ISIL APRENDE HACIENDO

SQL Server 2012

Plan de Ejecución

Temas

1 Introducción

Para realizar el SQL Tunning a una base de datos, en la mayoría de los casos es necesario paciencia y tiempo para analizar con detenimiento como está funcionando todo, así en base a eso, determinar medidas de acción para la optimización.

Cada vez que se ejecuta una consulta en un motor de bases de datos, internamente se ejecutan una serie de operaciones, que varían según la consulta, los datos y obviamente, el motor de base de datos.

El conjunto de pasos que tiene que realizar el motor para ejecutar la consulta, se llama **Plan de Ejecución**. En este documento se explica cómo entender el plan de ejecución de SQL Server.

SSMS permite revisar el plan de ejecución de una consulta, tanto el estimado como el real.

- **Plan de Ejecución Estimado:**

  Se ejecuta desde el menú **Consultas -> Mostrar plan de ejecución estimado**. Esto no ejecutará la consulta sino más bien lo analizará y mostrará una aproximación del costo de su ejecución.

- **Plan de Ejecución Real:**

  Se activa desde el menú **Consultas -> Incluir plan de ejecución actual**. Con esta opción podremos ver el costo real de una consulta, pero a diferencia del anterior, se muestra al terminar la ejecución de la misma.

2 Costo estimado de la ejecución

Lo primero que se tiene que entender acerca de los planes de ejecución es cómo se generan. SQL Server usa un optimizador de consultas basado en el costo, es decir, intenta generar un plan de ejecución con el costo estimado más bajo.

La estimación se basa en las estadísticas de distribución de datos que están disponibles para el optimizador cuando evalúa cada tabla implicada en la consulta. Si esas estadísticas se pierden o quedan obsoletas, el optimizador de consultas carecerá de la información esencial necesaria para el proceso de optimización de consultas y, por lo tanto, es probable que las estimaciones queden fuera de la marca. En dichos casos, el optimizador seleccionará un plan menos óptimo tanto sobrestimando como subestimando los costos de ejecución de los distintos planes.

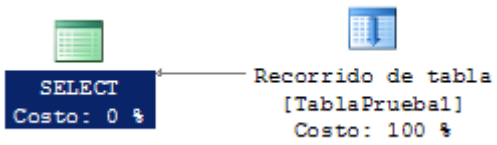
3 Table Scan

Significa que el motor tiene que leer toda la tabla. Esto solo puede suceder cuando la tabla es Heap (o sea, no tiene un índice clustered).

En algunos casos, cuando se trata de una tabla pequeña, un Table Scan es la mejor opción, ya que produce poco overhead. De hecho la tabla puede tener índices y sin embargo el SQL Server elige usar un Table Scan porque sería más rápido.

Pero cuando la tabla es más grande, no debería haber Table Scan, ya que es muy costoso. Para solucionar este problema, hay ver si la tabla tiene índices y si se están usando correctamente. Lo importante es prestarle atención cuando vemos un Table Scan. Muchas veces, el problemas de performance pasan por ahí.

```
CREATE TABLE dbo.TablaPrueba1
(
  Campo1 int IDENTITY (1, 1) NOT NULL,
  Campo2 int,
  Campo3 int,
  Campo4 int,
  Campo5 char (30)
);
go

SELECT * FROM TablaPrueba1;
go
```



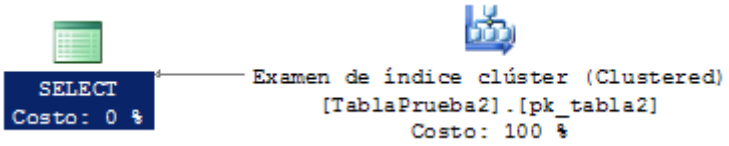
4 Clustered Index Scan

Esta operación es muy similar a un Table Scan. El motor recorre toda la tabla. La diferencia entre uno y otro, es que el Clustered Index Scan se realiza en una tabla que tiene un índice Clustered y el Table Scan en una tabla que no tiene este tipo de índice.

Otra vez tenemos que evaluar si esta opción es la que realmente queremos. Muchas veces, por un mal uso de los índices, se ejecuta esta operación, cuando en realidad queríamos otra más eficiente.

```
CREATE TABLE dbo.TablaPrueba2
(
  Campo1 int IDENTITY (1, 1) NOT NULL,
  Campo2 int,
  Campo3 int,
  Campo4 int,
  Campo5 char (30),
  constraint pk_tabla2 primary key clustered(Campo1)
);
go
```

```
SELECT * FROM TablaPrueba2;
go
```

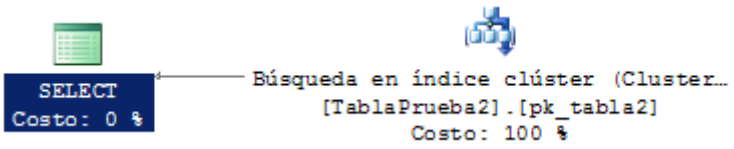


5 Clustered Index Seek

Si vemos esta operación, en general, podemos estar contentos. Significa que el motor está usando efectivamente el índice Clustered de la tabla.

Usamos la tabla creada en el caso anterior con un índice clustered, se insertan 10000 registros para que el motor prefiriera usar el índice antes que un scan y filtramos por el índice.

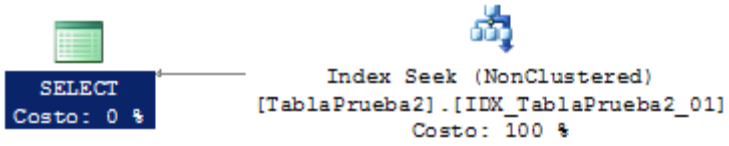
```
SET NOCOUNT ON;
DECLARE @Top int;
SET @Top = 0;

WHILE ( @Top <> 10000 )
BEGIN

  INSERT INTO TablaPrueba2(Campo2, Campo3, Campo4, Campo5)
  VALUES (convert(int,rand()*20000),convert(int,rand()*20000),
  convert(int,rand()*20000), 'P');

  SET @Top = @Top + 1;

END;
go

SELECT * FROM TablaPrueba2 WHERE Campo1 = 2;
go
```



6 Index Seek

Si vemos esta operación, quiere decir que se está utilizando el índice. Es similar que el Clustered Index Seek, pero con la diferencia de que se usa un índice Non Clustered.

```
CREATE INDEX IDX_TablaPrueba2_01
ON dbo.TablaPrueba2(Campo2,Campo3);
go

SELECT Campo2, Campo3
FROM TablaPrueba2
WHERE Campo2 = 2 and Campo3 = 2;
```

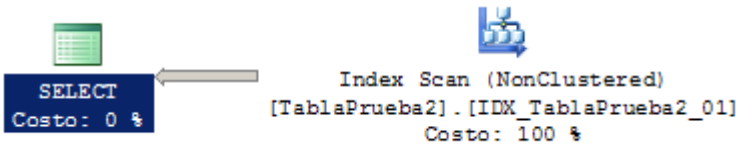


7 Index Scan

Esta operación se ejecuta cuando se lee el índice completo de una tabla. Es preferible a un Table Scan, ya que obviamente leer un índice es más pequeño que una tabla. Esta operación puede ser síntoma de un mal uso del índice, aunque también puede ser que el motor haya seleccionado que esta es la mejor operación. Es muy común un Index Scan en un JOIN o en un ORDER BY o GROUP BY.

Usemos la tabla TablaPrueba2 creada anteriormente.

```
SELECT Campo2 FROM TablaPrueba2;
go
```



Al no tener ningún filtro la consulta, el motor debe leer toda la tabla. Sin embargo, al traer solo el Campo2, que pertenece a un índice Non Clustered, en vez de hacer un Table Scan, es más óptimo hacer un Index Scan.

8 Bookmark Lookup

El Bookmark Lookup indica que SQL Server necesita ejecutar un salto del puntero desde la página de índice a la página de datos de la tabla para recuperar los datos. Esto sucede siempre que tenemos un índice Non Clustered. Para evitar esta operación, hay que limitar los campos que queremos traer en la consulta.

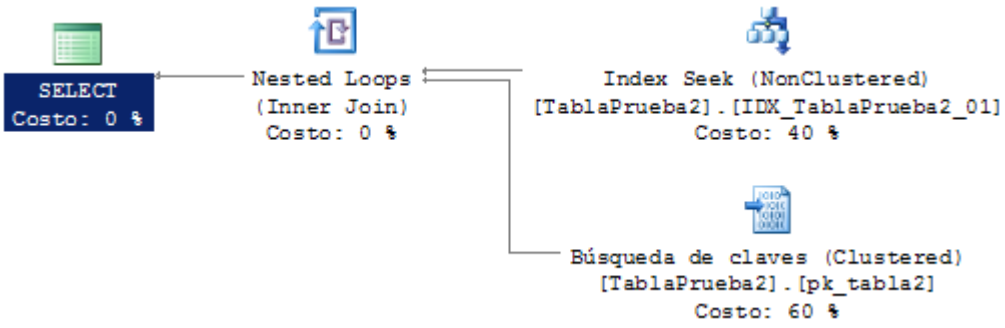
Si el campo que vamos a extraer, esta fuera del índice, entonces se va a ejecutar esta operación y no queda otra opción. Acá reside la importancia de evitar los SELECT * .

Debemos comprender que no es fácil evitar tener este tipo de operación en la BD, lo que si se debe buscar es que sea mínima su aparición y cuando lo hace, lo haga con un costo bajo.

En algunos casos extremos (muy extremos) se podría considerar la inclusión de todas las columnas de la tabla dentro del índice.

Veamos el siguiente ejemplo usando nuevamente la tabla TablaPrueba2:

```
SELECT Campo2, Campo3, Campo4
FROM TablaPrueba2
WHERE Campo2 = 2;
```



9 Joins

Un join es la relación entre 2 tablas. SQL tiene tres tipos de joins. Neested Loop Join, Merge Join y Hash Join. Dependiendo de las características de la consulta y de la cantidad de registros, el motor puede decidir uno u otro.

Ninguno es peor o mejor, todo depende de las características de la consulta y del volumen de datos.

- **Neested Loop Join:**

  Suele ser generalmente el más frecuente. Es también el algoritmo más simple de todo. Este operador físico es usado por el motor cuando tenemos un join entre 2 tablas y la cantidad de registros es relativamente baja. También aplica con cierto tipo de joins (cross joins por ejemplo).

- **Merge Join:**

  Otro de los tipos de join que existen. Generalmente se usa cuando las cantidades de registros a comparar son relativamente grandes y están ordenadas. Aun si no están ordenadas, el motor puede predecir que es más rápido ordenar la tabla y hacer el merge join que hacer un Neested Loop Join. En muchas situaciones es frecuente ver que una consulta anteriormente usaba Neested Loop Join y en algún momento paso a usar un Merge Join. La razón de esto, es porque el volumen de datos aumento y por lo tanto, es más óptimo usar un Merge join.

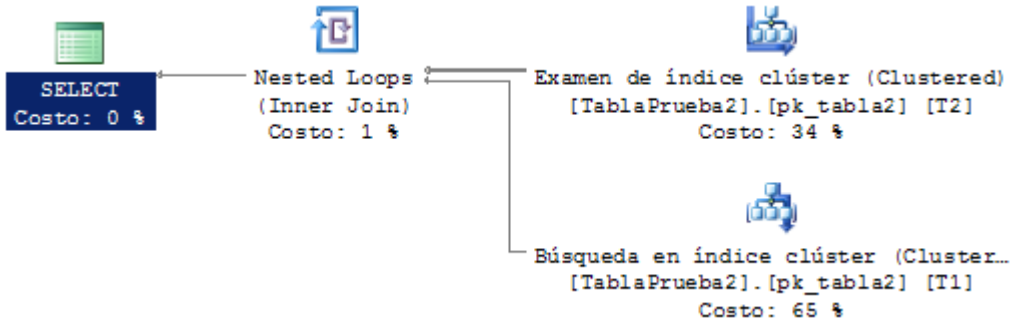
- **Hash Join:**

  Otro tipo más de join que existe. Mientras que los Loop Joins trabajan bien para conjuntos chicos de datos y los merge join para conjuntos moderados de datos, el hash join es especialmente útil en grandes conjuntos de datos, generalmente en datawarehouses. Este operador es mucho más escalable. Generalmente también se usa cuando las tablas relacionadas no tienen índice en ninguna de los campos a comparar. Hay que prestar atención si vemos este tipo de operaciones, ya que puede significar un mal uso de los índices. Sin embargo, los hash joins consumen mucha memoria y SQL Server tiene un límite en la cantidad de operaciones de este tipo que puede efectuar simultáneamente.

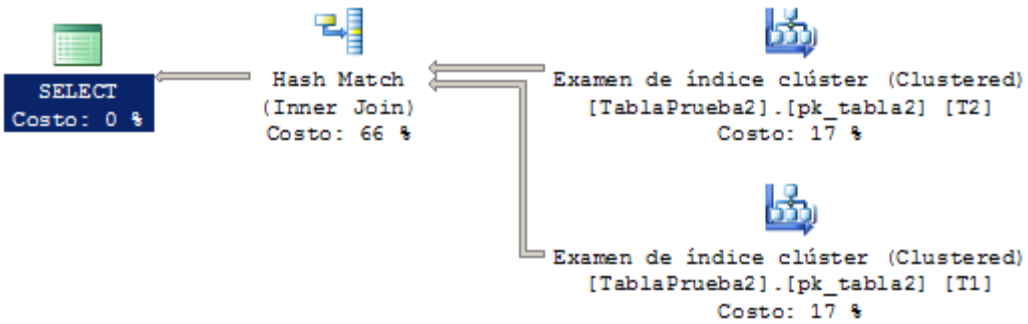
Se va a ejecutar exactamente la misma consulta con una tabla con 50 registros y con 2000 registros, para ver cómo cambia en función del volumen de datos, el tipo de operación.

```
SELECT T1.*
FROM tablaprueba2 T1
INNER JOIN TablaPrueba2 T2
ON T2.Campo4 = T1.Campo1;
```

Con 20 registros:



Con 2000 registros:



10 Agregaciones

Las agregaciones refieren a agrupar un conjunto grande de datos en un conjunto de datos más chico.

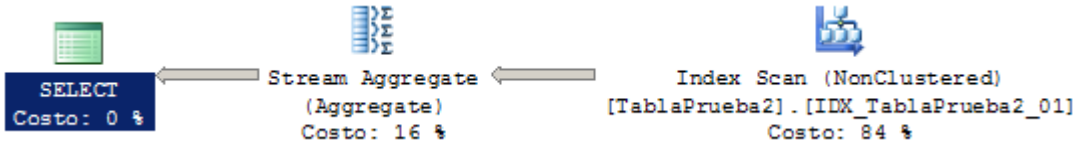
- **Stream Aggregate:**

  Este tipo de operaciones ocurre cuando se llama a un función de agregación, como MIN, COUNT, MAX, SUM, etc. El operador Stream Aggregate requiere que la información esté ordenada por las columnas dentro de sus grupos. Primero, el optimizador ordenará si los datos no están ordenados por un operador Sort anterior. En cierta manera, el Stream Aggregate es similar al Merge Join, en cuanto a en que situaciones se produce.

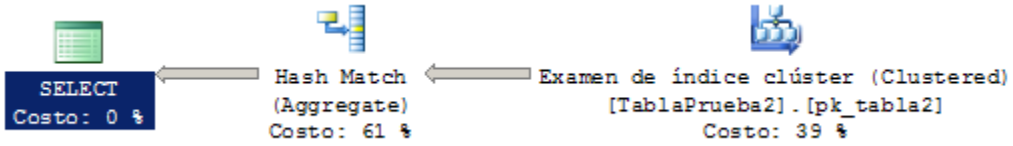
- **Hash Match (Aggregate):**

  Hay que tener cuidado cuando vemos este operador. Esta operación también ocurre cuando se llama a funciones de agregación del tipo MIN, COUNT, AVG, etc. Así como el Stream Aggregate es comparable al Merge Join, el Hash Match Aggregate es similar al Hash Join. Lo que hace internamente es armar una tabla de hash. En situaciones donde la cantidad de registros es elevada o no están indexadas las columnas por las cuales agrupa la consulta, el motor del SQL va a elegir esta operación.

```
SELECT MAX(Campo2)
FROM TablaPrueba2
GROUP BY Campo2;
```




```
SELECT MAX(Campo4)
FROM TablaPrueba2
GROUP BY Campo4;
```



Como podemos observar en este caso, las 2 consultas son prácticamente similares en estructura, solo que el primer caso agrupa el campo2 que esta indexado y en el segundo caso, agrupa el campo4, que no está indexado y por eso usa el operador Hash Match.

11 Sort

Otro punto para observar, es cuando vemos un sort. Como el nombre lo indica, esta operación ordena. Ahora, el Sort solo se hace cuando el campo o los campos que se desean

ordenar, no están indexados. A veces esta operación se ejecuta sola, sin que nosotros hayamos puesto en la consulta el ORDER BY, porque el motor necesita ordenar los datos por alguna razón, por ejemplo, para ejecutar un Merge Join. Pero recordemos que si ordenamos por un campo indexado y este índice está siendo utilizado, no se ejecuta esta operación.

Ejemplo de esta operación:

```
SELECT * FROM TablaPrueba2 ORDER BY Campo3;
```


```
SELECT campo2, campo3 FROM TablaPrueba2 ORDER BY Campo3;
```