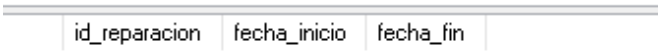
```sql
-- Ejecución del procedimiento de eliminación
EXEC eliminarReparacionConRepuesto @id_reparacion = 4;
EXEC eliminarReparacionConRepuesto @id_reparacion = 5;
EXEC eliminarReparacionConRepuesto @id_reparacion = 6;

-- Verificación de eliminacion en Reparacion
SELECT
    id_reparacion,
    fecha_inicio,
    fecha_fin
FROM Reparacion
WHERE id_reparacion IN (4, 5, 6);

-- Verificación de eliminacion en Reparacion_Repuesto
SELECT
    id_reparacion,
    id_repuesto
FROM Reparacion_Repuesto
WHERE id_reparacion IN (4, 5, 6);
```

Las llamadas a los procedimientos funcionaron correctamente, las fechas fueron actualizadas (primer SELECT) y los registros en las tablas Reparacion y Reparacion_Repuesto correspondientes fueron eliminadas (segundo y tercer SELECT):

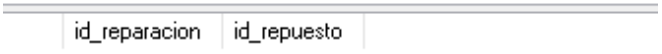


Results | Messages | Execution plan

	id_reparacion	fecha_inicio	fecha_fin
1	1	2024-01-13	2025-11-05
2	2	2024-02-07	2025-10-23
3	3	2024-03-16	2025-10-21



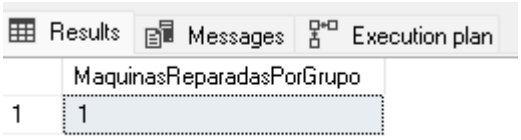
id_reparacion	fecha_inicio	fecha_fin



id_reparacion	id_repuesto

Se a creado la función cantMaquinasReparasGrupo, que devuelve la cantidad de maquinas que un grupo cuyo se pasa por parámetro a reparado. También se creó la función getHistorialReparacionesMaquina, que devuelve una tabla de informe con todas las reparaciones que ha recibido una maquina que se pasa por id. Por último se creo la función tieneReparacionActiva que comprueba si una máquina tiene una reparación que aún no ha terminado (fecha de finalización nula).

```sql
--Función que devuelve la cantidad de máquinas reparadas por un grupo
identificado por su id.
CREATE FUNCTION cantMaquinasReparadasGrupo
(
    @id_grupo INT
)
RETURNS INT
AS
BEGIN
    DECLARE @cantidad INT;

    SELECT @cantidad = COUNT(DISTINCT rv.id_maquina)
    FROM Grupo g
    INNER JOIN Reparacion r
        ON r.id_grupo = g.id_grupo
    INNER JOIN Revision rv
        ON rv.id_revision = r.id_revision
    WHERE g.id_grupo = @id_grupo
      AND r.fecha_fin IS NOT NULL          -- Reparación finalizada
      AND rv.fecha_fin IS NOT NULL;        -- Revisión finalizada

    RETURN @cantidad;
END;
GO

--Ejemplo de ejecucion de la funcion con id_grupo=3
SELECT dbo.cantMaquinasReparadasGrupo(3) AS [MaquinasReparadasPorGrupo];
```




```sql
--Función que devuelve una tabala con el historial de reparaciones de
una máquina dada su id.
CREATE FUNCTION getHistorialReparacionesMaquina (
    @idMaquina INT
)
RETURNS TABLE
AS
RETURN
(
    SELECT
        MOD.descripcion AS [Modelo maquina],
        M.matricula AS [Matricula maquina],
```

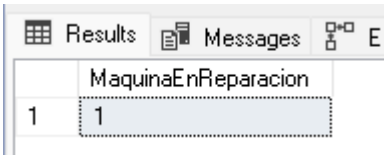
```
        E.nombre AS [Nombre establecimiento],
        R.fecha_inicio AS [Fecha inicio reparacion],
        R.fecha_fin AS [Fecha fin reparacion],
        G.id_grupo AS [ID grupo reparacion],
        DI.descripcion AS [Descripcion diagnostico],
        REP.descripcion AS [Repuesto utilizado]
    FROM
        Maquina M
    INNER JOIN
        Modelo MOD ON M.id_modelo = MOD.id_modelo
    INNER JOIN
        Revision REV ON M.id_maquina = REV.id_maquina
    INNER JOIN
        Reparacion R ON REV.id_revision = R.id_revision
    LEFT JOIN
        Establecimiento E ON M.id_establecimiento = E.id_establecimiento
    LEFT JOIN
        Diagnostico DI ON REV.id_diagnostico = DI.id_diagnostico
    LEFT JOIN
        Grupo G ON R.id_grupo = G.id_grupo
    LEFT JOIN
        Reparacion_Repuesto RR ON R.id_reparacion = RR.id_reparacion
    LEFT JOIN
        Repuesto REP ON RR.id_repuesto = REP.id_repuesto
    WHERE
        M.id_maquina = @idMaquina
);

--Ejemplo de ejecucion de la funcion con id_maquina=3
SELECT *
FROM dbo.getHistorialReparacionesMaquina(3);
```

Results | Messages | Execution plan

	Modelo maquina	Matricula maquina	Nombre establecimiento	Fecha inicio reparacion	Fecha fin reparacion	ID grupo reparacion	Descripcion diagnostico	Repuesto utilizado
1	Pala Cargadora L90	MAQ0003	Planta Sur	2024-03-16	2025-10-21	3	Pérdida de potencia	Juego de orugas
2	Pala Cargadora L90	MAQ0003	Planta Sur	2025-11-16	NULL	3	Pérdida de potencia	Juego de orugas
3	Pala Cargadora L90	MAQ0003	Planta Sur	2025-11-21	NULL	4	Pérdida de potencia	Juego de orugas
4	Pala Cargadora L90	MAQ0003	Planta Sur	2025-12-03	NULL	3	Pérdida de potencia	Juego de orugas
5	Pala Cargadora L90	MAQ0003	Planta Sur	2025-12-08	NULL	4	Pérdida de potencia	Juego de orugas
6	Pala Cargadora L90	MAQ0003	Planta Sur	2025-11-16	NULL	3	Pérdida de potencia	Juego de orugas
7	Pala Cargadora L90	MAQ0003	Planta Sur	2025-11-21	NULL	4	Pérdida de potencia	Juego de orugas
8	Pala Cargadora L90	MAQ0003	Planta Sur	2025-12-03	NULL	3	Pérdida de potencia	Juego de orugas
9	Pala Cargadora L90	MAQ0003	Planta Sur	2025-12-08	NULL	4	Pérdida de potencia	Juego de orugas
10	Pala Cargadora L90	MAQ0003	Planta Sur	2025-11-16	NULL	3	Pérdida de potencia	Juego de orugas
11	Pala Cargadora L90	MAQ0003	Planta Sur	2025-11-21	NULL	4	Pérdida de potencia	Juego de orugas
12	Pala Cargadora L90	MAQ0003	Planta Sur	2025-12-03	NULL	3	Pérdida de potencia	Juego de orugas
13	Pala Cargadora L90	MAQ0003	Planta Sur	2025-12-08	NULL	4	Pérdida de potencia	Juego de orugas
14	Pala Cargadora L90	MAQ0003	Planta Sur	2025-12-03	NULL	3	Pérdida de potencia	Juego de orugas
15	Pala Cargadora L90	MAQ0003	Planta Sur	2025-12-08	NULL	4	Pérdida de potencia	Juego de orugas

```
--Funcion que devuelve 1 si una máquina tiene reparaciones activas (sin
fecha de fin) o 0 si no la tiene
```

```
CREATE FUNCTION tieneReparacionActiva (@idMaquina INT)
RETURNS BIT
AS
BEGIN
    DECLARE @ReparacionesActivas INT;
    DECLARE @Resultado BIT;

    -- Cuenta el número de reparaciones activas para la máquina dada
    SELECT
        @ReparacionesActivas = COUNT(R.id_reparacion)
    FROM
        Maquina M
    INNER JOIN
        Revision REV ON M.id_maquina = REV.id_maquina
    INNER JOIN
        Reparacion R ON REV.id_revision = R.id_revision
    WHERE
        M.id_maquina = @idMaquina
        AND R.fecha_fin IS NULL; -- La fecha de fin NULL indica que la
reparación está activa

    -- Establece el resultado: 1 (True) si el conteo es mayor a 0, 0
(False) si es 0
    IF @ReparacionesActivas > 0
        SET @Resultado = 1;
    ELSE
        SET @Resultado = 0;

    RETURN @Resultado;
END;
--Ejemplo de ejecucion de la funcion con id_maquina=3
SELECT dbo.tieneReparacionActiva(3) AS [MaquinaEnReparacion];
```



Para comparar la eficiencia en términos de tiempo de ejecución de realizar operaciones con sentencias directas y llamando a una función, se creo el siguiente script que aplica las operaciones de la función cantMaquinasReparadasGrupo (manualmente) en los grupos del 1 al 5 en 100 iteraciones y luego aplica el llamado a la función en otras 100 iteraciones. El tiempo de ejecución de ambos métodos se registra para poder compararlos.

```
---------------------------
--PRUEBA DE EFICIENCIA DE OPERACIONES DIRECTAS VS FUNCIONES
```

```
--------------------------
SET NOCOUNT ON;

DECLARE @Iteraciones INT = 100;
DECLARE @Contador INT = 1;
DECLARE @GrupoID INT;
DECLARE @Inicio DATETIME;
DECLARE @Fin DATETIME;
DECLARE @Cantidad INT; -- Variable para almacenar la cantidad de
máquinas reparadas


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

--Funcion para comparar entre realizar operaciones directas y utilizar
funciones almacenadas.
-- 1 PRUEBA DE CONTEO DIRECTO (Manual)

PRINT '--- INICIANDO PRUEBA DE CONTEO DIRECTO (MANUAL) ---';
SET @Inicio = GETDATE();
SET @Contador = 1;

WHILE @Contador <= @Iteraciones
BEGIN
    -- Rota el ID de Grupo entre 1 y 5
    SET @GrupoID = (@Contador - 1) % 5 + 1;

    -- Ejecución Manual de la Consulta
    SELECT @Cantidad = COUNT(DISTINCT rv.id_maquina)
    FROM Grupo g
    INNER JOIN Reparacion r
        ON r.id_grupo = g.id_grupo
    INNER JOIN Revision rv
        ON rv.id_revision = r.id_revision
    WHERE g.id_grupo = @GrupoID
      AND r.fecha_fin IS NOT NULL
      AND rv.fecha_fin IS NOT NULL;

    -- Muestra el resultado de la iteración con el número de iteración
    PRINT 'Manual - Grupo ID: ' + CAST(@GrupoID AS VARCHAR) + ' -
Cantidad: ' + CAST(@Cantidad AS VARCHAR) + ' | Iteracion Nro: ' +
CAST(@Contador AS VARCHAR);

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

```
SET @Fin = GETDATE();
PRINT '--- FIN PRUEBA MANUAL ---';
SELECT DATEDIFF(ms, @Inicio, @Fin) AS [Tiempo Total (ms) - Conteo
Directo Manual];

-- 2 PRUEBA DE FUNCIÓN ESCALAR (cantMaquinasReparadasGrupo)

PRINT '--- INICIANDO PRUEBA DE FUNCIÓN ESCALAR
(cantMaquinasReparadasGrupo) ---';
SET @Inicio = GETDATE();
SET @Contador = 1;

WHILE @Contador <= @Iteraciones
BEGIN
    -- Rota el ID de Grupo entre 1 y 5
    SET @GrupoID = (@Contador - 1) % 5 + 1;

    -- Ejecución con la Función Escalar
    SET @Cantidad = dbo.cantMaquinasReparadasGrupo(@GrupoID);

    -- Muestra el resultado de la iteración con el número de iteración
    PRINT 'Función - Grupo ID: ' + CAST(@GrupoID AS VARCHAR) + ' -
Cantidad: ' + CAST(@Cantidad AS VARCHAR) + ' | Iteracion Nro: ' +
CAST(@Contador AS VARCHAR);

    SET @Contador = @Contador + 1;
END

SET @Fin = GETDATE();
PRINT '--- FIN PRUEBA FUNCIÓN ESCALAR ---';
SELECT DATEDIFF(ms, @Inicio, @Fin) AS [Tiempo Total (ms) - Función
Escalar];
```

Results | Messages | Execution plan

	Tiempo Total (ms) - Conteo Directo Manual
1	16123

	Tiempo Total (ms) - Función Escalar
1	370

Desarrollo del tema 3: Optimización de consultas a través de índices

Para evitar inconsistencias debido a relaciones con otras tablas, se creó una copia completa de la tabla original 'Reparacion' bajo el nombre 'Reparacion_Test'. Luego se vació para iniciar el proceso con un entorno limpio.

```
-- Copiamos la tabla reparacion
SELECT *
INTO Reparacion_Test
FROM Reparacion;
GO

-- Limpiamos los datos de la tabla de prueba
TRUNCATE TABLE Reparacion_Test;
GO

-- Verificamos que este vacia
SELECT COUNT(*)
FROM Reparacion_Test;
GO
```

Messages | Execution plan

```
(5 rows affected)

(1 row affected)

Completion time: 2025-11-16T17:37:35.2722178-03:00
```

Results | Messages

	(No column name)
1	0

Se realizó la carga de 1.000.000 de registros utilizando un script automatizado en lotes de 50.000 registros. Las fechas fueron generadas de manera cíclica dentro de un rango de 600 días.

```sql
-- creamos el script para generar la carga masiva

DECLARE @RevisionID INT = 1;
DECLARE @GrupoID INT = 1;
DECLARE @i INT = 1;
DECLARE @batchSize INT = 50000;
DECLARE @max INT = 1000000;

WHILE @i <= @max
BEGIN
    WITH N AS (
        SELECT TOP (@batchSize)
            ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) + (@i - 1) AS num
        FROM sys.all_objects a
        CROSS JOIN sys.all_objects b
    )
    INSERT INTO Reparacion_Test (fecha_inicio, fecha_fin, id_revision, id_grupo)
    SELECT
        DATEADD(DAY, (num - 1) % 600, '2024-01-01'),
        DATEADD(DAY, ((num - 1) % 600) + 1, '2024-01-01'),
        @RevisionID,
        @GrupoID
    FROM N;

    SET @i += @batchSize;
END
GO
```

```
(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

(50000 rows affected)

Completion time: 2025-11-16T16:11:44.5360522-03:00
```

Verificamos que se carguen todos los registros

```
-- Verificar cantidad de carga
SELECT COUNT(*) AS Total_Registros
FROM Reparacion_Test;
GO
```

	Total_Registros
1	1000000

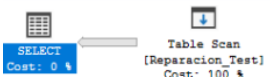
Antes de ejecutar la consulta, se limpiaron buffers y planes almacenados mediante los comandos: CHECKPOINT, DBCC DROPCLEANBUFFERS y DBCC FREEPROCCACHE. Se activaron las estadísticas de IO y TIME para capturar el rendimiento real. La consulta buscó registros dentro de un rango de fechas.

```
CHECKPOINT; -- Guarda las páginas modificadas en disco para dejar la base de datos en estado consistente.
DBCC DROPCLEANBUFFERS; -- Quita del buffer pool las páginas limpias para obligar a SQL Server a leer desde disco en la próxima consulta.
DBCC FREEPROCCACHE; -- Elimina los planes de ejecución almacenados para que SQL Server genere uno nuevo.
GO

SET STATISTICS TIME ON; -- Activa la medición del tiempo que tarda en ejecutarse cada consulta
SET STATISTICS IO ON; -- Activa la medición de estadísticas de entrada/salida

-- La consulta devuelve todas las reparaciones que esten en ese rango de fecha
SELECT *
FROM Reparacion_Test
WHERE fecha_inicio BETWEEN '2024-01-01' AND '2024-05-31';

SET STATISTICS TIME OFF;
SET STATISTICS IO OFF;
GO
```

Resultados obtenidos

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 250 ms, elapsed time = 330 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

(253384 rows affected)
Table 'Reparacion_Test'. Scan count 1, logical reads 4330, physical reads 0, page server reads 0, read-ahead reads 4260, page server read-ahead reads 0,

(1 row affected)

 SQL Server Execution Times:
   CPU time = 281 ms,  elapsed time = 3024 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

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

```
Query 1: Query cost (relative to the batch): 100%
-- La consulta devuelve todas las reparaciones que esten en ese rango de fecha SELECT * FROM Reparacion_Test WHERE fecha_inicio BETWEEN
```



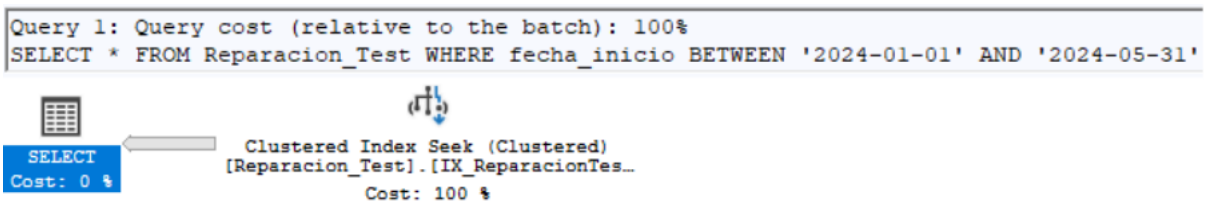
Se creó un índice clustered sobre la columna fecha_inicio. Luego se repitió la consulta para comparar el rendimiento respecto del escenario sin índices.

```
-- se crea indice clustered por fecha_inicio
CREATE CLUSTERED INDEX IX_ReparacionTest_Fecha
ON Reparacion_Test (fecha_inicio);
GO

CHECKPOINT;
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
GO

SET STATISTICS TIME ON;
SET STATISTICS IO ON;

SELECT *
FROM Reparacion_Test
WHERE fecha_inicio BETWEEN '2024-01-01' AND '2024-05-31';

SET STATISTICS TIME OFF;
SET STATISTICS IO OFF;
GO
```

Resultados Obtenidos

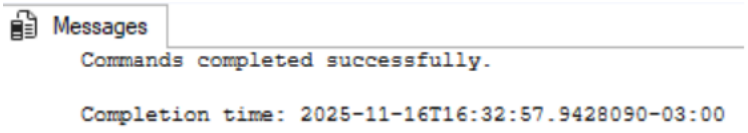
Messages | Execution plan

```
Commands completed successfully.

Completion time: 2025-11-16T16:26:53.4015325-03:00


SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 3 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

(253384 rows affected)
Table 'Reparacion_Test'. Scan count 1, logical reads 1102, physical reads 3, page server reads 0, read-ahead reads 1099, page server read-ahead reads 0,

(1 row affected)

 SQL Server Execution Times:
   CPU time = 203 ms,  elapsed time = 3380 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

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



Posteriormente se eliminó el índice clustered

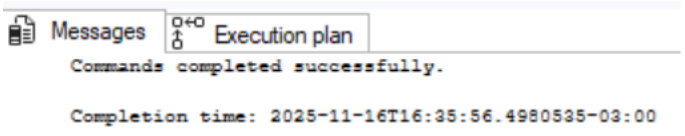
```
-- Eliminamos el indice clustered anterior
DROP INDEX IX_ReparacionTest_Fecha ON Reparacion_Test;
GO
```



Se creó un índice nonclustered sobre fecha_inicio, incluyendo id_revision, id_grupo y fecha_fin para cubrir completamente la consulta y se repitió nuevamente la operación de búsqueda registrando los tiempos y estadísticas.

```
-- se crea un indice nonclustered por fecha_inicio con columnas incluidas
CREATE NONCLUSTERED INDEX IX_ReparacionTest_FechaInicio_NC
ON Reparacion_Test(fecha_inicio)
INCLUDE (id_revision, id_grupo, fecha_fin);
GO

CHECKPOINT;
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
GO

SET STATISTICS TIME ON;
SET STATISTICS IO ON;

SELECT *
FROM Reparacion_Test
WHERE fecha_inicio BETWEEN '2024-01-01' AND '2024-05-31';

SET STATISTICS TIME OFF;
SET STATISTICS IO OFF;
GO
```

Resultados obtenidos



```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 5 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

(253384 rows affected)
Table 'Reparacion_Test'. Scan count 1, logical reads 3345, physical reads 0, page server reads 0, read-ahead reads 0,

(1 row affected)

 SQL Server Execution Times:
   CPU time = 250 ms,  elapsed time = 2930 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

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

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM Reparacion_Test WHERE fecha_inicio BETWEEN '2024-01-01' AND '2024-05-31'
Missing Index (Impact 79.6524): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Reparacion_Test]
```

Finalmente se eliminaron los índices utilizados en las pruebas y se borró la tabla de prueba Reparacion_Test.

```sql
-- Eliminamos el indice nonclustered por fecha_inicio con columnas incluidas
DROP INDEX IX_ReparacionTest_FechaInicio_NC ON Reparacion_Test;
DROP TABLE Reparacion_Test; -- eliminamos la tabla Reparacion_Test
GO
```

Messages

```
Commands completed successfully.

Completion time: 2025-11-16T16:46:36.0227355-03:00
```

Desarrollo del tema 4: Manipulación de datos JSON

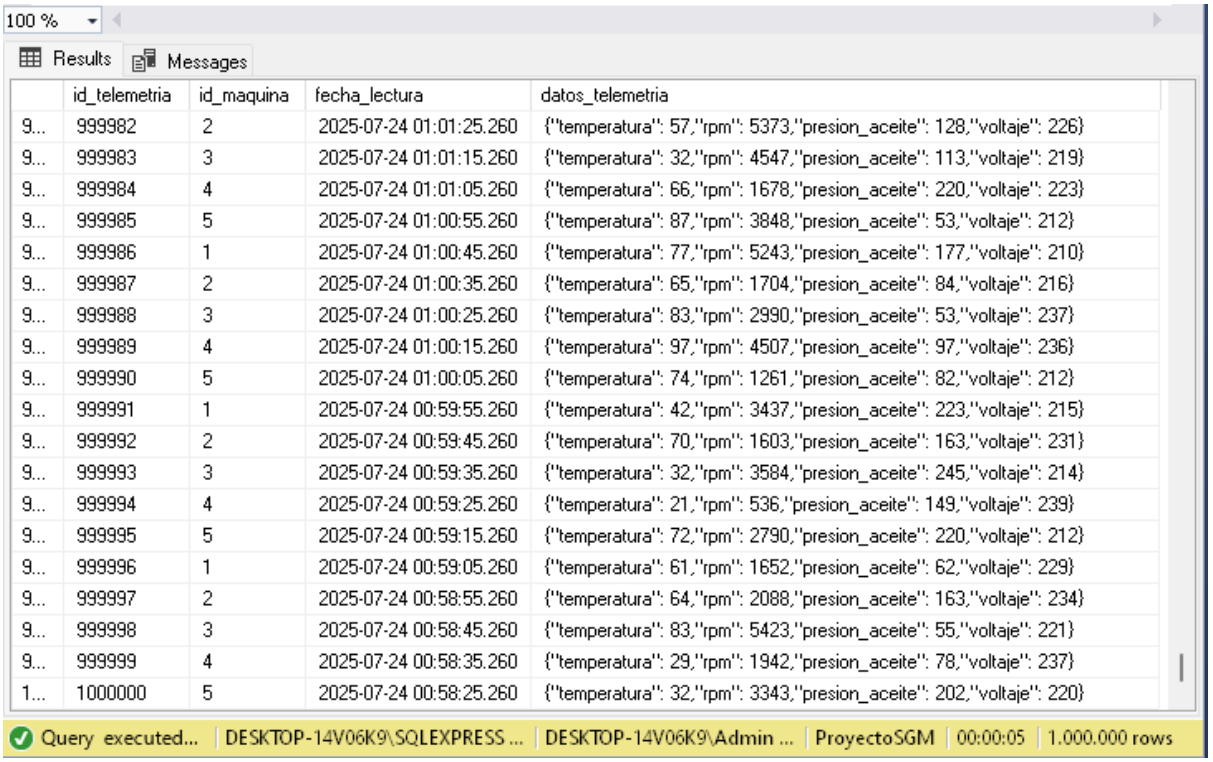
Se creo una tabla TelemetriaMaquina que guarda datos telemétricos de una maquina determinada, los

```sql
CREATE TABLE TelemetriaMaquina (
    id_telemetria INT IDENTITY(1,1) PRIMARY KEY,
    id_maquina INT NOT NULL,
    fecha_lectura DATETIME NOT NULL DEFAULT GETDATE(),
    datos_telemetria NVARCHAR(MAX) NOT NULL, -- Columna JSON

    CONSTRAINT FK_Telemetria_Maquina
        FOREIGN KEY (id_maquina) REFERENCES Maquina(id_maquina)
```

```
);
```

Se procedió a insertar un lote de datos aleatorios de 1000000 registros, con id_maquina del 1 al 5, mediante el siguiente script:

```
/* Lote de inserción de datos de telemetría en formato JSON para las
máquinas con maquina_id del 1 al 5 cada 10 segundos*/
DECLARE
    @i INT = 1,
    @id_maquina INT,
    @temp INT,
    @rpm INT,
    @presAceite INT,
    @volt INT;

WHILE @i <= 1000000
BEGIN
    -- Datos aleatorios
    SET @id_maquina = ((@i - 1) % 5) + 1;              -- máquinas 1 a 5
    SET @temp = (ABS(CHECKSUM(NEWID())) % 80) + 20;       -- 20 a 100 °C
    SET @rpm  = (ABS(CHECKSUM(NEWID())) % 5000) + 500;      -- 500 a
5500 RPM
    SET @presAceite = (ABS(CHECKSUM(NEWID())) % 200) + 50; --50 a 250
PSI
    SET @volt = (ABS(CHECKSUM(NEWID())) % 30) + 210;      -- 210 a 240 V

    --Insertar los datos
    INSERT INTO TelemetriaMaquina (id_maquina, fecha_lectura,
datos_telemetria)
    VALUES
    (
        @id_maquina,
        DATEADD(SECOND, -10 * @i, GETDATE()),   -- cada lectura = 10
segundos atrás
        CONCAT(
            '{',
                '"temperatura": ', @temp, ',',
                '"rpm": ', @rpm, ',',
                '"presion_aceite": ', @presAceite, ',',
                '"voltaje": ', @volt,
            '}'
        )
    );

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

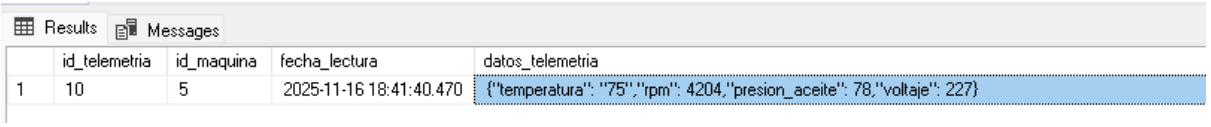


	id_telemetria	id_maquina	fecha_lectura	datos_telemetria
9...	999982	2	2025-07-24 01:01:25.260	{"temperatura": 57,"rpm": 5373,"presion_aceite": 128,"voltaje": 226}
9...	999983	3	2025-07-24 01:01:15.260	{"temperatura": 32,"rpm": 4547,"presion_aceite": 113,"voltaje": 219}
9...	999984	4	2025-07-24 01:01:05.260	{"temperatura": 66,"rpm": 1678,"presion_aceite": 220,"voltaje": 223}
9...	999985	5	2025-07-24 01:00:55.260	{"temperatura": 87,"rpm": 3848,"presion_aceite": 53,"voltaje": 212}
9...	999986	1	2025-07-24 01:00:45.260	{"temperatura": 77,"rpm": 5243,"presion_aceite": 177,"voltaje": 210}
9...	999987	2	2025-07-24 01:00:35.260	{"temperatura": 65,"rpm": 1704,"presion_aceite": 84,"voltaje": 216}
9...	999988	3	2025-07-24 01:00:25.260	{"temperatura": 83,"rpm": 2990,"presion_aceite": 53,"voltaje": 237}
9...	999989	4	2025-07-24 01:00:15.260	{"temperatura": 97,"rpm": 4507,"presion_aceite": 97,"voltaje": 236}
9...	999990	5	2025-07-24 01:00:05.260	{"temperatura": 74,"rpm": 1261,"presion_aceite": 82,"voltaje": 212}
9...	999991	1	2025-07-24 00:59:55.260	{"temperatura": 42,"rpm": 3437,"presion_aceite": 223,"voltaje": 215}
9...	999992	2	2025-07-24 00:59:45.260	{"temperatura": 70,"rpm": 1603,"presion_aceite": 163,"voltaje": 231}
9...	999993	3	2025-07-24 00:59:35.260	{"temperatura": 32,"rpm": 3584,"presion_aceite": 245,"voltaje": 214}
9...	999994	4	2025-07-24 00:59:25.260	{"temperatura": 21,"rpm": 536,"presion_aceite": 149,"voltaje": 239}
9...	999995	5	2025-07-24 00:59:15.260	{"temperatura": 72,"rpm": 2790,"presion_aceite": 220,"voltaje": 212}
9...	999996	1	2025-07-24 00:59:05.260	{"temperatura": 61,"rpm": 1652,"presion_aceite": 62,"voltaje": 229}
9...	999997	2	2025-07-24 00:58:55.260	{"temperatura": 64,"rpm": 2088,"presion_aceite": 163,"voltaje": 234}
9...	999998	3	2025-07-24 00:58:45.260	{"temperatura": 83,"rpm": 5423,"presion_aceite": 55,"voltaje": 221}
9...	999999	4	2025-07-24 00:58:35.260	{"temperatura": 29,"rpm": 1942,"presion_aceite": 78,"voltaje": 237}
1...	1000000	5	2025-07-24 00:58:25.260	{"temperatura": 32,"rpm": 3343,"presion_aceite": 202,"voltaje": 220}

Suponiendo que el id_telemetria tuvo una lectura de temperatura errónea, se procedió a modificarlo haciendo uso del procedimiento JSON_MODIFY

```
UPDATE TelemetriaMaquina
SET datos_telemetria = JSON_MODIFY(datos_telemetria, '$.temperatura',
'75')
WHERE id_telemetria = 10;
```

El campo JSON del registro correspondiente se actualiza correctamente.

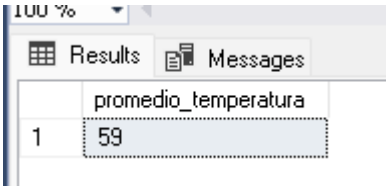


	id_telemetria	id_maquina	fecha_lectura	datos_telemetria
1	10	5	2025-11-16 18:41:40.470	{"temperatura": "75","rpm": 4204,"presion_aceite": 78,"voltaje": 227}

Se utilizó la operación de agregación AVG() para obtener la temperatura promedio de todas las máquinas:

```
SELECT AVG(CAST(JSON_VALUE(datos_telemetria, '$.temperatura') AS INT))
AS promedio_temperatura
FROM TelemetriaMaquina;
```

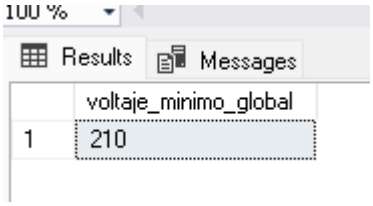
Resultado obtenido:



	promedio_temperatura
1	59

Luego se utilizó la instrucción MIN() sobre la clave “voltaje” de todas las máquinas, para asi obtener el menor voltaje registrado y evaluar caídas de tensión:

```
SELECT
    MIN(TRY_CAST(JSON_VALUE(datos_telemetria, '$.voltaje') AS FLOAT)) AS
voltaje_minimo_global
FROM TelemetriaMaquina;
```



100 %

Results | Messages

	voltaje_minimo_global
1	210

Supongamos que los registros de temperatura de la máquina con id_maquina 1 durante la última semana están erróneos porque el sensor de temperatura de dicha máquina esté defectuoso, se puede eliminar la clave temperatura para estos registros mediante el uso de JSON_MODIFY

```
UPDATE TelemetriaMaquina
SET datos_telemetria = JSON_MODIFY(datos_telemetria, '$.temperatura',
NULL)
WHERE id_maquina = 1
  AND fecha_lectura >= DATEADD(WEEK, -1, GETDATE());
```

La clave de temperatura se eliminó completamente para estos registros.

Results | Messages

	id_telemetria	id_maquina	fecha_lectura	datos_telemetria
1	1	1	2025-11-16 18:43:10.470	{"rpm": 4883,"presion_aceite": 187,"voltaje": 221}
2	6	1	2025-11-16 18:42:20.470	{"rpm": 5198,"presion_aceite": 100,"voltaje": 224}
3	11	1	2025-11-16 18:41:30.473	{"rpm": 1908,"presion_aceite": 186,"voltaje": 236}
4	16	1	2025-11-16 18:40:40.473	{"rpm": 1307,"presion_aceite": 198,"voltaje": 228}
5	21	1	2025-11-16 18:39:50.473	{"rpm": 5138,"presion_aceite": 100,"voltaje": 217}
6	26	1	2025-11-16 18:39:00.473	{"rpm": 2438,"presion_aceite": 103,"voltaje": 223}
7	31	1	2025-11-16 18:38:10.473	{"rpm": 878,"presion_aceite": 109,"voltaje": 239}
8	36	1	2025-11-16 18:37:20.477	{"rpm": 516,"presion_aceite": 70,"voltaje": 232}
9	41	1	2025-11-16 18:36:30.477	{"rpm": 5030,"presion_aceite": 181,"voltaje": 228}
10	46	1	2025-11-16 18:35:40.477	{"rpm": 1733,"presion_aceite": 102,"voltaje": 233}
11	51	1	2025-11-16 18:34:50.477	{"rpm": 2698,"presion_aceite": 160,"voltaje": 223}
12	56	1	2025-11-16 18:34:00.480	{"rpm": 5267,"presion_aceite": 186,"voltaje": 224}
13	61	1	2025-11-16 18:33:10.480	{"rpm": 4575,"presion_aceite": 186,"voltaje": 215}
14	66	1	2025-11-16 18:32:20.480	{"rpm": 2457,"presion_aceite": 224,"voltaje": 236}
15	71	1	2025-11-16 18:31:30.480	{"rpm": 3792,"presion_aceite": 63,"voltaje": 235}
16	76	1	2025-11-16 18:30:40.480	{"rpm": 4089,"presion_aceite": 107,"voltaje": 225}
17	81	1	2025-11-16 18:29:50.480	{"rpm": 3296,"presion_aceite": 136,"voltaje": 239}

La siguiente consulta muestra el promedio de las lecturas de cada parámetro para todas lás máquinas

```
SELECT
    id_maquina,
```

```
    ROUND(AVG(TRY_CAST(JSON_VALUE(datos_telemetria, '$.voltaje') AS
FLOAT)), 2) AS promedio_voltaje,
    ROUND(AVG(TRY_CAST(JSON_VALUE(datos_telemetria, '$.temperatura') AS
FLOAT)), 2) AS promedio_temperatura,
    ROUND(AVG(TRY_CAST(JSON_VALUE(datos_telemetria, '$.rpm') AS FLOAT)),
2) AS promedio_rpm,
    ROUND(AVG(TRY_CAST(JSON_VALUE(datos_telemetria, '$.presion_aceite')
AS FLOAT)), 2) AS promedio_presion_aceite
FROM TelemetriaMaquina
GROUP BY id_maquina;
```

Resultados de la consulta:

Results | Messages

	id_maquina	promedio_voltaje	promedio_temperatura	promedio_rpm	promedio_presion_aceite
1	3	224,52	59,49	3003,92	149,63
2	1	224,48	59,47	2997,81	149,45
3	4	224,52	59,53	2995,16	149,62
4	5	224,52	59,48	2997,9	149,43
5	2	224,51	59,42	2999,85	149,46

```
Tiempos de ejecución de SQL Server:
  Tiempo de CPU = 8984 ms, tiempo transcurrido = 10102 ms.
```

Como forma de optimizar esta consulta, se propone el uso de OPEN_JSON, esta operación toma el objeto JSON y lo transforma en una tabla temporal, con la que se puede operar mediante una cláusula JOIN

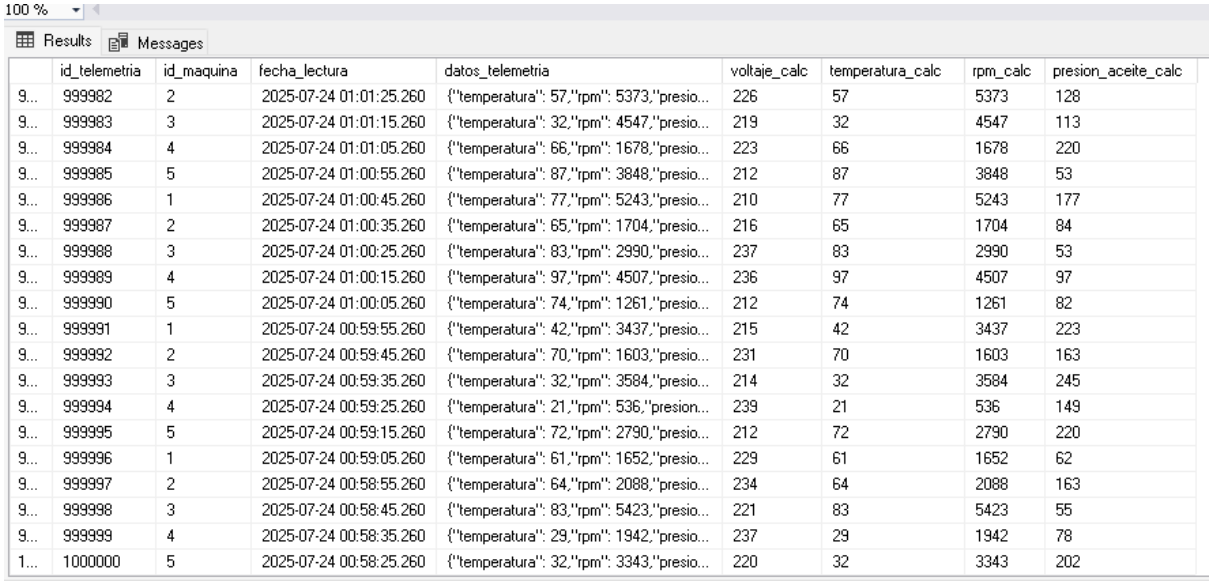
```
SELECT
    t.id_maquina,
    ROUND(AVG(TRY_CAST(j.voltaje AS FLOAT)), 2) AS promedio_voltaje,
    ROUND(AVG(TRY_CAST(j.temperatura AS FLOAT)), 2) AS
promedio_temperatura,
    ROUND(AVG(TRY_CAST(j.rpm AS FLOAT)), 2) AS promedio_rpm,
    ROUND(AVG(TRY_CAST(j.presion_aceite AS FLOAT)), 2) AS
promedio_presion_aceite
FROM TelemetriaMaquina AS t
CROSS APPLY OPENJSON(t.datos_telemetria)
WITH (
        voltaje          FLOAT '$.voltaje',
        temperatura      FLOAT '$.temperatura',
        rpm              FLOAT '$.rpm',
        presion_aceite   FLOAT '$.presion_aceite'
     ) AS j
GROUP BY t.id_maquina;
```

Otra forma en la que se propone optimizar las consultas en un campo JSON es el uso de columnas persistentes. Supongamos que los datos de lectura a registrar vendrán

siempre en forma de JSON, se puede crear columnas persistentes que toman el valor de los campos del JSON una vez mediante JSON_VALUE y lo guardan de forma persistente para que las consultas posteriores puedan tomar el valor de estas columnas directamente.

```
ALTER TABLE TelemetriaMaquina
ADD voltaje_calc AS CAST(JSON_VALUE(datos_telemetria, '$.voltaje') AS
FLOAT) PERSISTED;

ALTER TABLE TelemetriaMaquina
ADD temperatura_calc AS CAST(JSON_VALUE(datos_telemetria,
'$.temperatura') AS FLOAT) PERSISTED;

ALTER TABLE TelemetriaMaquina
ADD rpm_calc AS CAST(JSON_VALUE(datos_telemetria, '$.rpm') AS FLOAT)
PERSISTED;

ALTER TABLE TelemetriaMaquina
ADD presion_aceite_calc AS CAST(JSON_VALUE(datos_telemetria,
'$.presion_aceite') AS FLOAT) PERSISTED;
```

Las nuevas columnas tomaron los valores del campo JSON correctamente:



100 %

Results | Messages

	id_telemetria	id_maquina	fecha_lectura	datos_telemetria	voltaje_calc	temperatura_calc	rpm_calc	presion_aceite_calc
9...	999982	2	2025-07-24 01:01:25.260	{"temperatura": 57,"rpm": 5373,"presio...	226	57	5373	128
9...	999983	3	2025-07-24 01:01:15.260	{"temperatura": 32,"rpm": 4547,"presio...	219	32	4547	113
9...	999984	4	2025-07-24 01:01:05.260	{"temperatura": 66,"rpm": 1678,"presio...	223	66	1678	220
9...	999985	5	2025-07-24 01:00:55.260	{"temperatura": 87,"rpm": 3848,"presio...	212	87	3848	53
9...	999986	1	2025-07-24 01:00:45.260	{"temperatura": 77,"rpm": 5243,"presio...	210	77	5243	177
9...	999987	2	2025-07-24 01:00:35.260	{"temperatura": 65,"rpm": 1704,"presio...	216	65	1704	84
9...	999988	3	2025-07-24 01:00:25.260	{"temperatura": 83,"rpm": 2990,"presio...	237	83	2990	53
9...	999989	4	2025-07-24 01:00:15.260	{"temperatura": 97,"rpm": 4507,"presio...	236	97	4507	97
9...	999990	5	2025-07-24 01:00:05.260	{"temperatura": 74,"rpm": 1261,"presio...	212	74	1261	82
9...	999991	1	2025-07-24 00:59:55.260	{"temperatura": 42,"rpm": 3437,"presio...	215	42	3437	223
9...	999992	2	2025-07-24 00:59:45.260	{"temperatura": 70,"rpm": 1603,"presio...	231	70	1603	163
9...	999993	3	2025-07-24 00:59:35.260	{"temperatura": 32,"rpm": 3584,"presio...	214	32	3584	245
9...	999994	4	2025-07-24 00:59:25.260	{"temperatura": 21,"rpm": 536,"presion...	239	21	536	149
9...	999995	5	2025-07-24 00:59:15.260	{"temperatura": 72,"rpm": 2790,"presio...	212	72	2790	220
9...	999996	1	2025-07-24 00:59:05.260	{"temperatura": 61,"rpm": 1652,"presio...	229	61	1652	62
9...	999997	2	2025-07-24 00:58:55.260	{"temperatura": 64,"rpm": 2088,"presio...	234	64	2088	163
9...	999998	3	2025-07-24 00:58:45.260	{"temperatura": 83,"rpm": 5423,"presio...	221	83	5423	55
9...	999999	4	2025-07-24 00:58:35.260	{"temperatura": 29,"rpm": 1942,"presio...	237	29	1942	78
1...	1000000	5	2025-07-24 00:58:25.260	{"temperatura": 32,"rpm": 3343,"presio...	220	32	3343	202

Luego se pueden crear índices para estas últimas columnas creadas:

```
CREATE INDEX IX_Telemetria_Voltaje ON TelemetriaMaquina(voltaje_calc);
CREATE INDEX IX_Telemetria_Temperatura ON
TelemetriaMaquina(temperatura_calc);
CREATE INDEX IX_Telemetria_RPM ON TelemetriaMaquina(rpm_calc);
CREATE INDEX IX_Telemetria_Presion ON
TelemetriaMaquina(presion_aceite_calc);
```

Resultados:

Results Messages

	id_maquina	promedio_voltaje	promedio_temperatura	promedio_rpm	promedio_presion_aceite
1	3	224,52	59,49	3003,92	149,63
2	1	224,48	59,47	2997,81	149,45
3	4	224,52	59,53	2995,16	149,62
4	5	224,52	59,48	2997,9	149,43
5	2	224,51	59,42	2999,85	149,46

```
(5 rows affected)

 Tiempos de ejecución de SQL Server:
   Tiempo de CPU = 218 ms, tiempo transcurrido = 302 ms.
```

CAPÍTULO V: CONCLUSIONES

Transacciones: Las transacciones resultan muy útiles como prevención de errores en la base de datos. De la evaluación empírica de este proyecto se rescata que dada la naturaleza de que toda operación SQL es en sí una transacción, el fallo de cualquiera de estas provoca que la transacción que las contiene se revierta, como si fuese una transacción interna llamando a ROLLBACK. En la anidación de transacciones se observa que si una transacción interna realiza COMMIT, esta no se guardará hasta que la transacción más exterior haya hecho COMMIT también. Sin embargo, basta que cualquier transacción interna llame a ROLLBACK para que ningún cambio se guarde. El uso de SAVEPOINTS en SQL Sever y otros SGBD da más control sobre este comportamiento.

Procedimientos y funciones anidadas: al insertar un lote de datos con sentencias INSERT directamente y al hacerlo llamando a un procedimiento AgregarReparacionConRepuesto que hace lo mismo, se observó un tiempo de ejecución mucho mayor en el procedimiento. Esto puede explicarse en que las operaciones INSERT no son lo suficientemente complejas como para que el compilar el plan de ejecución una sola vez compense el costo de llamar al procedimiento (como validar los parámetros) varias veces. Sin embargo la prueba similar realizada sobre la función cantMaquinasReparadasGrupo demuestra un tiempo muchísimo menor al llamarse a la función, esto es porque la consulta es relativamente más compleja y el tiempo de compilar el plan de ejecución varias veces al hacerlo de forma manual es notoria, cosa que no pasa en la función cuyo plan de ejecución se compila una sola vez al crearse. Se debe mencionar también la mayor facilidad de reutilizar el código mediante el uso de los procedimientos y funciones.

Optimización a través de índices: En la consulta ejecutada sin índices se demostró un rendimiento limitado, ya que el motor debió realizar un Table Scan, generando un alto número de lecturas y un tiempo de respuesta considerable. Esto confirmó que, en tablas extensas, la ausencia de índices perjudica directamente la eficiencia de búsqueda.

La creación de un índice agrupado (clustered) sobre la columna fecha_inicio produjo una mejora notable. Las lecturas disminuyeron significativamente y el plan de ejecución se optimizó, mostrando que el ordenamiento físico de los datos favorece especialmente las consultas por rango.

Por otra parte, el índice non clustered con columnas incluidas no logró superar el rendimiento del índice agrupado y, en las pruebas realizadas, incluso presentó un mayor número de lecturas. Esto se debió a que la consulta devolvía un conjunto amplio de registros, lo que generó numerosas operaciones de búsqueda adicionales (lookups), afectando negativamente el desempeño.

Este análisis permitió no solo validar el impacto real de los índices en el rendimiento, sino también comprender la importancia de seleccionar adecuadamente la estructura de acceso a los datos según las necesidades de un sistema. Las pruebas reflejan de manera clara que la optimización de consultas no depende únicamente del hardware o del tamaño de la base de datos, sino principalmente de un diseño eficiente de índices.

Manipulación de datos JSON: La utilización de datos JSON en la base de datos permite una mayor flexibilidad al poder recibir y almacenar varios datos distintos en una única operación y sin especificación de tipado. Sin embargo resulta costoso en términos de rendimiento al tener que llamar a procedimientos determinados para poder crear, leer, modificar y borrar esos datos.

BIBLIOGRAFÍA.

SQLServerCentral. (s. f.). Explicit transactions. SQLServerCentral. https://www.sqlservercentral.com/articles/explicit-transactions

Torres, M. (s. f.). Bases de datos: Tema 6. Universidad de Alicante. https://w3.ual.es/~mtorres/BD/bdt6.pdf

Guillermo Cherencio. (s. f.). Apuntes de bases de datos: Tema 6. https://www.grch.com.ar/docs/bd/apuntes/BDTema06.pdf

SQLearning. (s. f.). Transacciones. https://sqlearning.com/es/elementos-lenguaje/transacciones/

Microsoft. (s. f.). Índices clúster y no clúster. Microsoft Learn. https://learn.microsoft.com/es-es/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver17

Microsoft. (s. f.). Indexes. Microsoft Learn. https://learn.microsoft.com/es-es/sql/relational-databases/indexes/indexes?view=sql-server-ver17

Informática Sierra. (s. f.). Optimización de consultas con índices: Guía práctica para programadores. https://informaticasierra.es/optimizacion-de-consultas-con-indices-guia-practica-para-programadores/

Universidad Don Bosco. (s. f.). Guía 10: Base de datos I. https://www.udb.edu.sv/udb_files/recursos_guias/informatica-ingenieria/base-de-datos-i/2019/i/guia-10.pdf

ScholarHat. (s. f.). Difference between stored procedure and function in SQL Server. https://www.scholarhat.com/tutorial/sqlserver/difference-between-stored-procedure-and-function-in-sql-server

MSSQLTips. (s. f.). SQL stored procedures, views, functions examples. https://www.mssqltips.com/sqlservertip/7437/sql-stored-procedures-views-functions-examples/