Curso de SQL Avanzado - Sesión 4

Scotiabank | **Belatrix**

Instructor: Alan Badillo Salas

Contenido

```
Módulo 4: Programación en SQL

1. Uso de funciones y procedimientos almacenados
2. Desarrollo de disparadores avanzados
3. Trabajo con cursores y variables
```

Temas

401. Introducción a SQL Server
402. Repaso de Consultas, Índices y Particionamiento en SQL Server
403. Uso de funciones y procedimientos almacenados
404. Desarrollo de disparadores avanzados
405. Trabajo con cursores y variables

401. Introducción a SQL Server

SQL Server es un sistema de gestión de bases de datos relacional desarrollado por Microsoft. Esta plataforma es ampliamente utilizada en empresas de todos los tamaños para almacenar, recuperar y gestionar datos. Aquí están algunos aspectos fundamentales que cubre la administración de SQL Server:

1. Instalación y configuración

La administración comienza con la instalación de SQL Server. Es crucial seleccionar la edición adecuada que se ajuste a las necesidades del negocio (por ejemplo, Express, Standard, Enterprise). Después de la instalación, la configuración del servidor es fundamental para optimizar el rendimiento, la seguridad y la accesibilidad. Esto incluye configurar instancias, bases de datos, y parámetros de red.

2. Gestión de bases de datos

El administrador de SQL Server debe saber cómo crear y configurar bases de datos. Esto incluye definir el tamaño inicial, el crecimiento incremental y las opciones de seguridad. La gestión de bases de datos también implica el diseño y la normalización de esquemas, la creación de tablas, índices, procedimientos almacenados y vistas.

3. Seguridad

La seguridad es crucial en la administración de SQL Server. Esto abarca la configuración de autenticación (Windows o SQL Server), asignación de roles y permisos a usuarios y grupos, y asegurar los datos mediante cifrado y auditorías. Se debe prestar atención especial a proteger los datos contra accesos no autorizados y amenazas externas.

4. Respaldos y recuperación

Esencial para cualquier sistema de base de datos, los administradores deben establecer políticas de respaldo para proteger los datos. Esto incluye realizar respaldos completos, diferenciales y de registros de transacciones regularmente. Además, deben estar preparados para restaurar bases de datos de estos respaldos en caso de pérdida de datos o desastres.

5. Monitoreo y optimización del rendimiento

El monitoreo regular del rendimiento de SQL Server ayuda a identificar y resolver cuellos de botella. Esto puede involucrar la supervisión del uso de la CPU, la memoria, el espacio en disco y la actividad de la red. La optimización puede requerir ajustes en las consultas, índices, o configuraciones del servidor para mejorar la eficiencia general.

6. Automatización de tareas

SQL Server proporciona herramientas como SQL Server Agent, que permite automatizar tareas repetitivas como respaldos, mantenimiento de bases de datos y ejecución de scripts. La automatización ayuda a reducir errores humanos y a liberar tiempo del administrador para centrarse en tareas más críticas.

7. Actualizaciones y migraciones

Mantener SQL Server actualizado es importante para aprovechar las mejoras en características y seguridad. Las actualizaciones pueden requerir planificación cuidadosa y pruebas para asegurar la compatibilidad con aplicaciones existentes. Las migraciones, ya sea a nuevas versiones o a plataformas en la nube como Azure SQL, también son parte del trabajo del administrador.

8. Solución de problemas

Los administradores deben ser adeptos en diagnosticar y resolver problemas que puedan surgir, como problemas de rendimiento, fallos en el hardware, y errores de software. Conocer las herramientas y técnicas de solución de problemas es crucial.

La administración de SQL Server es un campo dinámico y esencial que requiere una combinación de conocimientos técnicos y habilidades prácticas. Dominar estas áreas fundamentales puede llevar a una gestión eficaz y eficiente de los recursos de datos de una organización.

402. Repaso de Consultas, Índices y Particionamiento en SQL Server

Debido a la sustición de Oracle por SQL Server, vamos a repasar algunos conceptos en SQL Server y su manejo desde Python.

Lo principal será entender cómo se realiza la conexión a la base de datos desde Python (con usuario previamente configurado). En el módulo siguiente veremos cómo configurar una base de datos y usuario para implementar las auditorías.

Conectar a la base de datos desde Python

Lo principal para comenzar a trabajar con SQL Server desde Python es crear una conexión usando la librería `pymssql` que se puede instalar con:

```
pip install pymssql
```

Nota: En Google Colab deberemos usar `!pip install pymssql`.

Una vez instalado apuntaremos al servidor de la base de datos especificando las credenciales y demás configuraciones que deberán ser válidas en el servidor.

```
import pymssql

# Configuraciones de la conexión
server = '<IP | HOST | DOMINIO DNS>'
database = '<nombre de la base de datos>'
username = '<usuario con acceso a la base de datos>'
password = '<Contraseña del usuario>'

# Crear la conexión y extraer el cursor capaz de ejecutar las consultas
conn = pymssql.connect(server, username, password, database)
cursor = conn.cursor()

# Consultamos la versión para verificar que funcione
cursor.execute('SELECT @@VERSION')
row = cursor.fetchone()
while row:
    print("SQL Server version:", row[0])
    row = cursor.fetchone()
```

Nota: Del lado del servidor debemos verificar que esté habilitado el modo *SQL Server authentication* y el inicio de sesión otorgado a las bases de datos para el usuario que tendrá acceso.

Cuando no utilicemos más la conexión debemos cerrarla para no acumular conexiones abiertas:

```
# Cerrar conexión
cursor.close()
conn.close()
```

1. Consultas

Las consultas en SQL Server son instrucciones escritas en Transact-SQL (T-SQL) que permiten recuperar, actualizar, insertar y eliminar datos de las bases de datos. Para escribir consultas eficientes, es fundamental entender:

- **Selección de datos**: Usar `SELECT` para especificar las columnas y `FROM` para indicar las tablas. Las cláusulas `WHERE`, `JOIN`, y `GROUP BY` permiten filtrar, combinar y agrupar datos, respectivamente.
- **Funciones agregadas**: Como `SUM()`, `AVG()`, `COUNT()`, etc., que permiten realizar cálculos sobre un conjunto de valores.
- **Subconsultas y CTEs**: Las subconsultas permiten utilizar el resultado de una consulta como parte de otra. Las Common Table Expressions (CTEs) proporcionan una manera más legible y potente de estructurar las subconsultas.

En Python usaremos el cursor para ejecutar las sentencias de consulta:

```
# Consulta SQL que quieres ejecutar
sql = "SELECT id, nombre FROM usuarios"

# Ejecutar la consulta
cursor.execute(sql)

# Iterar sobre los resultados
for row in cursor:
    print(f"ID: {row[0]}, Nombre: {row[1]}")
```

Con algunas estrategias podemos recolectar los resultados y procesarlos mediante Python.

También podemos hacer actualizaciones:

```
# Consulta SQL para inserción
sql_insert = "INSERT INTO usuarios (id, nombre) VALUES (%d, %s)"

# Datos que quieres insertar
id = 2
name = 'Ana Ming'

# Ejecutar la consulta de inserción
cursor.execute(sql_insert, (id, name))
conn.commit() # <-- Necesario para que quede insertado correctamente
```

Y actualizaciones:

```
# Información para actualizar
user_id = 1
```

```
new_email = 'nuevo_email@example.com'

# Consulta SQL para actualizar
sql = "UPDATE usuarios SET email = %s WHERE id = %s"

# Ejecutar la consulta de actualización
cursor.execute(sql, (new_email, user_id))

# Asegurar que los cambios se reflejen en la base de datos
conn.commit()
```

Subconsultas en SQL Server

En SQL Server, las subconsultas son consultas anidadas dentro de otra consulta principal. Se utilizan para obtener datos que luego se utilizan en la consulta externa. Las subconsultas pueden aparecer en diversas partes de la consulta principal, incluyendo la cláusula `SELECT`, `FROM`, y `WHERE`.

Tipos de Subconsultas

1. **Subconsulta Escalar**: Retorna un único valor y se utiliza en la cláusula `SELECT` o `WHERE`.
2. **Subconsulta de Tabla**: Retorna una tabla completa y se utiliza como una fuente de datos en la cláusula `FROM`.
3. **Subconsulta Correlativa**: Referencia a una columna de la consulta externa dentro de la subconsulta, se evalúa fila por fila.

Ejemplo de Subconsulta

Considera una base de datos con dos tablas, `Empleados` y `Departamentos`, como en el ejemplo anterior:

- **Empleados**: `EmpleadoID`, `Nombre`, `DepartamentoID`, `Salario`
- **Departamentos**: `DepartamentoID`, `NombreDepartamento`

Supongamos que quieres encontrar los nombres de los empleados que ganan más que el promedio de salarios de su departamento.

Consulta con Subconsulta

```
SELECT e.Nombre, e.Salario
FROM Empleados e
WHERE e.Salario > (
    SELECT AVG(Salario)
    FROM Empleados
    WHERE DepartamentoID = e.DepartamentoID
)
```

En este ejemplo, la subconsulta calcula el salario promedio de los empleados dentro de cada departamento, y la consulta principal selecciona los empleados que ganan más que ese promedio. La subconsulta es correlativa porque referencia `DepartamentoID` de la consulta externa.

Uso de `WITH` (Common Table Expressions, CTE)

SQL Server también soporta el uso de Common Table Expressions (CTEs), que son una forma de crear una vista temporal que está disponible solo durante la ejecución de la consulta. Las CTEs son útiles para simplificar consultas complejas, especialmente cuando se requiere reutilizar los resultados de una subconsulta múltiples veces.

Sintaxis Básica de CTE

```
WITH CTE_Nombre AS (
    SELECT columna1, columna2, ...
    FROM tabla
    WHERE condicion
)
SELECT columna1, columna2, ...
FROM CTE_Nombre
WHERE condicion_adicional;
```

Ejemplo de Uso de CTE

Supongamos que quieres obtener los nombres de todos los empleados junto con el nombre de su departamento.

```
WITH DeptoCTE AS (
    SELECT DepartamentoID, NombreDepartamento
    FROM Departamentos
)
SELECT e.Nombre, d.NombreDepartamento
FROM Empleados e
JOIN DeptoCTE d ON e.DepartamentoID = d.DepartamentoID
```

En este caso, la CTE `DeptoCTE` simplifica la consulta principal al manejar por separado la obtención de los nombres de los departamentos. Luego, se realiza un `JOIN` entre `Empleados` y la CTE para obtener el resultado deseado.

2. Índices

Los índices en SQL Server ayudan a acelerar la recuperación de datos sin tener que buscar en toda la base de datos. Son particularmente útiles en tablas grandes y se utilizan para mejorar el rendimiento de las consultas. Los tipos principales de índices son:

- **Índices Clusterizados**: Reorganizan físicamente los registros de la tabla para que coincidan con el orden del índice. Una tabla solo puede tener un índice clusterizado, ya que define el orden de almacenamiento físico de los datos.
- **Índices No Clusterizados**: Mantienen un orden diferente del almacenamiento físico de los datos. Pueden apuntar a los registros de la tabla mediante un identificador de fila.
- **Índices Columnstore**: Optimizados para consultas de procesamiento analítico en línea (OLAP), estos índices almacenan datos de manera columnar, lo que puede mejorar significativamente el rendimiento de ciertas consultas.

La selección y configuración adecuada de los índices puede reducir significativamente los tiempos de consulta.

los índices en SQL Server son esenciales para mejorar el rendimiento de las consultas, especialmente en tablas grandes donde pueden reducir significativamente el tiempo de acceso a los datos. A continuación, veremos cómo se crean y utilizan los índices en SQL Server, y algunos ejemplos prácticos.

Tipos de Índices

1. **Índices Clusterizados**: Reorganizan físicamente los datos de la tabla en el orden del índice. Cada tabla puede tener solo un índice clusterizado, ya que define el almacenamiento físico de los datos en la tabla.
2. **Índices No Clusterizados**: Mantienen un orden lógico que no afecta el orden físico de los datos. Pueden apuntar a los registros de datos mediante un identificador de fila si la tabla tiene un índice clusterizado o mediante punteros de fila si no lo tiene.

Creación de Índices

Índice Clusterizado

Supongamos que tienes una tabla llamada `Clientes` con las siguientes columnas: `ClienteID`, `Nombre`, y `Ciudad`. Para crear un índice clusterizado en `ClienteID`, usarías el siguiente comando SQL:

```
CREATE CLUSTERED INDEX IDX_ClienteID ON Clientes(ClienteID);
```

Este índice organiza físicamente la tabla `Clientes` en el orden de `ClienteID`, lo que puede hacer que las operaciones de búsqueda, inserción y eliminación que utilizan esta columna sean más rápidas.

Índice No Clusterizado

Si deseas crear un índice no clusterizado en la columna `Ciudad` para mejorar el rendimiento de las consultas que filtran por esta columna, el comando sería:

```
CREATE NONCLUSTERED INDEX IDX_Ciudad ON Clientes(Ciudad);
```

Este índice ayuda a acelerar las consultas que utilizan la columna `Ciudad` en la cláusula `WHERE`, pero no reorganiza físicamente los datos de la tabla.

Uso de Índices

Los índices mejoran el rendimiento de las consultas que utilizan las columnas indexadas en cláusulas como `WHERE`, `JOIN`, y `ORDER BY`. Por ejemplo, si frecuentemente ejecutas una consulta para encontrar clientes en una ciudad específica, el índice no clusterizado `IDX_Ciudad` mejorará el rendimiento de esta consulta:

```
SELECT Nombre FROM Clientes WHERE Ciudad = 'Madrid';
```

Índices únicos

Un índice `UNIQUE` garantiza que los datos en una columna, o un conjunto de columnas, sean únicos para todas las filas en la tabla. Esto es útil para mantener la integridad de los datos al asegurarse de que no se inserten duplicados inadvertidamente en la tabla.

Función de un Índice UNIQUE

Los índices `UNIQUE` no solo ayudan a mantener la integridad de los datos, sino que también mejoran el rendimiento de las consultas que buscan o filtran por las columnas que han sido indexadas de forma única. A diferencia de un índice regular, un índice `UNIQUE` crea una restricción en la tabla que impide la entrada de datos duplicados.

Ejemplo de Creación de un Índice UNIQUE

Supongamos que tienes una tabla llamada `Usuarios` con las siguientes columnas: `UsuarioID` (que ya es la clave primaria y por lo tanto única) y `Email`. Quieres asegurarte de que los emails de los usuarios sean únicos en la base de datos. Podrías crear un índice `UNIQUE` en la columna `Email` con el siguiente comando SQL:

```
CREATE UNIQUE INDEX UX_Email ON Usuarios(Email);
```

Este índice impedirá que dos usuarios se registren con el mismo correo electrónico. Si intentas insertar o actualizar un registro que cause un duplicado en la columna `Email`, SQL Server rechazará la operación y devolverá un error.

Creación de índices desde Python

Puedes crear índices en SQL Server directamente desde Python usando la biblioteca `pymssql`. Al igual que cualquier otra instrucción SQL, puedes ejecutar comandos para crear índices utilizando un cursor obtenido de una conexión `pymssql`. Esto te permite automatizar la administración de bases de datos y la configuración inicial desde scripts de Python.

Ejemplo de cómo crear un índice desde pymssql

Vamos a crear un índice `UNIQUE` en una columna específica de una tabla utilizando `pymssql`. Vamos a suponer que tienes una tabla llamada `Usuarios` con una columna `Email` y quieres asegurarte de que cada email sea único.

```
# ... crea la conexión y el cursor

# Comando SQL para crear un índice UNIQUE
```

```
create_index_sql = """
CREATE UNIQUE INDEX UX_Email ON Usuarios(Email);
"""

try:
    # Ejecutar el comando SQL
    cursor.execute(create_index_sql)
    # Guardar los cambios
    conn.commit()
    print("Índice UNIQUE creado exitosamente.")
except Exception as e:
    # Manejo de errores, por ejemplo, si el índice ya existe o si hay datos duplicados
    print("Error al crear el índice:", e)
finally:
    # Cerrar el cursor y la conexión
    cursor.close()
    conn.close()
```

Detalles importantes

1. **Manejo de errores**: Es importante manejar excepciones al crear índices, especialmente para manejar casos donde el índice ya existe o hay datos que violan la restricción `UNIQUE`.
2. **Confirmación de cambios**: No olvides llamar a `conn.commit()` después de ejecutar el comando de creación del índice para asegurar que los cambios se apliquen permanentemente a la base de datos.
3. **Cierre de recursos**: Siempre cierra el cursor y la conexión para liberar recursos de manera adecuada.

Ventajas de crear índices desde Python

Crear índices directamente desde un script de Python puede ser muy útil en varios contextos, como:

- **Automatización de despliegues**: Automatizar la configuración inicial de una base de datos en entornos de desarrollo, pruebas o producción.
- **Mantenimiento programado**: Automatizar tareas de mantenimiento de base de datos, incluyendo la creación y reconstrucción de índices.
- **Integración en aplicaciones**: Integrar la administración de bases de datos directamente en aplicaciones back-end que utilicen Python.

Esta flexibilidad puede ser especialmente valiosa en entornos donde las bases de datos necesitan ser configuradas o modificadas dinámicamente en respuesta a cambios en los requisitos de la aplicación o del sistema.

Consideraciones al Usar Índices UNIQUE

- **Rendimiento de Inserción y Actualización**: Al igual que otros índices, los índices `UNIQUE` incurren en un costo adicional durante las operaciones de inserción y actualización porque el sistema necesita mantener el índice actualizado. Esto puede ser especialmente significativo en tablas con un alto volumen de transacciones.
- **Uso de NULLs**: SQL Server permite múltiples entradas `NULL` en columnas indexadas de forma única, a menos que la columna o columnas también estén definidas como `NOT NULL`. Esto es importante tener en cuenta al diseñar la estructura de la tabla y el índice.

Aplicaciones de Índices UNIQUE

Los índices `UNIQUE` son particularmente útiles en escenarios donde necesitas asegurar la no duplicidad de información para elementos como:

- Números de identificación personal.
- Direcciones de correo electrónico.
- Números de serie de productos.
- Otros identificadores únicos de negocio.

En resumen, los índices `UNIQUE` en SQL Server son una herramienta esencial para la gestión de la integridad de los datos y pueden ayudar a mejorar el rendimiento de las consultas al proporcionar un camino de acceso rápido y directo a datos únicos.

Consideraciones Adicionales

- **Mantenimiento de Índices**: Los índices deben ser mantenidos, especialmente en tablas con alta actividad de inserción, actualización, o eliminación. Las operaciones DML (Data Manipulation Language) pueden fragmentar el índice, lo que eventualmente degradará el rendimiento.
- **Costo de Índices**: Aunque los índices pueden mejorar el rendimiento de las consultas, también tienen un costo. Consumen espacio adicional en disco y pueden ralentizar las operaciones de inserción, actualización y eliminación, ya que el índice debe ser actualizado cada vez que se modifican los datos.

Conclusión

Los índices son una herramienta poderosa para los administradores de bases de datos y desarrolladores que buscan optimizar el rendimiento de las consultas en SQL Server. Es importante evaluar y planificar cuidadosamente los índices en función de los patrones de acceso a los datos y las necesidades específicas del sistema. La creación de índices debe equilibrar el rendimiento de las consultas de lectura con el costo adicional en operaciones de escritura y el uso de almacenamiento.

3. Particionamiento

El particionamiento en SQL Server es una técnica para dividir grandes tablas y bases de datos en partes más pequeñas y manejables, sin cambiar la lógica de acceso a los datos. Los principales beneficios del particionamiento incluyen:

- **Mejora de rendimiento**: Reduce el número de filas a leer en las consultas si están bien diseñadas para aprovechar el particionamiento.
- **Mantenimiento más fácil**: Operaciones como reconstrucción de índices, respaldos y otras tareas de mantenimiento pueden ser ejecutadas en particiones individuales en lugar de en la tabla completa.
- **Disponibilidad mejorada**: Permite a los administradores acceder y modificar partes de la tabla mientras otras partes permanecen disponibles para operaciones críticas.

El particionamiento puede ser basado en rangos (por ejemplo, fechas o identificadores numéricos) o en listas (por ejemplo, categorías o regiones).

Conclusión

Un buen entendimiento y aplicación de consultas eficientes, índices adecuados y estrategias de particionamiento son cruciales para el manejo eficiente de bases de datos en SQL Server. Estas herramientas y técnicas ayudan a los administradores y desarrolladores a manejar mejor el rendimiento y la escalabilidad de las bases de datos, especialmente en entornos con grandes volúmenes de datos.

Detalles del particionamiento en SQL Server

El particionamiento de tablas en SQL Server es una técnica poderosa para manejar grandes volúmenes de datos al dividirlos en partes más manejables, pero manteniendo la visibilidad de los datos como si fueran una sola tabla. Esto puede mejorar significativamente el rendimiento de las consultas y facilitar la administración de los datos, especialmente para las operaciones de mantenimiento y respaldo.

¿Cómo Funciona el Particionamiento?

En SQL Server, el particionamiento se realiza sobre tablas y/o índices y está basado en una columna de la tabla que sirve como columna de partición. Generalmente, se utiliza una columna que tenga un rango de valores amplio y claramente definido, como fechas, números de identificación o regiones geográficas.

Componentes Clave del Particionamiento

1. **Función de partición**: Define cómo se distribuyen las filas en diferentes particiones basadas en los valores de la columna de partición.
2. **Esquema de partición**: Asocia la función de partición con los grupos de archivos físicos donde se almacenarán las particiones.

Ejemplo de Particionamiento

Supongamos que tienes una tabla `Ventas` con millones de registros. La tabla incluye una columna `FechaVenta` que indica cuándo se realizó cada venta. Deseas particionar esta tabla por año para mejorar el rendimiento de las consultas de datos históricos.

Paso 1: Crear Grupos de Archivos

Primero, debes preparar la base de datos con grupos de archivos, cada uno podría estar dedicado a un año específico:

```
ALTER DATABASE MiBaseDatos ADD FILEGROUP FG_Ventas2019;
ALTER DATABASE MiBaseDatos ADD FILEGROUP FG_Ventas2020;
ALTER DATABASE MiBaseDatos ADD FILEGROUP FG_Ventas2021;
```

Paso 2: Crear Archivos en los Grupos de Archivos

```
ALTER DATABASE MiBaseDatos ADD FILE (
    NAME = 'Ventas2019',
    FILENAME = 'C:\Data\Ventas2019.ndf',
    SIZE = 100MB,
    FILEGROWTH = 50MB
) TO FILEGROUP FG_Ventas2019;

ALTER DATABASE MiBaseDatos ADD FILE (
    NAME = 'Ventas2020',
    FILENAME = 'C:\Data\Ventas2020.ndf',
    SIZE = 100MB,
    FILEGROWTH = 50MB
) TO FILEGROUP FG_Ventas2020;

ALTER DATABASE MiBaseDatos ADD FILE (
    NAME = 'Ventas2021',
    FILENAME = 'C:\Data\Ventas2021.ndf',
    SIZE = 100MB,
    FILEGROWTH = 50MB
) TO FILEGROUP FG_Ventas2021;
```

Paso 3: Crear Función y Esquema de Partición

```
-- Crear la función de partición
CREATE PARTITION FUNCTION pf_VentasFecha (DATE)
AS RANGE RIGHT FOR VALUES ('2020-01-01', '2021-01-01');

-- Crear el esquema de partición
CREATE PARTITION SCHEME ps_VentasFecha
AS PARTITION pf_VentasFecha
TO (FG_Ventas2019, FG_Ventas2020, FG_Ventas2021, [PRIMARY]);
```

Paso 4: Crear o Modificar la Tabla para Usar el Particionamiento

```
-- Crear la tabla con el esquema de partición
CREATE TABLE Ventas (
    VentaID INT IDENTITY(1,1) PRIMARY KEY,
    FechaVenta DATE,
    Monto DECIMAL(10, 2),
    ProductoID INT
) ON ps_VentasFecha(FechaVenta);
```

Cómo Funciona

- Las ventas de antes del 2020 se almacenan en `FG_Ventas2019`.
- Las ventas de 2020 se almacenan en `FG_Ventas2020`.
- Las ventas de 2021 se almacenan en `FG_Ventas2021`.
- Las ventas posteriores a 2021 se almacenan en el grupo de archivos `PRIMARY` por defecto.

Este enfoque permite que las consultas que buscan datos específicos del año se ejecuten más rápidamente porque solo tienen que buscar en los datos de un grupo de archivos, en lugar de en toda la base de datos. Además, las operaciones de mantenimiento como los respaldos y las restauraciones pueden ser más granulares y eficientes, al poder enfocarse en particiones específicas en lugar de en toda la tabla.

Exclusividad de las particiones

Las particiones deben ser mutuamente excluyentes en SQL Server y en la mayoría de los sistemas de gestión de bases de datos que soportan particionamiento. Esto significa que cada fila de la tabla debe pertenecer a una y solo una partición, sin superposiciones entre los rangos o criterios que definen a cada partición.

Importancia de la Exclusividad

La exclusividad de las particiones garantiza que cada dato solo pueda ser almacenado en una partición. Esto es crítico para:

- **Eficiencia en Consultas**: Permite que el motor de la base de datos aplique eficientemente estrategias de optimización como el pruning de particiones, donde el motor puede ignorar por completo las particiones que no cumplen con los criterios de la consulta.
- **Mantenimiento Claro y Organizado**: Facilita tareas administrativas como respaldos, restauraciones, y mantenimientos de índices, al poder operar en particiones individuales.
- **Integridad de los Datos**: Evita la duplicidad y inconsistencias en los datos, lo cual podría complicar las operaciones de CRUD (Crear, Leer, Actualizar, Borrar) y afectar negativamente el rendimiento y la precisión de los resultados.

Cómo Asegurar la Exclusividad

En SQL Server, la función de partición define cómo se distribuyen los datos entre las particiones. Aquí se pueden emplear dos estrategias principales:

1. **RANGE LEFT**: En esta estrategia, el valor límite especificado en la función de partición es incluido en la partición izquierda (la partición que corresponde a valores menores). Por ejemplo, si tienes puntos de corte en '2020-01-01' y '2021-01-01', entonces los datos desde '2020-01-01' hasta '2020-12-31' se incluyen en la primera partición y '2021-01-01' en adelante en la siguiente.

2. **RANGE RIGHT**: En esta estrategia, el valor límite es incluido en la partición derecha (la partición que corresponde a valores iguales o mayores). Siguiendo el mismo ejemplo anterior, los datos anteriores a '2020-01-01' quedarían en la primera partición y desde '2020-01-01' en adelante en la segunda.

Ejemplo

Supongamos que defines una función de partición con `RANGE RIGHT` para particionar datos basados en años:

```
CREATE PARTITION FUNCTION PartitionFunctionYear (DATE)
AS RANGE RIGHT FOR VALUES ('2020-01-01', '2021-01-01', '2022-01-01');
```

En este caso:

- Datos anteriores a '2020-01-01' van a la primera partición.
- Datos desde '2020-01-01' hasta '2020-12-31' van a la segunda partición.
- Datos desde '2021-01-01' hasta '2021-12-31' van a la tercera partición.
- Datos desde '2022-01-01' en adelante van a la cuarta partición.

Cada rango es exclusivo y mutuamente excluyente, asegurando que no hay superposición de datos entre particiones.

Conclusión

La mutua exclusividad en las particiones es fundamental para el manejo efectivo y eficiente de las bases de datos, especialmente en entornos de grandes volúmenes de datos. Al diseñar y configurar particiones, es esencial asegurar que cada partición tenga un rango de valores claramente definido y no solapado para mantener la integridad y optimización del rendimiento de las consultas.

Los archivos .ndf

En SQL Server, las particiones no toman automáticamente archivos `.ndf` a menos que se configuren explícitamente para hacerlo. Los archivos `.ndf` son archivos secundarios de datos que puedes utilizar en SQL Server para distribuir datos a través de varios discos si lo deseas. Estos archivos pueden albergar tablas completas o incluso partes de tablas, como en el caso de tablas particionadas. Veamos cómo se configura esto y qué consecuencias tiene borrar dichos archivos.

Configuración de Archivos `.ndf` para Particiones

Cuando creas un esquema de partición en SQL Server, debes especificar a qué grupo de archivos (filegroup) pertenece cada partición. Cada grupo de archivos puede contener uno o más archivos físicos, que pueden ser `.mdf` (archivo primario) o `.ndf` (archivos secundarios).

Ejemplo de la creación de grupos de archivos

1. **Crear Grupos de Archivos**: Como parte de tu estrategia de particionamiento, defines varios grupos de archivos.

   ```
   ALTER DATABASE MiBaseDatos ADD FILEGROUP FG2020;
   ALTER DATABASE MiBaseDatos ADD FILEGROUP FG2021;
   ```

2. **Asignar Archivos `.ndf` a los Grupos de Archivos**: Asignas archivos específicos a cada grupo de archivos.

   ```
   ALTER DATABASE MiBaseDatos ADD FILE (
       NAME = 'Datos2020',
       FILENAME = 'C:\Data\Datos2020.ndf',
       SIZE = 100MB,
       FILEGROWTH = 50MB
   ) TO FILEGROUP FG2020;

   ALTER DATABASE MiBaseDatos ADD FILE (
       NAME = 'Datos2021',
       FILENAME = 'C:\Data\Datos2021.ndf',
       SIZE = 100MB,
   ```

```
    FILEGROWTH = 50MB
) TO FILEGROUP FG2021;
```

3. **Usar los Grupos de Archivos en el Esquema de Partición**: Cuando creas el esquema de partición, especificas qué grupo de archivos se usa para cada partición.

```
CREATE PARTITION SCHEME MiEsquemaDeParticion AS PARTITION MiFuncionDeParticion TO (FG2020, FG2021, PRIMARY);
```

¿Qué Pasa si se Borran los Archivos `.ndf`?

Borrar un archivo `.ndf` utilizado por SQL Server puede tener consecuencias graves:

- **Pérdida de Datos**: Si el archivo `.ndf` contiene datos (como sería el caso en un entorno de particionamiento donde los grupos de archivos están en archivos `.ndf` específicos), borrar este archivo resultará en una pérdida de datos irrecuperable para esa parte de la base de datos.
- **Falla de la Base de Datos**: SQL Server espera que estos archivos estén disponibles y funcionales. Si se borra un archivo `.ndf`, cualquier intento de acceso a datos contenidos en ese archivo resultará en errores. La base de datos puede incluso dejar de funcionar correctamente, dependiendo de qué datos se vieron afectados.
- **Necesidad de Restauración**: Para recuperar los datos o restaurar la funcionalidad completa de la base de datos, necesitarías restaurar los datos desde un respaldo, suponiendo que dispones de respaldos recientes y completos.

Recomendaciones

- **Nunca Borrar Archivos `.ndf` Manualmente**: Si necesitas mover o eliminar un archivo `.ndf`, hazlo a través de SQL Server Management Studio o mediante scripts T-SQL que primero desvinculen el archivo del grupo de archivos de forma segura.
- **Monitoreo y Respaldos**: Asegúrate de monitorear el uso del espacio de disco y tener una estrategia de respaldos robusta para proteger tus datos.

En resumen, los archivos `.ndf` en el contexto de particiones deben ser manejados con cuidado, y cualquier operación que involucre manipulación física de los archivos de datos debe ser realizada con un conocimiento pleno de las implicaciones y procedimientos adecuados.

Deshacer una partición

Para revertir el particionamiento de una tabla en SQL Server y hacer que la tabla deje de usar particiones, básicamente necesitas realizar una serie de operaciones que trasladen los datos a una tabla no particionada y luego eliminen la configuración de particionamiento existente. Este proceso implica recrear la tabla sin el esquema de partición y luego mover los datos de la tabla particionada a esta nueva tabla.

Pasos para Revertir una Partición

Veamos un enfoque general sobre cómo deshacer el particionamiento de una tabla y eliminar la dependencia de los archivos `.ndf`:

1. Crear una Nueva Tabla

Primero, debes crear una nueva tabla que tenga la misma estructura que la tabla particionada, pero sin aplicar el esquema de partición. Asegúrate de que esta tabla se cree en el grupo de archivos predeterminado o en uno que no esté asociado a un archivo `.ndf` específico.

```
CREATE TABLE dbo.Usuarios_New (
    UsuarioID INT PRIMARY KEY,
    Nombre NVARCHAR(100),
    Email NVARCHAR(100)
    -- Incluye todas las columnas adicionales necesarias
);
```

2. Migrar los Datos

Después de crear la nueva tabla, copia los datos de la tabla particionada a la nueva tabla usando un comando `INSERT INTO ... SELECT FROM`.

```
INSERT INTO dbo.Usuarios_New (UsuarioID, Nombre, Email)
SELECT UsuarioID, Nombre, Email
FROM dbo.Usuarios;
```

3. Renombrar Tablas

Una vez que los datos están copiados, y después de verificar que todo está correcto, puedes eliminar la tabla original y cambiar el nombre de la nueva tabla para que tome el lugar de la antigua.

```
-- Eliminar la tabla particionada
DROP TABLE dbo.Usuarios;

-- Cambiar el nombre de la nueva tabla
EXEC sp_rename 'dbo.Usuarios_New', 'Usuarios';
```

4. Eliminar la Configuración de Particionamiento

Si ya no vas a usar los grupos de archivos y esquemas de partición, deberías eliminarlos para limpiar la base de datos y evitar confusiones en el futuro.

```
-- Eliminar esquema de partición
DROP PARTITION SCHEME MiEsquemaDeParticion;

-- Eliminar función de partición
DROP PARTITION FUNCTION MiFuncionDeParticion;
```

```
-- Eliminar grupos de archivos si ya no se necesitan
ALTER DATABASE MiBaseDatos REMOVE FILEGROUP FG2020;
-- Repite para cada grupo de archivos
```

Nota: Antes de poder eliminar un grupo de archivos, debes asegurarte de que esté vacío. Puede que necesites eliminar los archivos físicos `.ndf` del grupo de archivos utilizando `ALTER DATABASE` para remover los archivos antes de poder eliminar el grupo de archivos.

5. Limpieza Final

Es posible que también necesites limpiar los archivos `.ndf` si ya no se utilizan.

```
-- Asegúrate de que los archivos no contengan datos y no estén en uso
ALTER DATABASE MiBaseDatos REMOVE FILE Datos2020;
```

Consideraciones

- **Respaldos**: Asegúrate de hacer un respaldo completo de los datos antes de comenzar este proceso.
- **Verificación**: Verifica cada paso antes de proceder al siguiente para asegurarte de que los datos se han trasladado correctamente y que la aplicación o usuarios dependientes de la base de datos no se vean afectados negativamente.
- **Tiempo de Inactividad**: Dependiendo de la cantidad de datos, este proceso puede llevar tiempo y podría requerir tiempo de inactividad para la aplicación, así que planifícalo adecuadamente.

Estos pasos te ayudarán a revertir el particionamiento de una tabla en SQL Server y a eliminar la dependencia de los archivos `.ndf`, devolviendo la tabla a un estado no particionado y simplificando la estructura física de la base de datos.

403. Uso de funciones y procedimientos almacenados

Las funciones y procedimientos almacenados en SQL Server son esenciales para comprender cómo estructurar aplicaciones de base de datos de manera eficiente y segura. Ambos son objetos de base de datos que permiten encapsular lógica de negocio que se ejecuta en el servidor de bases de datos, lo cual puede mejorar el rendimiento, la reutilización del código y la seguridad.

Funciones Almacenadas

Las funciones almacenadas en SQL Server son objetos de base de datos que pueden aceptar parámetros, realizar operaciones y retornar un resultado. Las funciones pueden ser de varios tipos, incluidos:

1. **Funciones Escalares**: Devuelven un valor único (no una tabla) y pueden ser usadas en consultas SQL similares a cómo se usan las funciones integradas como `GETDATE()` o `SUM()`.
2. **Funciones de Tabla**: Retornan un conjunto de registros; es decir, una tabla. Estas son especialmente útiles para ser utilizadas en cláusulas `FROM` de consultas SQL.
3. **Funciones de Tabla con Valores de Tabla (TVF)**: Similar a las funciones de tabla, pero permiten una mayor flexibilidad y complejidad en la manipulación de datos.

Ejemplo de Función Escalar

```
CREATE FUNCTION dbo.FnObtenerImpuesto(@Precio DECIMAL(10,2))
RETURNS DECIMAL(10,2)
AS
BEGIN
    DECLARE @Impuesto DECIMAL(10,2)
    SET @Impuesto = @Precio * 0.16
    RETURN @Impuesto
END
```

Esta función calcula un impuesto del 16% sobre un precio dado y devuelve el valor del impuesto.

Ejemplo de Función de Tabla

```
CREATE FUNCTION dbo.FnEmpleadosPorDepartamento(@DepartamentoID INT)
RETURNS TABLE
AS
RETURN
    (SELECT EmpleadoID, Nombre FROM Empleados WHERE DepartamentoID = @DepartamentoID)
```

Esta función devuelve una tabla de empleados pertenecientes a un departamento específico.

Creación de funciones desde Python

La creación y utilización de funciones en SQL Server desde Python usando `pymssql` es similar a trabajar con procedimientos almacenados, pero con algunas diferencias clave en la forma en que se definen y se invocan las funciones. Vamos a ver un ejemplo de cómo crear y utilizar una función que calcule el impuesto sobre un precio dado, devolviendo el monto del impuesto.

Ejemplo: Creación de una Función para Calcular el Impuesto

Supongamos que quieres crear una función en SQL Server que acepte un precio como entrada y devuelva el impuesto calculado a un 16%.

Definir y Crear la Función

A continuación, define la función en SQL. En este ejemplo, crearemos una función escalar que devuelve un valor decimal.

```
# Definición de la función en SQL
create_function_sql = """
IF EXISTS (SELECT * FROM sys.objects WHERE type = 'FN' AND name = 'CalcularImpuesto')
    DROP FUNCTION CalcularImpuesto;
GO

CREATE FUNCTION CalcularImpuesto(@Precio DECIMAL(10,2))
RETURNS DECIMAL(10,2)
AS
BEGIN
    RETURN @Precio * 0.16
END
GO
"""

# Dividir el comando para evitar problemas con 'GO'
commands = create_function_sql.split('GO')
for command in commands:
    if command.strip() != '':
        cursor.execute(command)
conn.commit()
```

Invocar la Función desde Python

Una vez que la función está creada, puedes invocarla para calcular impuestos sobre diferentes precios directamente en una consulta SQL.

```
# Preparar la consulta para invocar la función
query = "SELECT dbo.CalcularImpuesto(1000) AS Impuesto"

# Ejecutar la consulta
cursor.execute(query)

# Obtener y imprimir el resultado
result = cursor.fetchone()
print(f"Impuesto calculado: {result[0]}")
```

Consideraciones Adicionales

1. **Control de Excepciones**: Agregar control de excepciones para gestionar posibles errores durante la conexión, ejecución de comandos o cierre de la conexión es una buena práctica.
2. **Uso de `GO`**: Como se mencionó, `pymssql` no maneja directamente los comandos `GO`. Debes dividir tu script SQL en bloques separados antes de `GO` y ejecutarlos uno por uno si es necesario.

Este enfoque te permite automatizar la gestión de la lógica de negocio en el servidor de base de datos, mejorando la eficiencia de tus aplicaciones al reducir la carga de cálculos del lado del cliente y aprovechar las capacidades de SQL Server para el procesamiento de datos.

Procedimientos Almacenados

Los procedimientos almacenados son bloques de código que se pueden ejecutar con parámetros y que están diseñados para realizar operaciones más complejas que las funciones. Pueden realizar una serie de pasos, ejecutar consultas, manipular datos y hasta controlar transacciones.

Ventajas de los Procedimientos Almacenados

- **Mejora del rendimiento**: El código se compila y optimiza al ser creado y luego se almacena en el servidor.
- **Seguridad**: Pueden ejecutarse con permisos específicos, ayudando a asegurar la aplicación contra ataques como la inyección SQL.
- **Reducción del tráfico de red**: Al ejecutar operaciones complejas en el servidor, se reduce la cantidad de datos enviados sobre la red.

Ejemplo de Procedimiento Almacenado

```
CREATE PROCEDURE spAgregarEmpleado
    @Nombre NVARCHAR(100),
    @DepartamentoID INT,
    @Salario DECIMAL(10,2)
AS
BEGIN
    INSERT INTO Empleados (Nombre, DepartamentoID, Salario)
    VALUES (@Nombre, @DepartamentoID, @Salario)
END
```

Este procedimiento almacena agrega un nuevo empleado a la base de datos.

Uso en Aplicaciones

Tanto las funciones como los procedimientos almacenados se utilizan para encapsular lógica de negocio, asegurando que las operaciones de datos sean coherentes, estén optimizadas y centralizadas. Esto es especialmente útil en entornos empresariales donde múltiples aplicaciones o múltiples instancias de una aplicación necesitan realizar las mismas operaciones de datos de forma consistente.

En resumen, las funciones y procedimientos almacenados son herramientas cruciales en SQL Server para mejorar el rendimiento, la seguridad y la gestión del código en aplicaciones de bases de datos. Su uso adecuado puede significar grandes beneficios en la eficiencia y mantenimiento de sistemas de bases de datos grandes y complejos.

Creación de Procedimientos almacenados desde Python

Este enfoque es útil si necesitas automatizar la creación de procedimientos desde aplicaciones Python, especialmente en entornos donde Python actúa como un lenguaje de scripting para tareas administrativas o de gestión de bases de datos.

Ejemplo: Creación de un Procedimiento Almacenado para Insertar un Empleado

Supongamos que quieres crear un procedimiento almacenado llamado `spAgregarEmpleado` que inserta un nuevo empleado en una tabla `Empleados`. La tabla `Empleados` tiene las siguientes columnas: `EmpleadoID`, `Nombre`, y `Email`.

Definir y Crear el Procedimiento Almacenado

Define el procedimiento almacenado usando SQL y ejecútalo a través del cursor:

```
# Definición del procedimiento almacenado en SQL
create_sp_sql = """
IF EXISTS (SELECT * FROM sys.objects WHERE type = 'P' AND name = 'spAgregarEmpleado')
    DROP PROCEDURE spAgregarEmpleado;
GO

CREATE PROCEDURE spAgregarEmpleado
    @Nombre NVARCHAR(100),
    @Email NVARCHAR(100)
AS
BEGIN
    INSERT INTO Empleados (Nombre, Email)
    VALUES (@Nombre, @Email)
END
GO
"""

# Ejecuta el comando para crear el procedimiento almacenado
cursor.execute(create_sp_sql)
conn.commit()
```

Notas Importantes

1. **Control de Excepciones**: Es buena práctica agregar control de excepciones alrededor de tus operaciones de base de datos para manejar cualquier error que pueda ocurrir durante la conexión, ejecución de comandos, o cierre de la conexión.
2. **GO Statement**: `pymssql` no admite directamente múltiples comandos en una sola llamada `execute()` que incluyan `GO` (un delimitador de lotes de SQL Server). Si necesitas ejecutar múltiples declaraciones que incluyan `GO`, deberás dividirlas y ejecutarlas por separado o usar otro enfoque para manejar scripts que incluyan `GO`.

Ejemplo de Uso del Procedimiento

Una vez que el procedimiento almacenado está en su lugar, puedes invocarlo desde Python de la siguiente manera:

```
# Ejecuta el procedimiento almacenado
cursor.callproc('spAgregarEmpleado', ('Ana Ruiz', 'ana.ruiz@example.com'))
conn.commit()
```

Este ejemplo te proporciona una forma completa de cómo manejar procedimientos almacenados en SQL Server desde Python utilizando `pymssql`, desde su creación hasta su ejecución.

Recuperar el ID creado en el procedimiento almacenado

Para recuperar el ID del objeto que acabas de insertar en SQL Server utilizando `pymssql` en un procedimiento almacenado, puedes utilizar la función `SCOPE_IDENTITY()`. Esta función es útil para obtener el último ID generado en la misma sesión y el mismo ámbito, evitando problemas que pueden surgir al usar `@@IDENTITY` o `IDENT_CURRENT`, que podrían retornar valores incorrectos si hay múltiples conexiones o inserciones en paralelo.

Ejemplo de Procedimiento Almacenado

Primero, vamos a definir un procedimiento almacenado que inserta un registro en una tabla y luego devuelve el ID generado de ese registro.

1. Definición del Procedimiento Almacenado

Supongamos que tienes una tabla llamada `Usuarios` con una columna `UsuarioID` que es una clave primaria con autoincremento, junto con otras columnas como `Nombre` y `Email`.

```
CREATE TABLE Usuarios (
    UsuarioID INT IDENTITY(1,1) PRIMARY KEY,
    Nombre NVARCHAR(100),
    Email NVARCHAR(100)
);
```

Ahora, define el procedimiento almacenado que inserta un nuevo usuario y devuelve el ID generado:

```
CREATE PROCEDURE spAgregarUsuario
    @Nombre NVARCHAR(100),
    @Email NVARCHAR(100)
AS
BEGIN
    INSERT INTO Usuarios (Nombre, Email)
    VALUES (@Nombre, @Email);
```

```
    SELECT SCOPE_IDENTITY() AS NuevoUsuarioID;
END
```

2. Llamar al Procedimiento Almacenado desde Python

Una vez definido el procedimiento, puedes llamarlo desde Python utilizando `pymssql` y recuperar el ID del nuevo usuario.

```
# ... crea la conexión y el cursor

# Llamada al procedimiento almacenado
cursor.callproc('spAgregarUsuario', ('Ana Ruiz', 'ana.ruiz@example.com'))

# Recuperar y mostrar el ID del nuevo usuario
new_user_id = cursor.fetchone()[0]
print(f'El ID del nuevo usuario es: {new_user_id}')

# Confirmar la transacción
conn.commit()

# Nota: no debemos olvidar cerrar el cursor y la conexión cuando ya no se utilice
```

Notas Adicionales

- `SCOPE_IDENTITY()` **vs** `@@IDENTITY`: `SCOPE_IDENTITY()` es preferido sobre `@@IDENTITY` porque devuelve el último ID generado en el mismo ámbito, lo que evita problemas si hay otros triggers o inserciones que ocurren al mismo tiempo.
- **Manejo de Conexiones**: Siempre es buena práctica manejar excepciones y errores en tu código Python para asegurar que la conexión se cierre adecuadamente incluso si ocurre un error durante la ejecución del procedimiento.
- **Commit de la Transacción**: No olvides llamar a `commit()` en la conexión si necesitas que las modificaciones en la base de datos sean permanentes.

Este método te permitirá integrar de manera eficiente SQL Server con aplicaciones Python, aprovechando `pymssql` para gestionar datos que requieren integridad y seguridad como los IDs de usuarios recién creados.

404. Desarrollo de disparadores avanzados

Los disparadores (triggers) en SQL Server son objetos de base de datos especiales que se asocian a tablas o vistas y que se activan automáticamente cuando se realizan operaciones específicas sobre estos objetos, como inserciones, actualizaciones o eliminaciones. Los disparadores son herramientas poderosas para mantener la integridad de los datos, automatizar el procesamiento de los mismos, y realizar tareas de auditoría, entre otros usos.

Tipos de Disparadores en SQL Server

1. **Disparadores DML** (Data Manipulation Language): Se activan en respuesta a cambios en los datos provocados por `INSERT`, `UPDATE`, o `DELETE`.
    - **AFTER Triggers** (también conocidos como FOR Triggers): Se ejecutan después de que la operación de inserción, actualización o eliminación se ha completado exitosamente.
    - **INSTEAD OF Triggers**: Se ejecutan en lugar de la operación que los activó, permitiendo sobreescribir el comportamiento estándar de las operaciones de `INSERT`, `UPDATE`, o `DELETE`.
2. **Disparadores DDL** (Data Definition Language): Se activan en respuesta a cambios en la definición de la base de datos, como crear o alterar objetos de base de datos.
3. **Disparadores de Logon**: Se activan en respuesta a un evento de inicio de sesión en SQL Server.

Desarrollo de Disparadores Avanzados

Los disparadores avanzados suelen incorporar lógica compleja que va más allá de las operaciones simples de registro o validación. Pueden incluir:

- **Control de la integridad referencial compleja**: Asegurarse de que las operaciones en la base de datos no violen reglas de negocio específicas que no se pueden implementar a través de restricciones o relaciones clave foránea estándar.
- **Auditoría y registro de cambios**: Automatizar el seguimiento de quién hizo qué cambios y cuándo, almacenando detalles en tablas de auditoría.
- **Cascada de cambios**: Propagar automáticamente cambios a otras partes de la base de datos para mantener la sincronización y la integridad de los datos.
- **Validaciones personalizadas**: Verificar que las transacciones cumplan con reglas de negocio específicas antes de permitir que se realicen.

Ejemplo de un Disparador AFTER

Vamos a crear un disparador AFTER en una tabla llamada `Pedidos` para asegurar que cada vez que se inserte un nuevo pedido, se verifique y actualice el inventario correspondiente.

```
CREATE TRIGGER trgAfterInsertPedido
ON Pedidos
AFTER INSERT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @ProductoID int, @Cantidad int;

    SELECT @ProductoID = i.ProductoID, @Cantidad = i.Cantidad
    FROM inserted i;

    -- Actualizar el inventario reduciendo la cantidad pedida
    UPDATE Inventario
    SET Cantidad = Cantidad - @Cantidad
    WHERE ProductoID = @ProductoID;

    -- Insertar en la tabla de auditoría
    INSERT INTO AuditoriaPedidos(ProductoID, CantidadCambiada, FechaCambio)
    VALUES (@ProductoID, -@Cantidad, GETDATE());
```

```
END
GO
```

Consideraciones al Desarrollar Disparadores

- **Rendimiento**: Los disparadores pueden afectar significativamente el rendimiento de las operaciones de base de datos. Deben ser utilizados sabiamente y optimizados para no degradar el rendimiento.
- **Complejidad y mantenimiento**: Los disparadores que contienen lógica compleja pueden ser difíciles de mantener y depurar. Es vital mantener la lógica tan simple y clara como sea posible.
- **Pruebas exhaustivas**: Los disparadores deben ser probados exhaustivamente en escenarios que simulen el uso real para evitar efectos secundarios no deseados y asegurar que funcionan como se espera.

El desarrollo de disparadores avanzados requiere una comprensión sólida de los eventos que activan los disparadores y cómo interactúan con otras operaciones de base de datos para implementar soluciones eficaces y eficientes que apoyen las operaciones y la integridad de los datos.

Intercepción en las consultas

En SQL Server, no existe un disparador `BEFORE` como tal, como lo encuentras en otros sistemas de gestión de bases de datos como PostgreSQL o MySQL. Sin embargo, SQL Server ofrece los disparadores `INSTEAD OF`, que funcionan de manera similar al permitirte intervenir antes de que se complete una operación de inserción, actualización o eliminación. Los disparadores `INSTEAD OF` pueden ser utilizados para revisar o modificar los datos antes de que se realicen cambios en la base de datos, ofreciendo un nivel de control previo a la acción que es similar al proporcionado por un disparador `BEFORE`.

Uso de un Disparador `INSTEAD OF`

Los disparadores `INSTEAD OF` se ejecutan en lugar de la operación de inserción, actualización o eliminación original, lo que te permite realizar validaciones o cambios en los datos antes de proceder con la operación. Veamos un ejemplo de cómo utilizar un disparador `INSTEAD OF` para verificar y modificar datos antes de una inserción:

Ejemplo de Disparador `INSTEAD OF INSERT`

Supongamos que tienes una tabla llamada `Empleados` con las columnas `EmpleadoID`, `Nombre`, y `Email`, y quieres asegurarte de que el email no esté vacío antes de insertar un nuevo registro.

```
CREATE TABLE Empleados (
    EmpleadoID INT IDENTITY(1,1) PRIMARY KEY,
    Nombre NVARCHAR(100),
    Email NVARCHAR(100) NOT NULL
);

CREATE TRIGGER trgInsteadOfInsert
ON Empleados
INSTEAD OF INSERT
AS
BEGIN
    SET NOCOUNT ON;

    -- Verificar que el email no esté vacío
    IF EXISTS (SELECT * FROM inserted WHERE Email IS NULL OR Email = '')
    BEGIN
        RAISERROR ('El email no puede estar vacío', 16, 1);
        RETURN;
    END

    -- Si todo está correcto, insertar el registro
    INSERT INTO Empleados (Nombre, Email)
    SELECT Nombre, Email FROM inserted;
END
GO
```

Cómo Funciona Este Disparador

1. **Verificación de la Condición**: Antes de que se inserte el registro, el disparador verifica si el campo `Email` está vacío. Si encuentra que el email está vacío, genera un error y detiene la inserción.
2. **Inserción de Datos**: Si el campo `Email` pasa la verificación, el disparador procede a insertar los datos en la tabla `Empleados`.

Consideraciones

- **Rendimiento**: Aunque los disparadores `INSTEAD OF` ofrecen una gran flexibilidad, pueden afectar el rendimiento si se usan en operaciones que involucran grandes volúmenes de datos o son muy frecuentes. Debe considerarse cuidadosamente dónde y cómo se utilizan.
- **Mantenimiento**: La lógica dentro de los disparadores puede hacer que el mantenimiento de la base de datos sea más complicado. Es importante mantener la lógica de los disparadores tan simple y clara como sea posible y asegurarse de que esté bien documentada.

En resumen, aunque SQL Server no tiene disparadores `BEFORE` específicos, los disparadores `INSTEAD OF` ofrecen una funcionalidad comparable que te permite manejar y validar datos antes de que las operaciones de inserción, actualización o eliminación sean efectivamente realizadas en la base de datos.

En SQL Server, los disparadores `INSTEAD OF` reemplazan completamente la sentencia original de inserción, actualización o eliminación por la que se activan. Esto significa que, a diferencia de un disparador `AFTER`, que simplemente responde a una operación que ya ha ocurrido, un disparador `INSTEAD OF` toma el control total de la operación y es responsable de implementar explícitamente cualquier lógica de inserción, actualización o eliminación que se supone debe ocurrir.

¿Qué Implica Esto?

Cuando creas un disparador `INSTEAD OF`, debes proporcionar toda la lógica necesaria para manejar los datos de manera apropiada, incluyendo la inserción, actualización o eliminación de registros en la base de datos. Si no incluyes explícitamente esta lógica en el cuerpo del disparador, las operaciones originales que deberían haber ocurrido (como insertar un nuevo registro o actualizar uno existente) no se realizarán automáticamente.

Ejemplo Práctico

Vamos a revisar nuevamente el ejemplo del disparador `INSTEAD OF INSERT` para la tabla `Empleados` y explicar cómo asegurarte de que las inserciones se realicen adecuadamente a pesar de interceptar la operación original:

```
CREATE TRIGGER trgInsteadOfInsert
ON Empleados
INSTEAD OF INSERT
AS
BEGIN
    SET NOCOUNT ON;

    -- Verificar que el email no esté vacío
    IF EXISTS (SELECT * FROM inserted WHERE Email IS NULL OR Email = '')
    BEGIN
        RAISERROR ('El email no puede estar vacío', 16, 1);
        RETURN;
    END

    -- Inserción explícita de los registros validados
    INSERT INTO Empleados (Nombre, Email)
    SELECT Nombre, Email FROM inserted;
END
GO
```

En este disparador:

- **Validación**: Se verifica primero que ningún `Email` esté vacío.
- **Inserción Explícita**: Solo si la validación es exitosa, se procede a insertar explícitamente los datos en la tabla `Empleados` utilizando los datos de la pseudo-tabla `inserted`. Esto es crucial porque la inserción original que habría activado el disparador no ocurrirá a menos que se especifique explícitamente aquí.

Consideraciones Importantes

- **Control Completo**: Los disparadores `INSTEAD OF` te dan un control completo sobre lo que sucede, pero también te imponen la responsabilidad de asegurarte de que todas las operaciones necesarias se lleven a cabo correctamente.
- **Complejidad y Riesgo de Error**: Dado que debes manejar explícitamente las operaciones de la base de datos, hay un mayor riesgo de errores si no se replican completamente las intenciones originales de la operación.
- **Uso con Cautela**: Debido a su potencial para complicar y posiblemente ralentizar las operaciones de la base de datos, los disparadores `INSTEAD OF` deben usarse con cautela y solo cuando sean claramente beneficiosos para manejar casos específicos que no pueden ser tratados de manera más sencilla.

En resumen, los disparadores `INSTEAD OF` en SQL Server no permiten que la sentencia original se ejecute automáticamente. Es tu responsabilidad dentro del disparador asegurarte de que cualquier acción necesaria, como inserciones o actualizaciones, se realice explícitamente.

Acceder a los valores originales

En SQL Server, cuando se usa un disparador `INSTEAD OF` o un disparador `AFTER`, puedes acceder a los valores de la sentencia original que activó el disparador a través de las tablas especiales llamadas `inserted` y `deleted`. Estas tablas son usadas por SQL Server para almacenar los valores de fila antes y después de que ocurra un evento de datos, permitiendo que el disparador actúe basado en estos valores. Veamos cómo funcionan estas tablas y cómo utilizarlas para extraer valores:

Tablas `inserted` y `deleted`

- **Tabla `inserted`**: En los disparadores de `INSERT` y `UPDATE`, esta tabla contiene los valores de las filas tal como aparecerán después de la operación. Para un disparador `INSERT`, `inserted` contiene los valores de las nuevas filas que se están insertando en la base de datos.
- **Tabla `deleted`**: En los disparadores de `DELETE` y `UPDATE`, esta tabla contiene los valores de las filas antes de la operación. Para un disparador `DELETE`, `deleted` contiene los valores de las filas que están siendo eliminadas.

Acceder a los Valores en Disparadores `INSTEAD OF`

Cuando creas un disparador `INSTEAD OF`, puedes usar estas tablas para acceder a los valores originales de la operación que el disparador intercepta, lo que te permite realizar lógica condicional, validaciones, o manipulaciones antes de que se complete la operación real.

Ejemplo con `INSTEAD OF UPDATE`

Supongamos que tienes una tabla `Empleados` y quieres asegurarte de que ciertas condiciones se cumplan antes de permitir una actualización en la tabla.

```
CREATE TRIGGER trgInsteadOfUpdate
ON Empleados
INSTEAD OF UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    -- Ejemplo de verificación: asegurar que el email no esté vacío
    IF EXISTS (SELECT * FROM inserted WHERE Email IS NULL OR Email = '')
    BEGIN
        RAISERROR ('El email no puede estar vacío', 16, 1);
        RETURN;
    END

    -- Actualizar la fila solo si pasa las validaciones
    UPDATE Empleados
    SET
        Nombre = i.Nombre,
        Email = i.Email
    FROM
```

```
        inserted i
    WHERE
        Empleados.EmpleadoID = i.EmpleadoID;
END
GO
```

En este ejemplo, el disparador usa la tabla `inserted` para verificar si algún `Email` está vacío antes de permitir la actualización. Si la validación falla, se detiene la operación con un error.

Consideraciones

- **Rendimiento**: El uso de disparadores, especialmente aquellos que implementan lógica compleja o actúan sobre tablas grandes, puede tener un impacto en el rendimiento. Asegúrate de probar y optimizar tus disparadores.
- **Manejo de múltiples filas**: Ten en cuenta que `inserted` y `deleted` pueden contener múltiples filas. Asegúrate de que tu disparador pueda manejar correctamente múltiples filas de manera eficiente.
- **Pruebas**: Realiza pruebas exhaustivas para asegurarte de que tus disparadores se comportan como se espera en todos los casos posibles, incluyendo transacciones que involucren múltiples filas o valores límite.

Utilizar las tablas `inserted` y `deleted` permite a los desarrolladores acceder y manipular datos basándose en el estado antes y después de una operación propuesta, proporcionando un poderoso mecanismo para asegurar la integridad y las reglas de negocio directamente dentro de la base de datos.

405. Trabajo con cursores y variables

Trabajar con cursores y variables en SQL Server te permite manejar datos de manera más controlada y realizar operaciones más complejas que simplemente seleccionar o actualizar datos.

Variables en SQL Server

Las variables en SQL Server te permiten almacenar datos temporales para usar en tus consultas. Se declaran usando la palabra clave `DECLARE` y luego se les puede asignar un valor con `SET` o `SELECT`.

Ejemplo:

```
DECLARE @EmployeeID int;
SET @EmployeeID = 1;

-- Usar la variable en una consulta
SELECT * FROM Employees WHERE EmployeeID = @EmployeeID;
```

Cursores en SQL Server

Los cursores son útiles cuando necesitas procesar las filas de una tabla de manera secuencial, realizar operaciones fila por fila, o cuando cada fila requiere una lógica compleja que no puede ser realizada con un simple `SET` de operaciones. Sin embargo, deben ser usados con cautela ya que pueden ser menos eficientes que las operaciones de conjunto.

Pasos para usar un cursor:

1. **Declarar el cursor:** Define el conjunto de resultados y el cursor con `DECLARE CURSOR`.
2. **Abrir el cursor:** Abre el cursor para comenzar a acceder a las filas.
3. **Fetch:** Recupera la siguiente fila del conjunto de resultados del cursor y mueve el cursor adelante.
4. **Procesar la fila:** Realiza las operaciones necesarias con los datos de la fila.
5. **Cerrar el cursor:** Cierra el cursor una vez que terminas de procesar todas las filas.
6. **Desasignar el cursor:** Libera los recursos asociados con el cursor.

Ejemplo de uso de cursor:

```
DECLARE @EmployeeID int, @Name nvarchar(50);

-- Declarar el cursor
DECLARE employee_cursor CURSOR FOR
SELECT EmployeeID, Name FROM Employees;

-- Abrir el cursor
OPEN employee_cursor;

-- Fetch inicial
FETCH NEXT FROM employee_cursor INTO @EmployeeID, @Name;

-- Loop para procesar las filas
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'Employee ID: ' + CAST(@EmployeeID AS nvarchar(10)) + ', Name: ' + @Name;

    -- Fetch siguiente
    FETCH NEXT FROM employee_cursor INTO @EmployeeID, @Name;
END

-- Cerrar el cursor
CLOSE employee_cursor;

-- Desasignar el cursor
DEALLOCATE employee_cursor;
```

Recomendaciones

- **Evitar el uso de cursores siempre que sea posible.** En lugar de usar cursores, intenta usar operaciones de conjunto que son generalmente más eficientes en SQL Server.
- **Usa transacciones si modificas datos:** Esto asegura que tus modificaciones sean manejadas correctamente y puedas revertirlas si algo sale mal.

- **Optimiza tu cursor si es necesario:** Al declarar un cursor, puedes especificar opciones como `READ_ONLY` o `FAST_FORWARD` para mejorar el rendimiento dependiendo del caso de uso.

Utiliza estas herramientas con precaución y siempre que sea posible, intenta reemplazar los cursores por consultas que manejen sets de datos para obtener un rendimiento óptimo.

Uso de variables fuera de la transacción

Sí, las variables en SQL Server pueden ser usadas fuera de transacciones sin ningún problema. Las variables son independientes de las transacciones, lo que significa que puedes declararlas y usarlas para almacenar y manipular datos en cualquier parte de tu script o procedimiento almacenado, ya sea dentro o fuera de una transacción.

Características de las variables fuera de transacciones

- **Ámbito de vida:** Las variables tienen un ámbito local al batch, procedimiento almacenado, trigger o bloque de código donde son declaradas. Una vez que el código termina de ejecutarse, las variables son desechadas.
- **Independencia:** Las operaciones realizadas con variables no son afectadas por `COMMIT` o `ROLLBACK` de una transacción, dado que las variables no forman parte del estado de la transacción. Esto significa que los cambios en las variables no se revertirán si la transacción se deshace.
- **Flexibilidad de uso:** Puedes usar variables para preparar datos, guardar resultados temporales, o controlar la lógica de flujo en tus scripts, independientemente de las transacciones.

Ejemplo de uso de variables fuera de transacciones

```
DECLARE @Count int;

-- Asignar un valor a la variable
SET @Count = (SELECT COUNT(*) FROM Employees);

-- Usar la variable para realizar una condición
IF @Count > 0
BEGIN
    PRINT 'Hay empleados en la tabla.';
END
ELSE
BEGIN
    PRINT 'No hay empleados en la tabla.';
END
```

En este ejemplo, la variable `@Count` se usa para almacenar el número de empleados en una tabla y luego se usa en una instrucción condicional para imprimir un mensaje basado en el valor de la variable. Todo esto se realiza fuera de cualquier transacción.

Consideraciones

Mientras que las variables son muy útiles para almacenar información temporal, es importante recordar que su uso debe ser bien pensado, especialmente en procedimientos almacenados y scripts que se ejecutan frecuentemente, para evitar problemas de rendimiento. Además, asegúrate de limpiar o reasignar variables en scripts largos para evitar confusión o resultados incorrectos debido a valores residuales de ejecuciones anteriores.

Uso de Variables en transacciones

Veamos cómo podemos combinar el uso de variables y transacciones en SQL Server para realizar operaciones seguras sobre la base de datos y generar un reporte sencillo al final de la transacción. El ejemplo se enfocará en una transacción que actualiza los salarios de los empleados y luego imprime un reporte del estado final de los cambios.

Ejemplo de Transacción con Variables y Reporte

Imaginemos que necesitamos actualizar los salarios de algunos empleados y queremos asegurarnos de que todas las actualizaciones se hagan correctamente o que ninguna se haga (para mantener la integridad de la base de datos). Además, queremos un reporte que muestre cuántos empleados fueron actualizados y el total de incrementos realizados.

```
-- Declara las variables necesarias para la transacción
DECLARE @EmployeeCount int = 0;
DECLARE @TotalIncrement decimal(10,2) = 0.00;

BEGIN TRY
    -- Inicia la transacción
    BEGIN TRANSACTION

    -- Actualiza el salario de los empleados y cuenta los afectados
    UPDATE Employees
    SET Salary = Salary + 1000
    WHERE DepartmentID = 5 AND IsActive = 1;

    -- Obtiene el número de empleados actualizados
    SELECT @EmployeeCount = @@ROWCOUNT;

    -- Calcula el total incrementado
    SET @TotalIncrement = @EmployeeCount * 1000;

    -- Si todo va bien, confirma la transacción
    COMMIT TRANSACTION;

    -- Imprime el reporte
    PRINT 'Transacción completada exitosamente.';
    PRINT 'Número de empleados actualizados: ' + CAST(@EmployeeCount AS varchar(10));
    PRINT 'Total incrementado en salarios: $' + CAST(@TotalIncrement AS varchar(10));

END TRY
BEGIN CATCH
    -- Si algo sale mal, revierte la transacción
    ROLLBACK TRANSACTION;
```

```
    -- Imprime el error ocurrido
    PRINT 'Error en la transacción: ' + ERROR_MESSAGE();
END CATCH;
```

Detalles del Código

1. **Variables:** Se declaran variables para almacenar el número de empleados actualizados y el total de incrementos realizados en sus salarios.
2. **Transacción:** Utiliza `BEGIN TRANSACTION` para asegurar que todas las actualizaciones se realicen como una sola unidad de trabajo.
3. **Actualización y Cálculo:** Actualiza los salarios y calcula el número de empleados afectados y el total de los incrementos.
4. **Manejo de Excepciones:** Utiliza bloques `TRY...CATCH` para manejar errores. Si ocurre un error durante la transacción, se ejecutará el bloque `CATCH`, que revertirá la transacción y notificará el error.
5. **Reporte:** Imprime un reporte con los resultados de la transacción, utilizando las variables para mostrar cuántos empleados fueron afectados y cuánto fue el total del incremento.

Este ejemplo demuestra cómo las transacciones y variables pueden ser utilizadas conjuntamente para realizar operaciones seguras en la base de datos y proporcionar feedback inmediato sobre los resultados de estas operaciones.

Manejo de errores en las transacciones

Para simular un error dentro de una transacción en SQL Server y luego atraparlo para hacer un rollback, puedes usar el bloque `TRY...CATCH` junto con una condición de error artificial o una operación que sepas que fallará. Esto permite probar la robustez de la transacción y asegurarte de que tu código maneja correctamente los errores para mantener la integridad de la base de datos.

Ejemplo de Transacción con Error Simulado y Rollback

En este ejemplo, vamos a simular un error en una transacción que intenta actualizar los salarios de los empleados. Simularemos el error intentando insertar un valor que viole una restricción de clave foránea o que sea de un tipo de datos incorrecto.

```
-- Declara las variables necesarias para la transacción
DECLARE @EmployeeCount int = 0;
DECLARE @TotalIncrement decimal(10,2) = 0.00;

BEGIN TRY
    -- Inicia la transacción
    BEGIN TRANSACTION

    -- Actualización válida del salario
    UPDATE Employees
    SET Salary = Salary + 1000
    WHERE DepartmentID = 5 AND IsActive = 1;

    -- Obtiene el número de empleados actualizados
    SELECT @EmployeeCount = @@ROWCOUNT;

    -- Calcula el total incrementado
    SET @TotalIncrement = @EmployeeCount * 1000;

    -- Instrucción que se espera que falle: inserción incorrecta
    -- Aquí intentamos insertar un valor incorrecto en una columna que espera un entero
    INSERT INTO Employees(DepartmentID) VALUES ('invalid_data'); -- Suponiendo que DepartmentID es de tipo INT

    -- Si todo va bien (no se espera), confirma la transacción
    COMMIT TRANSACTION;

    -- Imprime el reporte
    PRINT 'Transacción completada exitosamente.';
    PRINT 'Número de empleados actualizados: ' + CAST(@EmployeeCount AS varchar(10));
    PRINT 'Total incrementado en salarios: $' + CAST(@TotalIncrement AS varchar(10));

END TRY
BEGIN CATCH
    -- Si algo sale mal, revierte la transacción
    ROLLBACK TRANSACTION;

    -- Imprime el error ocurrido
    PRINT 'Error en la transacción: ' + ERROR_MESSAGE();
END CATCH;
```

Explicación del Código

- **Variables:** Se inicia declarando variables para contar los empleados y sumar el incremento total.
- **Bloque `TRY...CATCH`:** Se inicia una transacción para actualizar los salarios y luego se introduce deliberadamente una instrucción `INSERT` con un valor erróneo. Esto es para garantizar que se genere un error.
- **Error Simulado:** La inserción de un valor de texto en una columna que espera un entero (`DepartmentID`) causa un error de tipo de datos que es inmediatamente atrapado por el bloque `CATCH`.
- **Manejo de Errores:** En el bloque `CATCH`, la transacción se revierte utilizando `ROLLBACK TRANSACTION`, asegurando que ninguna de las operaciones anteriores afecte permanentemente la base de datos.
- **Reporte de Errores:** Se imprime un mensaje de error usando `ERROR_MESSAGE()`, el cual proporciona detalles sobre el error que causó la entrada al bloque `CATCH`.

Este enfoque garantiza que si algo va mal durante la transacción, los cambios no se cometerán y el sistema informará adecuadamente sobre el error. Es una buena práctica para mantener la consistencia y la integridad de los datos en aplicaciones de bases de datos críticas.

Validaciones con error en las transacciones

Podemos adaptar el ejemplo para incluir una validación que verifique si el nuevo salario de un empleado excede un límite específico antes de aplicar la actualización. Si este límite se supera, la transacción se revertirá y se generará un mensaje de error.

Ejemplo de Transacción con Validación de Límite de Salario

En este ejemplo, estableceremos un límite máximo para el salario de $100,000.00. Si el incremento del salario resulta en un salario que excede este límite, la transacción se revertirá.

```
-- Declara las variables necesarias para la transacción
DECLARE @EmployeeCount int = 0;
DECLARE @TotalIncrement decimal(10,2) = 0.00;
DECLARE @MaxSalaryLimit decimal(10,2) = 100000.00; -- Límite máximo del salario

BEGIN TRY
    -- Inicia la transacción
    BEGIN TRANSACTION

    -- Actualización del salario con validación del límite
    UPDATE Employees
    SET Salary = Salary + 1000
    WHERE DepartmentID = 5 AND IsActive = 1
      AND (Salary + 1000) <= @MaxSalaryLimit;

    -- Obtiene el número de empleados actualizados
    SELECT @EmployeeCount = @@ROWCOUNT;

    -- Calcula el total incrementado, solo si la actualización fue exitosa
    IF @EmployeeCount > 0
    BEGIN
        SET @TotalIncrement = @EmployeeCount * 1000;
        -- Si todo va bien, confirma la transacción
        COMMIT TRANSACTION;
        PRINT 'Transacción completada exitosamente.';
        PRINT 'Número de empleados actualizados: ' + CAST(@EmployeeCount AS varchar(10));
        PRINT 'Total incrementado en salarios: $' + CAST(@TotalIncrement AS varchar(10));
    END
    ELSE
    BEGIN
        -- Ningún empleado fue actualizado, lo que implica superación del límite
        THROW 50000, 'El incremento de salario supera el límite máximo permitido.', 1;
    END

END TRY
BEGIN CATCH
    -- Si algo sale mal, revierte la transacción
    ROLLBACK TRANSACTION;

    -- Imprime el error ocurrido
    PRINT 'Error en la transacción: ' + ERROR_MESSAGE();
END CATCH;
```

Detalles del Código

1. **Validación de Límite:** Antes de actualizar el salario, se verifica si el nuevo salario (salario actual + incremento) supera el límite establecido.
2. **Actualización Condicional:** La cláusula `WHERE` en la declaración `UPDATE` incluye la condición del límite de salario, asegurando que solo se actualicen aquellos registros que no superen el máximo.
3. **Manejo de Transacciones:** Si no se actualiza ningún registro debido a la restricción del límite de salario, se utiliza `THROW` para lanzar una excepción explícitamente, lo cual es capturado por el bloque `CATCH` para hacer un `ROLLBACK`.
4. **Reporte y Error:** Si la transacción es exitosa, se imprime un mensaje de éxito. Si se lanza una excepción, ya sea por la validación del límite o por otro error, se maneja en el bloque `CATCH`, revirtiendo cualquier cambio y reportando el error.

Este enfoque asegura que la integridad de los datos se mantiene al prevenir actualizaciones de datos que violen las políticas de negocio establecidas, y proporciona un mecanismo robusto para manejar errores y excepciones.

Capturar el código de error en una transacción

Para capturar y verificar específicamente el código de error en el bloque `CATCH` de una transacción SQL Server y reaccionar en consecuencia, puedes usar la función `ERROR_NUMBER()`. Esta función retorna el número de error que provocó la entrada al bloque `CATCH`. Puedes utilizar este número en una condición `IF` para determinar la causa del error y responder de manera apropiada.

Ejemplo Modificado para Manejar el Código de Error 50000

Vamos a adaptar el ejemplo anterior para incluir la verificación del código de error. Si el código de error es 50000, imprimirá un mensaje específico que explique que el incremento del salario superó el límite permitido.

```
-- Declara las variables necesarias para la transacción
DECLARE @EmployeeCount int = 0;
DECLARE @TotalIncrement decimal(10,2) = 0.00;
DECLARE @MaxSalaryLimit decimal(10,2) = 100000.00; -- Límite máximo del salario

BEGIN TRY
    -- Inicia la transacción
    BEGIN TRANSACTION

    -- Actualización del salario con validación del límite
    UPDATE Employees
    SET Salary = Salary + 1000
    WHERE DepartmentID = 5 AND IsActive = 1
      AND (Salary + 1000) <= @MaxSalaryLimit;
```

```
    -- Obtiene el número de empleados actualizados
    SELECT @EmployeeCount = @@ROWCOUNT;

    -- Calcula el total incrementado, solo si la actualización fue exitosa
    IF @EmployeeCount > 0
    BEGIN
        SET @TotalIncrement = @EmployeeCount * 1000;
        -- Si todo va bien, confirma la transacción
        COMMIT TRANSACTION;
        PRINT 'Transacción completada exitosamente.';
        PRINT 'Número de empleados actualizados: ' + CAST(@EmployeeCount AS varchar(10));
        PRINT 'Total incrementado en salarios: $' + CAST(@TotalIncrement AS varchar(10));
    END
    ELSE
    BEGIN
        -- Ningún empleado fue actualizado, lo que implica superación del límite
        THROW 50000, 'El incremento de salario supera el límite máximo permitido.', 1;
    END

END TRY
BEGIN CATCH
    -- Si algo sale mal, revierte la transacción
    ROLLBACK TRANSACTION;

    -- Verifica el código de error
    IF ERROR_NUMBER() = 50000
    BEGIN
        PRINT 'La transacción falló porque el incremento del salario supera el límite máximo permitido.';
    END
    ELSE
    BEGIN
        PRINT 'Error en la transacción: ' + ERROR_MESSAGE();
    END
END CATCH;
```

Explicación del Código

1. **Manejo de Error Específico:** Dentro del bloque `CATCH`, se usa `IF ERROR_NUMBER() = 50000` para verificar si el error que provocó el `CATCH` es debido a que el incremento del salario supera el límite establecido.
2. **Mensaje Específico:** Si el error es 50000, se imprime un mensaje detallado explicando que el fallo se debe al exceso del límite de salario permitido. Para otros errores, se imprime el mensaje de error general usando `ERROR_MESSAGE()`.

Usar `ERROR_NUMBER()` de esta manera permite manejar errores de forma más granular, proporcionando retroalimentación específica y clara sobre la naturaleza del problema, lo cual es particularmente útil en ambientes de producción donde el diagnóstico rápido de problemas puede ser crucial.

Uso de Variables en Funciones

Usar variables dentro de una función en SQL Server es bastante similar al uso en procedimientos almacenados, pero generalmente las funciones están diseñadas para devolver un valor o una tabla y son más restrictivas en términos de lo que pueden hacer (por ejemplo, no pueden realizar operaciones que modifiquen la base de datos directamente).

Tipos de Funciones en SQL Server

1. **Funciones Escalares:** Devuelven un valor único.
2. **Funciones de Tabla:** Devuelven una tabla que puede ser utilizada como cualquier otra tabla en una consulta SQL.

Ejemplo de una Función Escalar

Vamos a crear una función escalar que calcula el salario anual de un empleado basándose en su salario mensual. Este ejemplo demostrará cómo se pueden usar variables dentro de una función escalar para realizar cálculos.

```
CREATE FUNCTION CalculateAnnualSalary
(
    @MonthlySalary decimal(10,2)
)
RETURNS decimal(10,2)
AS
BEGIN
    -- Declara una variable para almacenar el salario anual
    DECLARE @AnnualSalary decimal(10,2);

    -- Calcula el salario anual
    SET @AnnualSalary = @MonthlySalary * 12;

    -- Devuelve el salario anual
    RETURN @AnnualSalary;
END;
GO
```

Cómo Usar la Función Escalar

Una vez creada la función, puedes llamarla como parte de una consulta SQL para calcular el salario anual de los empleados basado en su salario mensual almacenado en una tabla.

```
SELECT EmployeeID, Name, MonthlySalary, dbo.CalculateAnnualSalary(MonthlySalary) AS AnnualSalary
FROM Employees;
```

Ejemplo de una Función de Tabla

Las funciones de tabla pueden ser un poco más complejas ya que devuelven conjuntos de datos. A continuación, un ejemplo de cómo usar variables en una función de tabla para devolver una lista filtrada de empleados basada en el salario mínimo.

```
CREATE FUNCTION GetEmployeesByMinSalary
(
    @MinSalary decimal(10,2)
)
RETURNS TABLE
AS
RETURN
(
    SELECT EmployeeID, Name, Salary
    FROM Employees
    WHERE Salary >= @MinSalary
);
```

Cómo Usar la Función de Tabla

Esta función puede ser utilizada en una consulta como una tabla normal, permitiendo un filtrado adicional o unión con otras tablas.

```
SELECT *
FROM dbo.GetEmployeesByMinSalary(30000.00)
ORDER BY Salary DESC;
```

Consideraciones

- **Variables:** Las funciones, especialmente las escalares, pueden utilizar variables para realizar cálculos, pero estas variables solo existen dentro del contexto de la función.
- **Restricciones:** Las funciones en SQL Server no deben realizar modificaciones a la base de datos, y las operaciones dentro de ellas deben ser deterministas y repetibles.
- **Rendimiento:** El uso excesivo de funciones, especialmente funciones escalares en cláusulas `SELECT` sobre grandes conjuntos de datos, puede afectar negativamente el rendimiento. Considera usar funciones de tabla en línea cuando sea posible para mejorar el rendimiento.

El uso de variables dentro de funciones es una práctica común para manejar cálculos y lógica temporal que necesita ser encapsulada y reutilizada en múltiples lugares de tu base de datos.

Llamada a Funciones

Veamos cómo crear y ejecutar una función en SQL Server. Para este ejemplo, vamos a crear una función escalar simple que calcula el salario anual de un empleado a partir de su salario mensual. Luego, veremos cómo ejecutar o llamar a esa función desde una consulta SQL.

Creación de una Función Escalar en SQL Server

Primero, crearemos una función llamada `CalculateAnnualSalary`. Esta función toma un parámetro de entrada, que es el salario mensual, y devuelve el salario anual calculado multiplicando el salario mensual por 12.

```
CREATE FUNCTION dbo.CalculateAnnualSalary
(
    @MonthlySalary decimal(10,2)
)
RETURNS decimal(10,2)
AS
BEGIN
    -- Declara una variable para almacenar el salario anual
    DECLARE @AnnualSalary decimal(10,2);

    -- Calcula el salario anual multiplicando el salario mensual por 12
    SET @AnnualSalary = @MonthlySalary * 12;

    -- Devuelve el resultado
    RETURN @AnnualSalary;
END;
GO
```

Ejecución de la Función

Una vez que la función está creada, puedes llamarla de varias maneras en tus consultas SQL. Por ejemplo, puedes usar esta función para calcular el salario anual de cada empleado, suponiendo que tienes una tabla `Employees` que incluye una columna `MonthlySalary`.

Ejemplo 1: Llamada a la función en una consulta `SELECT`

```
-- Suponiendo que tienes una tabla llamada Employees con una columna MonthlySalary
SELECT EmployeeID,
       Name,
       MonthlySalary,
       dbo.CalculateAnnualSalary(MonthlySalary) AS AnnualSalary
FROM Employees;
```

Ejemplo 2: Llamada a la función para un valor específico

También puedes llamar a la función directamente con un valor específico para realizar cálculos rápidos o pruebas.

```
-- Calcular el salario anual para un salario mensual específico
SELECT dbo.CalculateAnnualSalary(5000) AS AnnualSalary;
```

Consejos para el Uso de Funciones

- **Funciones Escalares vs. Funciones de Tabla**: Las funciones escalares devuelven un solo valor y son útiles para cálculos que afectan a una columna. Sin embargo, pueden ser menos eficientes si se llaman en una consulta grande que afecta a muchas filas. Considera usar funciones de tabla cuando necesites devolver conjuntos de filas.
- **Optimización**: Asegúrate de que tus funciones estén bien optimizadas para evitar cuellos de botella en tus consultas, especialmente cuando se usan en bases de datos grandes o en consultas complejas.

Este ejemplo cubre la creación y ejecución de una función escalar simple en SQL Server, proporcionando una herramienta útil para calcular valores basados en la lógica de negocio directamente dentro de tu base de datos.

Uso de las Funciones Tabla

Las funciones de tabla en SQL Server son una herramienta poderosa que te permite definir funciones que devuelven un conjunto de filas, similar a como lo haría una tabla. Estas funciones pueden ser especialmente útiles para encapsular consultas complejas y reutilizarlas en varias partes de tus aplicaciones o consultas.

Tipos de Funciones de Tabla

Hay dos tipos principales de funciones de tabla en SQL Server:

1. **Funciones de Tabla con Valores de Tabla (TVFs - Table-Valued Functions):** Estas pueden ser más específicamente clasificadas como:
   - **Funciones de Tabla de Valor en Línea (Inline Table-Valued Functions):** No tienen un cuerpo de función que use la palabra clave `BEGIN...END` y consisten en una única instrucción `SELECT`.
   - **Funciones de Tabla de Valor con Múltiples Instrucciones (Multi-statement Table-Valued Functions):** Estas funciones incluyen un cuerpo de función que puede tener varias declaraciones SQL, incluyendo la creación de una tabla temporal en memoria para almacenar y manipular datos antes de devolverlos.

Ejemplo de Función de Tabla de Valor en Línea

Estas funciones son generalmente más eficientes y deben ser tu primera opción cuando sea posible usarlas.

```
CREATE FUNCTION dbo.GetEmployeesByDepartment (@DepartmentID int)
RETURNS TABLE
AS
RETURN
(
    SELECT EmployeeID, Name, Position, Salary
    FROM Employees
    WHERE DepartmentID = @DepartmentID
)
GO
```

Ejemplo de Función de Tabla de Valor con Múltiples Instrucciones

Estas funciones son útiles cuando necesitas realizar operaciones complejas antes de devolver los datos.

```
CREATE FUNCTION dbo.GetEmployeeStatistics (@DepartmentID int)
RETURNS @EmployeeStats TABLE
(
    EmployeeID int,
    Name nvarchar(50),
    TotalSales decimal(10,2),
    AverageMonthlySales decimal(10,2)
)
AS
BEGIN
    -- Inserta datos en la tabla de variables
    INSERT INTO @EmployeeStats (EmployeeID, Name, TotalSales, AverageMonthlySales)
    SELECT e.EmployeeID, e.Name, SUM(s.Amount), AVG(s.Amount)
    FROM Employees e
    JOIN Sales s ON e.EmployeeID = s.EmployeeID
    WHERE e.DepartmentID = @DepartmentID
    GROUP BY e.EmployeeID, e.Name;

    -- Devuelve la tabla variable
    RETURN;
END
GO
```

Cómo Usar Funciones de Tabla

Ambos tipos de funciones de tabla se pueden usar de la misma manera en tus consultas SQL:

```
-- Llamada a una función de tabla de valor en línea
SELECT *
FROM dbo.GetEmployeesByDepartment(1);

-- Llamada a una función de tabla de valor con múltiples instrucciones
SELECT *
FROM dbo.GetEmployeeStatistics(1);
```

Consideraciones

- **Rendimiento:** Las funciones de tabla en línea generalmente ofrecen mejor rendimiento porque el optimizador de consultas puede integrar la consulta de la función en la consulta principal.
- **Flexibilidad:** Las funciones de tabla con múltiples instrucciones son más flexibles y permiten lógicas más complejas, como variables temporales y operaciones lógicas.

Utiliza funciones de tabla para estructurar tus bases de datos de forma más eficiente y para hacer tus consultas más organizadas y mantenibles.

Llamada de Funciones desde Python

Para llamar a una función desde una base de datos SQL Server utilizando Python, puedes usar el módulo `pymssql`, que es una interfaz popular para conectar Python con SQL Server. Veamos cómo configurar la conexión y ejecutar una llamada a una función SQL que has definido previamente, como las funciones de tabla que discutimos.

Pre-requisitos

Antes de comenzar, asegúrate de tener `pymssql` instalado en tu entorno de Python. Si no lo tienes instalado, puedes hacerlo utilizando pip:

```
pip install pymssql
```

Ejemplo de Código para Llamar a una Función en SQL Server usando pymssql

Aquí está un ejemplo básico de cómo llamar a una función de SQL Server (por ejemplo, una función que devuelve una tabla) usando `pymssql`:

```
import pymssql

# Configura tus credenciales y detalles de la base de datos
server = 'your_server_address'
user = 'your_username'
password = 'your_password'
database = 'your_database_name'

# Conecta a la base de datos
conn = pymssql.connect(server, user, password, database)
cursor = conn.cursor(as_dict=True)  # Use as_dict=True para obtener los resultados como diccionarios

# Llamada a la función
# Asegúrate de reemplazar 'dbo.GetEmployeesByDepartment' con el nombre de tu función
# y ajustar los parámetros según sea necesario
department_id = 1  # Ejemplo de parámetro
cursor.execute("SELECT * FROM dbo.GetEmployeesByDepartment(%d)", (department_id,))

# Imprime los resultados
for row in cursor:
    print(row)

# Cierra la conexión
cursor.close()
conn.close()
```

Explicación del Código

1. **Importar pymssql:** Incluye el módulo para poder usar sus funciones.
2. **Conectar a la base de datos:** Utiliza `pymssql.connect()` con los detalles de tu servidor para establecer una conexión.
3. **Cursor:** Utiliza el objeto `cursor` para ejecutar consultas. El argumento `as_dict=True` hace que los resultados de las consultas se devuelvan como diccionarios, lo cual es más fácil de manejar en Python.
4. **Ejecutar la Función:** Utiliza `cursor.execute()` para llamar a la función SQL Server. Aquí debes reemplazar `'dbo.GetEmployeesByDepartment'` con el nombre de tu función. Pasa los parámetros necesarios directamente en la consulta.
5. **Manejar los Resultados:** Itera sobre `cursor` para procesar cada fila del resultado.
6. **Cerrar Conexiones:** Cierra el cursor y la conexión para liberar recursos.

Consejos Adicionales

- **Manejo de Errores:** Considera agregar manejo de errores utilizando bloques `try...except` para capturar y manejar excepciones que puedan ocurrir durante la conexión o la ejecución de la consulta.
- **Seguridad:** Evita poner contraseñas directamente en el código. Considera usar variables de entorno u otros métodos seguros para manejar credenciales y configuraciones sensibles.
- **Reutilización de Conexiones:** Para aplicaciones más complejas, considera usar un gestor de contexto (`with` statement) para manejar automáticamente el cierre de conexiones y cursores.

Este ejemplo proporciona una base sobre cómo interactuar con SQL Server usando `pymssql` en Python, permitiéndote realizar llamadas a funciones almacenadas y manejar los datos devueltos de manera eficiente.

Uso de Variables dentro de un Procedimiento Almacenado

Utilizar variables dentro de un procedimiento almacenado en SQL Server sigue la misma lógica que usarlas en scripts de SQL estándar, pero con la ventaja adicional de poder organizar código repetitivo y lógica en un solo lugar, permitiendo la reutilización y mantenimiento más fácil. Veamos cómo definir y usar variables dentro de un procedimiento almacenado.

Crear un Procedimiento Almacenado

Un procedimiento almacenado es esencialmente una colección de declaraciones SQL con un nombre dado que puedes ejecutar repetidamente. Es muy útil para encapsular la lógica de negocio, realizar operaciones repetitivas, y gestionar la seguridad y el acceso a los datos.

Pasos básicos para definir y usar variables:

1. **Definición del procedimiento:** Usa la declaración `CREATE PROCEDURE` para comenzar la definición de tu procedimiento almacenado.

2. **Declarar variables:** Dentro del procedimiento, declara las variables usando la palabra clave `DECLARE`.
3. **Inicializar variables:** Asigna valores iniciales a las variables con la declaración `SET` o `SELECT`.
4. **Usar las variables:** Implementa la lógica del procedimiento utilizando las variables donde sea necesario.
5. **Finalizar el procedimiento:** Termina el procedimiento con las operaciones necesarias, como `SELECT`, `INSERT`, `UPDATE`, `DELETE`, etc., usando las variables declaradas.

Ejemplo de un Procedimiento Almacenado

Supongamos que queremos crear un procedimiento que actualice la información de salario de un empleado basado en su ID y luego devuelva el nuevo salario.

```
CREATE PROCEDURE UpdateEmployeeSalary
    @EmployeeID int,
    @SalaryIncrement decimal(10,2),
    @NewSalary decimal(10,2) OUTPUT
AS
BEGIN
    -- Declara una variable local para almacenar el salario actual
    DECLARE @CurrentSalary decimal(10,2);

    -- Obtiene el salario actual del empleado
    SELECT @CurrentSalary = Salary FROM Employees WHERE EmployeeID = @EmployeeID;

    -- Calcula el nuevo salario
    SET @NewSalary = @CurrentSalary + @SalaryIncrement;

    -- Actualiza el salario del empleado en la base de datos
    UPDATE Employees
    SET Salary = @NewSalary
    WHERE EmployeeID = @EmployeeID;

    -- Retorna el nuevo salario como parámetro de salida
END;
GO
```

Cómo ejecutar el procedimiento almacenado

Para ejecutar el procedimiento almacenado con parámetros y obtener un valor de salida, puedes hacerlo así:

```
DECLARE @MyNewSalary decimal(10,2);

EXEC UpdateEmployeeSalary
    @EmployeeID = 123,
    @SalaryIncrement = 500.00,
    @NewSalary = @MyNewSalary OUTPUT;

SELECT 'El nuevo salario es: ' + CAST(@MyNewSalary AS varchar(20));
```

Consideraciones

- **Documentación:** Documenta tus procedimientos almacenados para que otros desarrolladores entiendan qué hacen y cómo usarlos.
- **Seguridad:** Considera quién tiene permiso para ejecutar o modificar tus procedimientos almacenados.
- **Depuración:** Depura los procedimientos almacenados cuidadosamente para asegurar que funcionan como se espera, especialmente en entornos donde los datos críticos son manipulados.

Usar procedimientos almacenados te permite encapsular y optimizar las operaciones de la base de datos, manteniendo tu código organizado y reutilizable.

Llamada de Procedimientos Almacenados desde Python

Llamar a un procedimiento almacenado desde una base de datos SQL Server utilizando Python con `pymssql` es bastante similar a ejecutar una función, como expliqué anteriormente. La principal diferencia es en la forma en que especificas y ejecutas el procedimiento almacenado, especialmente si tiene parámetros de salida o requiere transacciones.

Ejemplo de Código para Llamar a un Procedimiento Almacenado en SQL Server usando pymssql

A continuación, vamos a ver cómo ejecutar un procedimiento almacenado con y sin parámetros. Primero, consideremos un procedimiento almacenado simple sin parámetros de salida.

Procedimiento Almacenado Sin Parámetros de Salida

Supongamos que tienes un procedimiento almacenado llamado `UpdateEmployeeSalaries` que actualiza los salarios en la base de datos y no necesita ningún parámetro de entrada ni devuelve parámetros.

```
import pymssql

# Configuración de conexión
server = 'your_server_address'
user = 'your_username'
password = 'your_password'
database = 'your_database_name'

# Establece la conexión
conn = pymssql.connect(server, user, password, database)
cursor = conn.cursor()

# Llamar al procedimiento almacenado
cursor.callproc('UpdateEmployeeSalaries')
conn.commit()  # Importante para asegurarse de que los cambios sean guardados si el SP hace cambios en la BD
```

```
# Cierra el cursor y la conexión
cursor.close()
conn.close()
```

Procedimiento Almacenado con Parámetros

Ahora, si tu procedimiento almacenado requiere parámetros, por ejemplo, un procedimiento `IncreaseSalary` que aumenta el salario en un porcentaje específico y tiene un parámetro de entrada y uno de salida, veamos cómo hacerlo:

```
import pymssql

# Configuración de conexión
server = 'your_server_address'
user = 'your_username'
password = 'your_password'
database = 'your_database_name'

# Establece la conexión
conn = pymssql.connect(server, user, password, database)
cursor = conn.cursor()

# Preparar los parámetros para el procedimiento almacenado
employee_id = 123
salary_increment = 5.0  # Incremento del salario en porcentaje
new_salary = 0  # Variable para almacenar el nuevo salario

# Llamar al procedimiento almacenado
cursor.execute('EXEC IncreaseSalary @EmployeeID=%d, @SalaryIncrement=%f, @NewSalary=%d OUTPUT', (employee_id, salary_increment, new_salary))

# Obtener el parámetro de salida (si es necesario)
new_salary = cursor.fetchone()

# Confirmar los cambios si el procedimiento realiza modificaciones en la base de datos
conn.commit()

# Imprimir el nuevo salario
print('Nuevo salario:', new_salary)

# Cierra el cursor y la conexión
cursor.close()
conn.close()
```

Notas Importantes

- **Commit de la Transacción:** Si el procedimiento realiza cambios en la base de datos (como inserciones, actualizaciones o eliminaciones), debes llamar a `conn.commit()` para asegurar que los cambios se guarden.
- **Manejo de Errores:** Es bueno envolver el código en un bloque `try...except` para manejar posibles errores en la conexión o la ejecución del procedimiento almacenado.
- **Cerrar Recursos:** Asegúrate de cerrar siempre el cursor y la conexión para liberar recursos, idealmente utilizando un bloque `with` o asegurándote de llamar a `close` en un bloque `finally`.

Este método te permitirá integrar procedimientos almacenados de SQL Server en tus aplicaciones Python de manera eficaz, manteniendo el código organizado y manejable.