CAPÍTULO IV: DESARROLLO DEL TEMA

Título: Manejo y Optimización de Datos Semi-Estructurados (JSON) en SQL Server

1. ¿Qué es el Almacenamiento JSON en SQL Server?

En nuestro Sistema de Gestión Hospitalario (SGH), no toda la información es estructurada. Mientras que un Paciente siempre tiene un DNI y un nombre (datos estructurados), sus "alergias", "preferencias de contacto" o "historial de síntomas" pueden ser cero, uno o veinte (datos semi-estructurados).

El almacenamiento JSON en SQL Server no es un tipo de dato propio, sino un conjunto de funcionalidades sobre el tipo NVARCHAR(MAX). Se utiliza para guardar estos datos "semi-estructurados" dentro de nuestro modelo relacional.

- Implementación: Se añade una columna (ej. datos_adicionales) a una tabla existente (ej. Paciente).

```
IF NOT EXISTS (SELECT 1 FROM sys.columns WHERE Name = N'datos_adicionales' AND Object_ID = Object_ID(N'paciente'))
BEGIN
    ALTER TABLE paciente
    ADD datos_adicionales NVARCHAR(MAX);
    PRINT 'Columna [datos_adicionales] agregada.';
END
GO
```

- Integridad: Para asegurar que solo se guarde texto con formato JSON válido, se utiliza la restricción CHECK (ISJSON(datos_adicionales) > 0).

```
IF NOT EXISTS (SELECT 1 FROM sys.check_constraints WHERE name = 'CHK_Paciente_DatosAdicionales_IsJSON')
BEGIN
    ALTER TABLE paciente
    ADD CONSTRAINT CHK_Paciente_DatosAdicionales_IsJSON
    CHECK (ISJSON(datos_adicionales) > 0);
    PRINT 'Restriccion CHK_Paciente_DatosAdicionales_IsJSON agregada.';
END
GO
```

- Finalidad: Se usa para añadir flexibilidad al modelo sin tener que crear múltiples tablas (ej. Paciente_Alergias, Paciente_Contactos, Paciente_Preferencias).

2. ¿Qué son las Funciones de Consulta JSON?

Son rutinas nativas de SQL Server que nos permiten leer, extraer y modificar datos guardados *dentro* de un texto JSON. Las principales para la consulta son:

- Funciones Escalares (JSON_VALUE): Recibe un texto JSON y una "ruta" (path), y devuelve un solo valor (un string, un número, una fecha). Es ideal para extraer un dato puntual.
- Funciones de Tabla (OPENJSON): Recibe un texto JSON y una "ruta", y devuelve un conjunto de filas (una tabla virtual). Es ideal para

"descomponer" un array o un objeto en filas que se pueden cruzar con otras tablas.

3. Diferencias Clave entre JSON_VALUE y OPENJSON

Aspecto	JSON_VALUE	OPENJSON
Devuelve valor	Obligatorio: Un solo valor escalar (INT, VARCHAR, etc.).	Obligatorio: Una tabla (conjunto de filas y columnas).
Uso en Consulta	Directamente en SELECT o WHERE (ej. WHERE JSON_VALUE(...) = 'O+').	En la cláusula FROM, usualmente con CROSS APPLY (ej. FROM Paciente CROSS APPLY OPENJSON(...)).
Enfoque Típico	Extraer un dato simple y conocido de un objeto (ej. "tipo de sangre", "email de contacto").	"Destruir" (shred) un array o un objeto complejo en filas separadas para hacer JOINs.

1. Funciones JSON en Operaciones CRUD

INSERT: Se realiza un INSERT normal, pasando el JSON como un string de texto:

```
INSERT INTO paciente (nombre, apellido, dni, direccion, telefono, fecha_nacimiento, datos_adicionales)
VALUES ('Fernando', 'Arce', 45644949, 'Calle Falsa 123', 3794999999, '2000-01-01',
        '{"alergia": "polen", "tipo_sangre": "A-"}');
GO
```

UPDATE (Modificar): La función clave es JSON_MODIFY(). Permite añadir, actualizar o eliminar claves.

```
UPDATE paciente
SET datos_adicionales = JSON_MODIFY(datos_adicionales, '$.contacto_emergencia', 'Ana')
WHERE dni = 45644949; -- Fernando Arce
GO
```

DELETE (Borrar datos *dentro* del JSON): También se usa JSON_MODIFY() poniendo el valor en NULL.

```
UPDATE paciente
SET datos_adicionales = JSON_MODIFY(datos_adicionales, '$.tipo_sangre', NULL)
WHERE dni = 45644949;
GO
```

- **SELECT (Consultar): Se usan JSON_VALUE y OPENJSON como se vio en el punto 2.**

```
PRINT 'Pacientes con tipo de sangre A-';
SELECT nombre, apellido, JSON_VALUE(datos_adicionales, '$.tipo_sangre') AS TipoSangre
FROM paciente
WHERE JSON_VALUE(datos_adicionales, '$.tipo_sangre') = 'A-';
GO
```

```
SELECT p.nombre, p.apellido, j.nombre AS ContactoNombre, j.telefono AS ContactoTelefono
FROM paciente p
    CROSS APPLY OPENJSON(p.datos_adicionales, '$.contacto_emergencia')
        WITH (
            nombre VARCHAR(50) '$.nombre',
            telefono VARCHAR(20) '$.telefono'
        ) AS j;
GO
```

5. Ventajas y Desventajas (Especialmente Rendimiento)

- **Ventajas:**
  - Flexibilidad de Esquema: Permite añadir nuevos datos (ej. "redes sociales" del paciente) sin un ALTER TABLE, facilitando el mantenimiento y la evolución del sistema.
  - Consolidación: Evita la "explosión" de tablas satélite para datos simples (ej. Paciente_Telefonos, Paciente_Emails, Paciente_Alergias).
- **Desventajas / Cuidados (Rendimiento):**
  - El Problema de la "Caja Negra": De forma similar a cómo una función escalar definida por el usuario puede ser costosa, usar JSON_VALUE en una cláusula WHERE sobre millones de filas es extremadamente costoso.
  - El motor no puede "ver" dentro del texto JSON, por lo que no puede usar un índice estándar.
  - Esto obliga al motor a ejecutarse "fila por fila" (Row Mode), leer el JSON de *cada* paciente, analizarlo (parsearlo) y luego ver si cumple la condición. Esto genera un alto costo de CPU y lecturas lógicas.

6. Comparación Eficiencia: "JSON Directo vs. JSON Indexado"

Para cumplir con el objetivo de evaluar la eficiencia, se plantea una metodología de pruebas comparativas. Se creará una tabla con una carga masiva de datos (ej. 1 millón de registros) y se comparará el rendimiento de una consulta antes y después de una optimización.

Consulta Directa (Sin Optimizar)

Este escenario representa la implementación "ingenua", donde solo se consulta el JSON directamente.

- **Consulta de Prueba:**

```
SELECT id_paciente, nombre, apellido, dni
FROM paciente
WHERE JSON_VALUE(datos_adicionales, '$.alergia') = 'penicilina';
GO
```

Resultados:

(1002 filas afectadas)

Table 'paciente'. Scan count 1, logical reads 188, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 35 ms.

Consulta Optimizada (Con Índice en Columna Computada)

Este escenario implementa la optimización recomendada para JSON en SQL Server.

```
PRINT '--- 5.2. CREANDO OPTIMIZACION (Columna Computada + INDICE CON INCLUDE) ---';
IF COL_LENGTH('paciente', 'alergia_computada') IS NULL
BEGIN
    ALTER TABLE paciente
    ADD alergia_computada AS JSON_VALUE(datos_adicionales, '$.alergia');
    PRINT 'Columna [alergia_computada] agregada.';
END
GO
IF NOT EXISTS (SELECT 1 FROM sys.indexes WHERE name = 'IDX_paciente_alergia_json_COV')
BEGIN
    --
    CREATE INDEX IDX_paciente_alergia_json_COV
    ON paciente(alergia_computada)
    INCLUDE (nombre, apellido, dni);

    PRINT 'Indice de Cobertura [IDX_paciente_alergia_json_COV] creado.';
END
GO
```

Resultados:

(1002 filas afectadas)

Table 'paciente'. Scan count 1, logical reads 10, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 16 ms.

Conclusiones

Al evaluar la eficiencia del manejo de datos JSON en SQL Server, el objetivo era determinar la estrategia de optimización correcta para su uso en un Sistema de Gestión Hospitalario (SGH) de alto rendimiento.

Escenario 1: Consulta Directa (Sin Optimización)

- **Resultado:** Table Scan con **188 lecturas lógicas**.
- **Análisis:** Tal como se esperaba, la función JSON_VALUE en el WHERE actúa como una "caja negra". El motor se ve forzado a escanear la tabla entera (ir "casa por casa" en la analogía de *Memento*) y aplicar la función 10.000 veces, resultando en un costo alto (188 lecturas) que no escala.

2. Escenario 2: Optimización Real (Índice de Cobertura)

- **Resultado:** Index Seek (Puro) con **10 lecturas lógicas**.
- **Análisis:** La solución fue crear un "Índice de Cobertura" (Covering Index). Esto se logró creando una columna computada (alergia_computada) y luego creando un índice sobre ella que, crucialmente, **INCLUYE** las otras columnas pedidas en el SELECT (nombre, apellido, dni).
- **La Solución:** Al incluir las columnas del SELECT en el propio índice, la "guía telefónica" ya tenía toda la información. El motor pudo encontrar los datos y devolverlos leyendo *únicamente* el índice (Index Seek), sin tocar la tabla principal. El costo se desplomó a solo **10 lecturas lógicas**.

Conclusión Final

El uso de JSON en SQL Server es una herramienta poderosa para dar flexibilidad a nuestro SGH. Sin embargo, su implementación "ingenua" (sin optimización) es ineficiente y no es viable para grandes volúmenes de datos.

La comparativa de **188 lecturas lógicas** (Table Scan) contra **10 lecturas lógicas** (Index Seek) demuestra que la estrategia correcta es un **diseño híbrido**:

1. Mantener el **Modelo Relacional** (Creacion de la base de datos.sql para datos fijos y conocidos.
2. Diseñar **Columnas JSON** (ej. datos_adicionales) para datos variables (alergias, historial, contactos).
3. Implementar **Índices de Cobertura No Agrupados** (con INCLUDE) sobre **Columnas Computadas**. Estos índices deben ser diseñados *específicamente* para "cubrir" las consultas analíticas y filtros más frecuentes.