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).
- Integridad: Para asegurar que solo se guarde texto con formato JSON válido, se utiliza la restricción CHECK (ISJSON(datos_adicionales) > 0).
- 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, datos_adicionales)
 VALUES ('Fer', 'Arce', 46074586, '{"alergia": "penicilina", "tipo_sangre": "O+"}');
```

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

```
 UPDATE Paciente
SET datos_adicionales = JSON_MODIFY(datos_adicionales, '$.alergia', 'polvo')
WHERE dni = 46074586;

-- Añadir un nuevo valor
UPDATE Paciente
SET datos_adicionales = JSON_MODIFY(datos_adicionales, '$.contacto_emergencia', 'Ana')
WHERE dni = 46074586;
```

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

```
-- Borrar la clave "alergia"
UPDATE Paciente
SET datos_adicionales = JSON_MODIFY(datos_adicionales, '$.alergia', NULL)
WHERE dni = 46074586;
```

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

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:**

```
-- Buscar todos los pacientes con una alergia específica
SELECT id, nombre
FROM Pacientes_Test
WHERE JSON_VALUE(datos_adicionales, '$.alergia') = 'penicilina';
```

Resultados:

```
Tiempos de SQL Server:
(Tiempos de SQL Server: Tiempo de CPU = 8150 ms, tiempo transcurrido = 9320 ms.)

Estadísticas de E/S:
Tabla 'Pacientes_Test'. Recuentos de examen 1, lecturas lógicas 245810, lecturas físicas 0, lecturas
anticipadas 0, ...
```

Consulta Optimizada (Con Índice en Columna Computada)

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

- **Crear la optimización (Índice):**

```
-- 1. Se crea una columna "virtual" que expone el dato del JSON
ALTER TABLE Pacientes_Test
ADD alergia_computada AS JSON_VALUE(datos_adicionales, '$.alergia');

-- 2. Se crea un índice estándar sobre esa nueva columna "virtual"
CREATE INDEX IDX_paciente_alergia
ON Pacientes_Test(alergia_computada);
```

Resultados:

```
Tiempos de SQL Server:
(Tiempos de SQL Server: Tiempo de CPU = 85 ms, tiempo transcurrido = 102 ms.)

Estadísticas de E/S:
Tabla 'Pacientes_Test'. Recuentos de examen 1, lecturas lógicas 148, lecturas físicas 0, lecturas
anticipadas 0, ...
```

Conclusiones

Al evaluar la eficiencia del manejo de JSON, se observa una diferencia drástica entre las dos implementaciones.

1. La **consulta directa (6.1)** trata la función JSON_VALUE como una "caja negra". El optimizador no puede usar índices sobre el contenido del JSON, forzando un **"Table Scan"** que resulta en miles de lecturas lógicas y un alto tiempo de CPU. El rendimiento se degrada linealmente con el tamaño de la tabla.
2. La **consulta optimizada (6.2)**, al utilizar un **Índice sobre una Columna Computada**, resuelve este problema. La columna computada expone el dato del JSON de una forma que el motor *sí* puede indexar. El optimizador es capaz de cambiar un "Table Scan" por un **"Index Seek"**, reduciendo las lecturas lógicas y el tiempo de CPU en más de un 98%.
   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 estrategia correcta es un **diseño híbrido**:

- Mantener el **Modelo Relacional** para datos fijos y conocidos.
- Diseñar **Columnas JSON** para datos variables (alergias, historial, contactos).

- Implementar **Índices No Agrupados sobre Columnas Computadas** para optimizar las consultas analíticas y filtros más frecuentes sobre esos datos JSON.