Mejorar el rendimiento de QUERIES en SQL Server

Uso de sintaxis UNION

Debemos tener en cuenta que por defecto un UNION equivale a realizar un SELECT DISTINCT sobre el resultado final de una query. En otras palabras, UNION toma los resultados de dos recordsets, los combina y realiza un SELECT DISTINCT de cara a eliminar las filas duplicadas. Este proceso ocurre incluso si no hay registros duplicados en el recordset final. Si sabemos que hay registros duplicados, y esto representa un problema para la aplicación, entonces haremos uso de UNION para eliminar estas filas duplicadas.

Por otro lado si sabemos que nunca habrá duplicado de filas o si las hay pero no representa un problema para la aplicación entonces deberemos usar UNION ALL en lugar de UNION. La ventaja de UNION ALL es que no realiza el SELECT DISTINCT, lo cual evita gran cantidad de trabajo y recursos al servidor SQL.

Mejorar rendimiento de UNION

Otro caso bastante común es el que se explica en el siguiente ejemplo, imaginemos que queremos realizar una query para mezclar dos conjuntos de datos:

```
SELECT column_name1, column_name2
FROM table_name1
WHERE column_name1 = some_value
UNION
SELECT column_name1, column_name2
FROM table_name1
WHERE column_name2 = some_value
```

La misma query puede ser reescrita como se explica a continuación para mejorar el rendimiento de la misma:

```
SELECT DISTINCT column_name1, column_name2
FROM table_name1
WHERE column_name1 = some_value OR column_name2 = some_value
```

Y puede mejorarse aún más si sabemos que la mezcla de estos dos grupos de datos a pesar de contener duplicados no afectan al funcionamiento de la aplicación eliminando el DISTINCT.

Evaluar el uso de DISTINCT

Un aspecto relativo al rendimiento es la evaluación del uso de la sentencia DISTINCT. Muchos desarrolladores aplican esta sentencia por defecto, aunque no se necesite, Sólo debe usarse si sabemos que la query puede devolver duplicados, y además esto puede provocar un mal funcionamiento de la aplicación que hace uso de los datos.

La sentencia DISTINCT genera una gran cantidad de trabajo extra a SQL Server debido a que consume muchos recursos que son necesarios para otras queries que sean lanzadas dentro de la base de datos. Sólo se debe usar si es necesario.

Devolver los datos que se necesitan

Un aspecto que siempre se menciona en todos los libros de SQL es que debemos devolver nada más que los datos que se necesitan, y esto sobre todo referido a las columnas de datos. Debemos evitar el uso de SELECT * ya que esto además de devolver más datos de los que seguramente necesitemos, impide el uso de índices, añadiendo mayor degradación al rendimiento del sistema.

Uso de TOP

Si nuestra aplicación permite a los usuarios ejecutar queries, pero es difícil evitar en la aplicación de forma sencila el retorno de cientos de registros o incluso miles es posible usar el operador TOP con una sentencia SELECT. De este modo se puede limitar cuantas filas son devueltas incluso si el usuario no introduce ningún criterio que ayude a reducir el número de filas a devolver al cliente. Por ejemplo:

```
SELECT TOP 100 fname, lname FROM customers
WHERE state = 'mo'
```

Esta query limita los resultados a las 100 primeras filas, incluso si el criterio del WHERE de devuelve 10000 registros. Cuando el número especificado se alcanza, todos los procesos de la query se paran salvaguardando así recursos del SQL.

El operador TOP también permite seleccionar porcentajes, tal y como indica el siguiente ejemplo:

```
SELECT TOP 10 PERCENT fname, lname FROM customers
WHERE state = 'mo'
```

Clausulas WHERE NON-SARGABLE

Como norma hay que evitar las cláusulas WHERE NON-SARGABLE, (esta palabra proviene de SARG que significa Search Argument y se refiere a una cláusula WHERE que compara una columna con una constante).

En estos casos cuando una cláusula es SARGABLE puede aprovecharse la funcionalidad de los INDICES (asumiendo que al menos hay uno) para acelerar la ejecución de la Query. En aquellos casos que la cláusula WHERE sea NON-SARGABLE, significará que dicha cláusula WHERE (o parte de ella) no puede aprovecharse de los INDICES, dando como consecuencia un table/scan y degradando el rendimiento de la Query.

Los argumentos de búsqueda en la cláusula WHERE, tales como “IS NULL”, “<>”, “!=”, “!>”,”!<”, “NOT”, “NOT EXIST”, “NOT IN”, “NOT LIKE” y “LIKE ‘%500’” generalmente previenen (pero no siempre) la utilización de un INDICE para realizar la búsqueda.

Adicionalmente las expresiones que incluyen una función sobre una columna, expresiones que tienen la misma columna en ambos lados de un operador o comparaciones contra una columna (y no una constante) son NO-SARGABLE.

Pero no todas cláusulas WHERE que tienen una expresión NON-SARGABLE provocan un TABLE/INDEX SCAN. Si la cláusula WHERE incluye cláusulas NON-SARGABLE y SARGABLE, entonces las cláusulas SARGABLE pueden usar un INDICE (si existe uno) para ayudar a acceder a los datos rápidamente.

En muchos casos si hay un COVERED INDEX en la tabla, que incluye todas las columnas en la SELECT, JOIN y WHERE entonces dicho INDICE puede ser utilizado en lugar de un TABLE/INDEX SCAN para devolver los datos, incluso si tiene un NON-SARGABLE en la cláusula WHERE.

Pero se debe tener en cuenta que los COVERED INDEX tienen sus propias desventajas, tales como producir INDICES muy extensos que pueden incrementar los I/O cuando son leídos. En algunos casos, es posible reescribir una cláusula WHERE NON-SARGABLE, por ejemplo:

```
WHERE substring(firstname,1,1) = ‘m’
```

Por

```
WHERE firstname LIKE ’m%’
```

Ambas cláusulas WHERE producen el mismo resultado, pero la primera es NON-SARGABLE (usa una función) y se ejecutará más lenta, mientras que la segunda es SARGABLE e irá más rápida.

Las cláusulas WHERE que realizan alguna función sobre una columna son NON-SARGABLE. Por otro lado si se puede reescribir la cláusula WHERE dando como resultado que la columna y la función estén separadas, entonces la QUERY puede usar un INDICE que esté disponible, aumentando así el rendimiento por ejemplo:

La función actúa directamente sobre la columna y el INDICE no puede usarse:

```
SELECT member_number, first_name, last_name
FROM members
WHERE DATEDIFF(yy,datofbirth,GETDATE()) > 21
```

La función ha sido separada de la columna y puede usar un INDICE:

```
SELECT member_number, first_name, last_name
FROM members
WHERE dateofbirth < DATEADD(yy,-21,GETDATE())
```

Cada una de las QUERIES anteriores produce el mismo resultado, pero la segunda QUERY usará un INDICE porque la función no se realiza directamente sobre la columna, como si ocurre en el primer ejemplo. El mensaje de esta historia es que se debe intentar reescribir las cláusulas WHERE que tengan funciones que actúen sobre columnas por cláusulas que no lo hagan.

Las cláusulas WHERE que usen el operador NOT son todas NON-SARGABLE, pero pueden ser reescritas para eliminar dicho operador, por ejemplo:

```
WHERE NOT column_name > 5
```

A

```
WHERE column_name <= 5
```

Es fácil identificar si una cláusula es NON-SARGABLE, se ejecuta en el Query Analizer mostrando el plan de ejecución para ver si se realiza un INDEX LOOKUP o un TABLE/INDEX SCAN.

Funciones en cláusulas WHERE NON-SARGABLE

Cuando nos encontramos con cláusulas WHERE NON-SARGABLE que contienen una función en el lado derecho de un signo igual, y no es posible reescribir dicha cláusula para que sea SARGABLE, existe la opción de crear un INDICE en una COLUMNA COMPUTADA. De este modo se evitará el WHERE NON-SARGABLE usando los resultados de la función la dicha cláusula.

Debido a la sobrecarga que requiere para los INDICES en COLUMNAS COMPUTADAS, sólo realizaremos esto si la QUERY la ejecutaremos una y otra vez desde la aplicación.

Uso de NOT IN

En el caso que tengamos que hacer uso del comando NOT IN tendremos que tener especial cuidado en su uso ya que posee un mal rendimiento ya que obliga al SQL Server Optmizer a realizar un SCAN, en su lugar mejor utilizaremos las siguientes opciones ordenadas de mejor a peor rendimiento:

- Usar EXISTS o NOT EXISTS
- Usar IN
- Realizar un LEFT OUTER JOIN y chequear por una condición NULL

Cuando exista la posibilidad de elegir entre IN o EXISTS utilizaremos siempre EXISTS, ya que tiene mejor rendimiento.

Forzado de INDICES

Es posible que podamos encontrarnos TABLE SCAN en las queries que lancemos, a pesar de existir INDICES que mejorarían el rendimiento. En estos casos la mejor opción para forzar el uso de un INDICE es realizar los siguiente, como muestra el ejemplo:

```
SELECT * FROM tblTaskProcesses WHERE nextprocess = 1 AND processid IN (8,32,45)
```

Esto tarda 3 segundos y al siguiente QUERY tarda menos de un segundo:

```
SELECT * FROM tblTaskProcesses (INDEX = IX_ProcessID) WHERE nextprocess = 1 AND processid IN (8,32,45)
```

Mejores prácticas en el uso de LIKE

En el caso del comando LIKE es necesario entender que hay maneras más optimas en el uso de dicho comando. En el caso que nos trata del comando LIKE debemos en la medida de los posible usar una "leading character" esto es un carácter diferente de un "wildcard" (%,*, etc…).

Por ejemplo la query LIKE '%m' tiene peor rendimiento que LIKE 'm%' ya que en el segundo caso (más optimo rendimiento) esta cláusula puede usar en el caso que existiera un INDICE para agilizar la búsqueda por el carácter. En el caso de que el "leading character" fuera un "wildcard" SQL tendría que realizar un SCAN previo.

Sumarios de datos

A veces nuestras aplicaciones necesitan calcular sumarios de datos, con el consecuente coste de rendimiento. Estos cálculos al vuelo para mantener un sumario son muy costosos, y en muchos casos la mejor opción es utilizar un trigger que una vez ejecutada la transacción haga el sumario de datos en una tabla auxiliar que se pueda consultar en cualquier momento.

A veces el coste de este trigger puede tener mejor rendimiento que calcular el sumario junto con la transacción lanzada.

Inserción de datos binarios de gran tamaño

Si nuestra aplicación necesita inserta insertar datos binarios de gran tamaño en una columna de datos, se debe realizar en primer lugar a través de un Store Procedure y no usar nunca una sentencia INSERT dentro de nuestra aplicación.

LA razón es que la aplicación debe primero convertir los datos binarios en una cadena de caracteres (lo que hace doblar su tamaño incrementando el tráfico de red y llevando más tiempo) antes de que pueda ser enviada al servidor. Y cuando el servidor recibe la cadena de caracteres, tiene que convertirla de nuevo a datos binarios (llevándose aún más tiempo que en la primera conversión).

El uso de Store Procedures evita todo esto ya que la actividad ocurre en el servidor SQL Server y los datos transmitidos a través de la red son menores.

Cuando usar IN o BETWEEN

En el caso que nuestras QUERIES tenga que hacer uso de los comandos IN o BETWEEN haremos siempre uso de BETWEEN siempre que se pueda (existen casos que no es posible). Para entender porque veamos el siguiente ejemplo:

```
SELECT customer_number, customer_name
FROM customer
WHERE customer_number in (1000, 1001, 1002, 1003, 1004)
```

Es menos eficiente que la siguiente QUERY:

```
SELECT customer_number, customer_name
FROM customer
WHERE customer_number BETWEEN 1000 and 1004
```

Asumiendo que existe un INDICE en customer_number el Query Optimizer de SQL Server puede localizar un rango de números más rápidamente mediante el uso de BETWEEN que con el uso de IN.

Uso de SUBSTRING en cláusulas WHERE

Si es posible debemos evitar el uso de la función SUBSTRING en las cláusulas WHERE. Dependiendo de cómo se construya la función SUBSTRING puede forzar un SCAN de TABLA en lugar de permitir al optimizador de SQL a usar un INDICE (asumiendo que exista uno). Si la subcadena que estamos buscando no incluye el primer carácter de la columna por la cual buscamos entonces se realiza un SCAN. Veamos un ejemplo:

```
WHERE SUBSTRING(column_name,1,1) = 'b'
```

Intentaremos usar en su lugar lo siguiente:

```
WHERE column_name LIKE 'b%'
```

Si nos decidimos a usar esta opción debemos considerar usar una condición para el LIKE que sea SARGABLE , esto significa que no podemos por un “wirldcard” en primer lugar.

Concatenación ANDs

Si existe una cláusula WHERE que incluye expresiones conectadas por dos o más operadores AND, SQL Server evaluará desde la izquierda hacia la derecha en el orden que hayan sido escritas. Esto asume que no se hayan usado paréntesis para cambiar el orden de la ejecución. Por esta razón se debe considerar lo siguiente cuando usemos el operador AND:

- Localizaremos la expresión menos probable de suceder y la pondremos en primer lugar de la expresión AND. De este modo si una expresión AND es falsa la cláusula finalizará inmediatamente ahorrando tiempo

- Si ambas partes de una expresión AND son iguales o tienen igual peso, y son falsas, pondremos la menos compleja primero. De este modo si es falsa la expresión se realizará menos trabajo para evaluar la expresión.

Como mejorar el rendimiento de las QUERIES con operadores AND

Si lo que queremos es mejorar el rendimiento de las QUERIES que contienen operadores AND en la cláusula WHERE debemos considerar lo siguiente:

- En las condiciones que se establezcan en la cláusula WHERE al menos una de ellas debería basarse en una columna lo más selectiva posible y que sea parte de un INDICE.
- Si al menos uno de los criterios de búsqueda en la cláusula WHERE no es altamente selectivo, consideraremos añadir INDICES para todas las columnas referenciadas en la cláusula WHERE.

Si ninguna de las columnas en la cláusula WHERE son suficientemente selectivas para usar su propio INDICE, consideraremos crear un "Covering Index" para esta QUERY.

Uso de operador OR

Es posible que las QUERIES que lancemos nos encontremos que existen operadores OR que muchas veces pueden reescribirse mediante la sentencia UNION ALL, de cara a mejorar el rendimiento de la QUERY. Por ejemplo echemos un vistazo a la siguiente QUERY:

```
SELECT employeeID, firstname, lastname
FROM names
WHERE dept = 'prod' or city = 'Orlando' or division = 'food'
```

Esta QUERY tiene tres condiciones en la cláusula WHERE separadas. De cara a que esta QUERY use un INDICE, debemos tener un INDICE sobre todas las columnas que están en dicha cláusula.

Este ejemplo puede ser reescrito usando un UNION ALL en lugar de un OR, tal y como muestra el ejemplo:

```
SELECT employeeID, firstname, lastname FROM names WHERE dept = 'prod'
UNION ALL
SELECT employeeID, firstname, lastname FROM names WHERE city = 'Orlando'
UNION ALL
SELECT employeeID, firstname, lastname FROM names WHERE division = 'food'
```

Cada una de estas QUERIES producirá el mismo resultado. Si hay sólo un INDICE pero ninguna otra columna está en la cláusula WHERE entonces la primera versión no usará ningún INDICE y se realizará un TABLE SCAN.

Pero en la segunda versión de la QUERY usaremos el INDICE para parte de la QUERY pero no para toda la QUERY.

Admitimos que esto es un ejemplo muy sencillo, pero muestra cómo podemos mejorar la QUERY reescribiéndola, aunque la diferencia de rendimiento sería casi inapreciable, sin embargo, con QUERIES más complejas las diferencias de rendimiento se hacen más notorias.

Debemos tener en cuenta que estamos usando UNION ALL en lugar de UNION. La razón es que el uso de UNION previene de realizar ordenaciones y evitar repeticiones, lo cual mejora el rendimiento. Naturalmente existe la posibilidad de duplicados, por lo que en esos casos si usaremos UNION siempre y cuando no haya otra opción tal y como explicamos en el punto Mejorar el rendimiento de UNION.

Uso de ORDER BY

Usaremos ORDER BY en las QUERIES que lancemos sólo si es absolutamente indispensable, es decir, que si es posible realizar la ordenación en el lado del cliente siempre será mejor que realizarla desde el lado del servidor SQL Server.

En el caso que sea absolutamente necesario realizar la ordenación en el lado del servidor SQL Server, deberemos atender a las siguientes recomendaciones:

- Mantener el número de filas a ordenar al mínimo, haciendo esto mediante la devolución de aquellas filas que son absolutamente necesarias.
- Mantener el número de columnas a ordenar al mínimo. En otras palabras, no ordenar columnas no requeridas.
- Mantener el ancho (tamaño físico) de las columnas a ordenar al mínimo
- Ordenar columnas con tipos de datos numéricos en lugar de tipos de datos carácter

Cuando usemos cualquier mecanismo de ordenación en Transact –SQL, debemos tener en mente todas estas recomendaciones para la mejora del rendimiento.

Si se ha de ordenar por una columna a menudo, debemos considerar el realizar un “Clustered Index” sobre esa columna para la mejora del rendimiento.

Uso del operador IN en cláusulas WHERE

Es posible que nos podamos encontrar el operador IN en cláusulas WHERE (y que este sea imposible sustituirlo por otro operador). En estos casos lo mejor que podemos hacer es lo siguiente; si hemos de contrastar una condición contra unos valores en un IN, debemos poner en la parte izquierda de dichos valores los que sean más frecuentes de encontrar, y al final de la lista los menos frecuentes.

Uso de SELECT INTO

Cuando nos encontremos con la necesidad de utilizar SELECT INTO debemos tener en cuenta que este tipo de sentencias pueden provocar un bloqueo en las tablas de sistema, impidiendo a otros usuarios el acceso a los datos que necesiten. Si se necesita usar SELECT INTO, deberemos intentar programar su uso cuando el usuario esté menos ocupado, y teniendo en cuenta que la cantidad de datos insertados sea la mínima posible.

Uso de la sentencia HAVING

Si la sentencia SELECT contiene una cláusula HAVING, debemos escribir la QUERY de forma que la cláusula WHERE realice la mayor parte del trabajo (eliminando filas no deseadas) en lugar de hacer que la cláusula HAVING haga el trabajo de eliminar dichas filas. Usando la cláusula WHERE apropiadamente podemos eliminar filas innecesarias antes de lanzar el GROUP BY y el HAVING evitando así trabajo extra y mejorando así el rendimiento.

Por ejemplo, en una SELECT con cláusulas WHERE, GROUP BY y HAVING ocurre lo siguiente:

En primer lugar, la cláusula WHERE es usada para seleccionar las filas apropiadas que necesitan ser agrupadas. Lo próximo es agrupar mediante GROUP BY que divide las filas en grupos y agrega sus valores. Por último, la cláusula HAVING elimina los grupos no deseados.

Si la cláusula WHERE es utilizada para eliminar el mayor número posible de filas no deseadas implicará que la sentencia GROUP BY y HAVING tendrán menos trabajo que hacer y mejorará el rendimiento.

No usar el comando GROUP BY sin una función de agregación

La cláusula GROUP BY puede usarse con o sin una función de agregación. Pero si queremos obtener un mejor rendimiento, no usaremos la cláusula GROUP BY sin una función de agregación. Esto es porque produce el mismo resultado usar DISTINCT y es más rápido. Veamos un ejemplo:

```
USE Northwind
SELECT OrderID
FROM [Order Details]
WHERE UnitPrice > 10
GROUP BY OrderID
```

O

```
USE Northwind
SELECT DISTINCT OrderID
FROM [Order Details]
WHERE UnitPrice > 10
```

Ambas QUERIES dan el mismo resultado, pero la segunda obtendrá mejor rendimiento, ya que usa menos recursos.

¿Cómo acelerar la cláusula GROUP BY?

Para acelerar el uso de la cláusula GROUP BY debemos seguir las siguientes recomendaciones:

- Mantener el número de filas a devolver por la QUERY tan pequeño como sea posible
- Mantener el número de agrupaciones tan limitado como sea posible
- No agrupar columnas redundantes
- Si hay un JOIN en la misma SELECT que tiene un GROUP BY, intentar reescribir la QUERY usado una SUBQUERY en lugar de usar un JOIN. Si es posible hacer esto, el rendimiento será mejor. Si se tiene que usar un JOIN, intentaremos hacer el GROUP BY por columna desde la misma tabla que la columna o columnas sobre la cual la se usa la función.

Consideraremos el añadir un ORDER BY a la SELECT que ordena por la misma columna que el GROUP BY. Eso puede producir que el GROUP BY tenga mejor rendimiento.

La percepción “mejora” el rendimiento

A veces la percepción que tenemos del rendimiento de las QUERIES es sólo eso, una percepción, ya que en muchas ocasiones los resultados a devolver son tan extensos, que las QUERIES llevan su tiempo. Es en estos casos cuando podemos aprovechar las funcionalidades que brinda SQL Server para cambiar la percepción sobre el rendimiento, para ello podemos hacer uso de FAST, de la siguiente forma:

```
OPTION(FAST number_of_rows)
```

Donde number_of_rows es el número de filas a devolver mientras la QUERY sigue ejecutándose en background hasta devolver todos los demás registros.

DERIVED TABLES en lugar de TEMPORARY TABLES

En lugar de usar tablas temporales, usaremos tablas derivadas para mejorar el rendimiento de nuestra QUERY. Esto funciona de la siguiente forma:

```
SELECT num_Customer, dt_Date FROM (SELECT num_Customer, dt_Date,
nom_Customer  FROM Customers)
```

Con este tipo de tablas mejoramos el rendimiento de nuestra QUERY ya que se producen menos operaciones de I/O sobre el servidor.

Comparaciones entre TABLAS PADRE y TABLAS HIJAS

Es bastante común realizar comparaciones entre tablas padre y tablas hijas, existen tres formas de realizar dichas comparaciones que a continuación explicamos en ejemplos y que sirven para encontrar si hay registros padre que no tienen su igual en la tabla hija:

Usando NOT EXISTS

```
SELECT a.hdr_key
FROM hdr_tbl a
WHERE NOT EXISTS (SELECT * FROM dtl_tbl b WHERE a.hdr_key = b.hdr_key)
```

Usando LEFT JOIN

```
SELECT a.hdr_key
FROM hdr_tbl a
LEFT JOIN dtl_tbl b ON a.hdr_key = b.hdr_key
WHERE b.hdr_key IS NULL
```

Usando NOT IN

```
SELECT hdr_key
FROM hdr_tbl
WHERE hdr_key NOT IN (SELECT hdr_key FROM dtl_tbl)
```

En cada caso las QUERIES anteriores devuelven el mismo resultado. Pero ¿cuál de ellas tiene mejor rendimiento? Asumiendo que todo lo demás es igual, la versión que tiene mejor rendimiento es la primera y la última es la que peor rendimiento tiene (están ordenadas de mejor a peor rendimiento). En otras palabras, la variación NOT EXISTS es la QUERY más eficiente.

Esto es generalmente, quiero decir que a pesar de esto y dependiendo de los INDICES y el número de filas en cada tabla puede variar el resultado.

Comprobar la existencia de un registro en una tabla

Si necesitamos verificar si un registro existe en una tabla no usaremos nunca SELECT COUNT (*) para identificarla ya que es muy ineficiente y utiliza recursos de servidor de forma masiva.

En su lugar la sentencia Transact-SQL IF EXISTS para determinar si el registro en cuestión existe que es mucho más eficiente. Por ejemplo:

Usando SELECT COUNT (*):

```
IF (SELECT COUNT(*) FROM table_name WHERE column_name = 'xxx')
```

Usando IF EXISTS (mucho más rápido):

```
IF EXISTS (SELECT * FROM table_name WHERE column_name = 'xxx')
```

La razón por la cual IF EXISTS es más rápido que SELECT COUNT (*) es porque en el momento que dicha QUERY encuentra el registro finaliza inmediatamente, mientras que COUNT(*) debe contar todas las filas.