Daniel Granados Retana

Diego Granados Retana

Bases de Datos I

Prof. Rodrigo Núñez

Query con múltiples joins

Para este ejercicio, utilizamos la siguiente query:

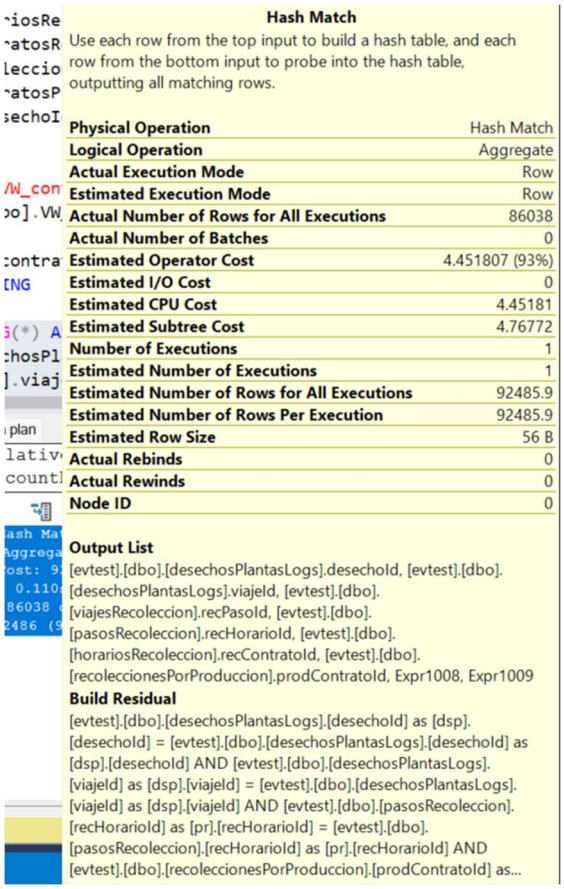
```
SELECT COUNT_BIG(*) AS countbig, dsp.desechoId AS desecho,
dsp.viajeId AS viaje, vr.recPasoId AS paso, pr.recHorarioId AS horario, hr.recContratoId AS contratoRec,
rpp.prodContratoId AS contratoProd FROM desechosPlantasLogs dsp
INNER JOIN viajesRecoleccion vr ON vr.viajeId = dsp.viajeId
INNER JOIN pasosRecoleccion pr ON pr.recPasoId = vr.recPasoId
INNER JOIN horariosRecoleccion hr ON hr.recHorarioId = pr.recHorarioId
INNER JOIN contratosRecoleccion cr ON cr.recContratoId = hr.recContratoId
INNER JOIN recoleccionesPorProduccion rpp ON rpp.recContratoId = cr.recContratoId
INNER JOIN contratosProduccion cp ON cp.prodContratoId = rpp.prodContratoId
GROUP BY dsp.desechoId, dsp.viajeId, vr.recPasoId, pr.recHorarioId, hr.recContratoId, rpp.prodContratoId
```

El objetivo de este query es obtener el contrato de producción de cada desecho que se añade a una planta luego de su recolección, con el fin de saber en qué producción serán utilizados. Luego, creamos los views. El primer índice que se crea sobre una vista debe ser un unique clustered index. Cada vez que intentábamos crear el index, nos daba un error: había valores repetidos. Para solucionar esto, intentamos utilizar un distinct, pero tampoco se puede usarlo. Nos decía que podíamos usar un aggregate function o un COUNT_BIG(*), entonces también le pusimos el COUNT_BIG(*), que es una función que retorna la cantidad de elementos en un grupo. Ya con esto logramos crear el siguiente índice porque el COUNT_BIG(*) cuenta y agrupa los registros iguales y así los simplifica en uno solo:

```
CREATE UNIQUE CLUSTERED INDEX IDX_contratoDesechosView
    ON VW_contratoDesechosLogsIDX (desecho, viaje, paso, horario, contratoRec, contratoProd);
GO
```

Utilizamos estas columnas porque las posibles combinaciones únicas que se dan para cada desecho de cada viaje, por lo que accederlas rápidamente es muy ventajoso para el proceso.

De acuerdo con el execution plan, la instrucción que toma más tiempo es:



Hash Match

Use each row from the top input to build a hash table, and each row from the bottom input to probe into the hash table, outputting all matching rows.

Physical Operation	Hash Match
Logical Operation	Aggregate
Actual Execution Mode	Row
Estimated Execution Mode	Row
Actual Number of Rows for All Executions	86038
Actual Number of Batches	0
Estimated Operator Cost	4.451807 (93%)
Estimated I/O Cost	0
Estimated CPU Cost	4.45181
Estimated Subtree Cost	4.76772
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows for All Executions	92485.9
Estimated Number of Rows Per Execution	92485.9
Estimated Row Size	56 B
Actual Rebinds	0
Actual Rewinds	0
Node ID	0

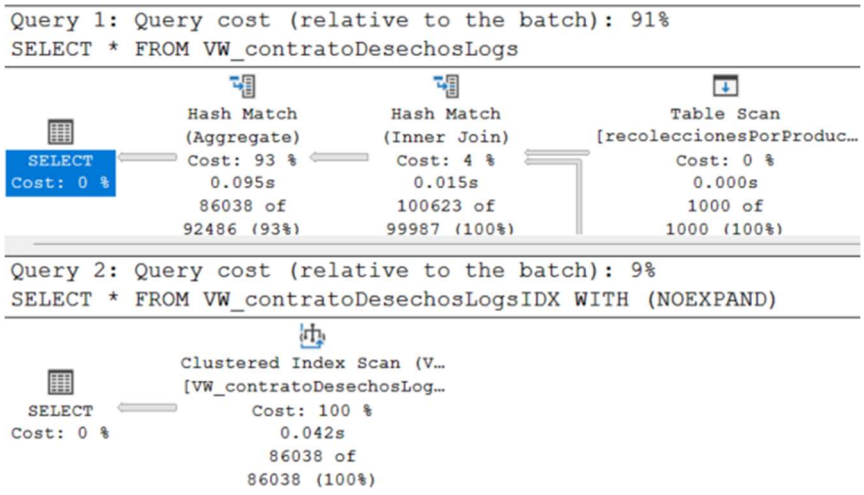
Output List

[evtest].[dbo].[desechosPlantasLogs].desechoId, [evtest].[dbo].[desechosPlantasLogs].viajeId, [evtest].[dbo].[viajesRecoleccion].recPasoId, [evtest].[dbo].[pasosRecoleccion].recHorarioId, [evtest].[dbo].[horariosRecoleccion].recContratoId, [evtest].[dbo].[recoleccionesPorProduccion].prodContratoId, Expr1008, Expr1009

Build Residual

[evtest].[dbo].[desechosPlantasLogs].[desechoId] as [dsp].[desechoId] = [evtest].[dbo].[desechosPlantasLogs].[desechoId] as [dsp].[desechoId] AND [evtest].[dbo].[desechosPlantasLogs].[viajeId] as [dsp].[viajeId] = [evtest].[dbo].[desechosPlantasLogs].[viajeId] as [dsp].[viajeId] AND [evtest].[dbo].[pasosRecoleccion].[recHorarioId] as [pr].[recHorarioId] = [evtest].[dbo].[pasosRecoleccion].[recHorarioId] as [pr].[recHorarioId] AND [evtest].[dbo].[recoleccionesPorProduccion].[prodContratoId] as...

Este hash match lo que está haciendo es crear un hash table con base en las columnas desechosPlantasLogs.desechoId, desechosPlantasLogs.viajeId, pasosRecoleccion.recHorarioId y recoleccionesPorProducto.prodContratoId. Luego de crear el índice, vimos que hubo una diferencia sustancial en el costo del query y un leve aumento en la velocidad.



```
(86038 rows affected)

(1 row affected)

 SQL Server Execution Times:
   CPU time = 63 ms,  elapsed time = 630 ms.

(86038 rows affected)

(1 row affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 567 ms.

Completion time: 2023-04-30T20:17:11.1691048-06:00
```

Incluso, los resultados de la vista indexada estaban ordenados porque el clustered index los ordena.

	countbig	desecho	viaje	paso	horario	contratoRec	contratoProd
1	1	1	370	154	912	8	628
2	1	5	597	268	308	3	147
3	1	5	524	372	604	7	144
4	1	5	180	604	51	5	919
5	2	5	902	868	880	5	189
6	1	2	405	363	358	4	320
7	1	1	118	431	475	7	639
8	1	4	663	630	380	1	655

	countbig	desecho	viaje	paso	horario	contratoRec	contratoProd
1	1	1	4	877	879	1	31
2	1	1	4	877	879	1	34
3	1	1	4	877	879	1	39
4	1	1	4	877	879	1	54
5	1	1	4	877	879	1	55
6	1	1	4	877	879	1	63
7	1	1	4	877	879	1	72
8	1	1	4	877	879	1	77

No obstante, luego intentamos crear un nonclustered index sobre cada campo del build residual, pero eso no afectó positivamente al rendimiento.

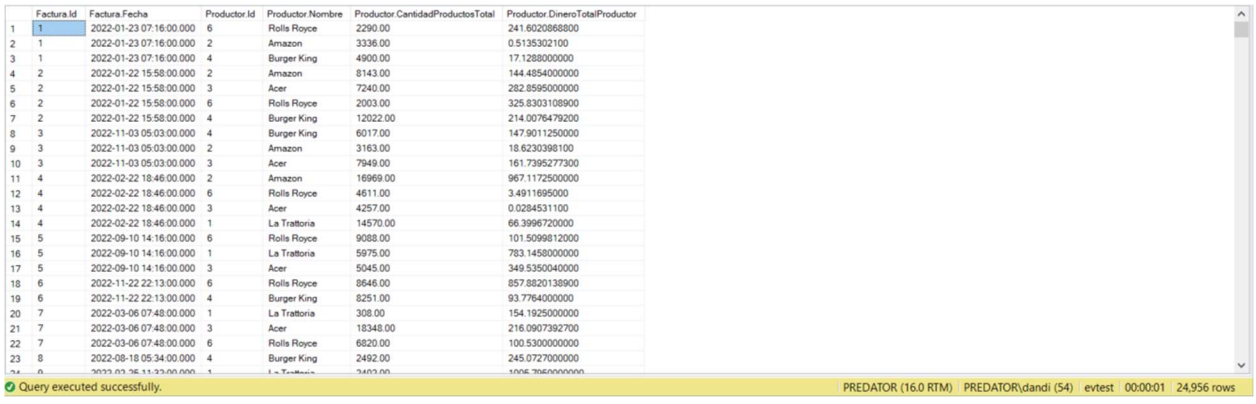
La razón por la cual disminuyó tanto el costo de la vista indexada es porque los resultados de la vista se almacenan como si fueran una tabla (Yaseen, 2019). En una vista normal dinámica, los resultados se calculan todas las veces con base en el query del view, por lo que no hay un gran beneficio en el rendimiento. Por lo tanto, acceder a la información con una vista indexada va a ser mucho más rápido, debido a que ya está calculada. Por otro lado, si los datos de la vista son actualizados frecuentemente o la vista no se llama mucho, es posible que el uso de espacio y el costo de actualizar los datos sea mayor al beneficio que trae la vista indexada. Es por esta razón que se utiliza la instrucción SCHEMABINDING, que hace que no se puedan aplicar cambios estructurales a las tablas involucradas en el índice.

Por lo tanto, como norma, si la tabla va a cambiar mucho, es mejor usar una vista dinámica. Si no, es mejor usar una vista indexada para facilitar el acceso a la información.

Norma de optimización

Por cada factura, la cual se compone de ítems de venta de productos, obtener la cantidad total de los productos vendidos, donde el tipo de producto no sea 2 ("colchón"), que fueron producidos en un proceso donde participó un productor (objectType = 7). Obtener el dinero correspondiente a cada productor en cada factura, con base en los porcentajes de ganancia y el monto de la venta. El dinero se presenta en la moneda base. Se excluyen los resultados correspondientes al productor 5 (GGGames).

```
SELECT facturas.facturaId AS [Factura.Id], facturas.fecha AS [Factura.Fecha],
productores.productorId AS [Productor.Id], productores.nombre AS [Productor.Nombre],
SUM(items.cantidadProductos) AS [Productor.CantidadProductosTotal],
SUM((items.montoTotal/tiposDeCambio.conversion) * porcentajes.porcentaje / 100) AS [Productor.DineroTotalProductor]
FROM facturas LEFT JOIN itemsFactura ON facturas.facturaId = itemsFactura.facturaId
INNER JOIN itemsProductos items ON items.itemProdId = itemsFactura.itemId
INNER JOIN lotesProduccionLogs lpl ON items.loteId = lpl.loteId
INNER JOIN contratosProduccion contProd ON contProd.prodContratoId = lpl.prodContratoId
INNER JOIN actoresContratoProd actores ON actores.prodContratoId = contProd.prodContratoId
INNER JOIN porcentajesActores porcentajes ON porcentajes.actorId = actores.actorId
RIGHT JOIN productores ON productores.productorId = actores.genericId
INNER JOIN tiposDeCambio ON tiposDeCambio.monedaCambioId = items.monedaId
WHERE items.fecha BETWEEN '2022-01-01 00:00:00' AND GETDATE() AND
itemsFactura.tipoItemId = 3 AND -- si es un item de venta de producto
actores.objectTypeId = 7 AND -- si el actor es un productor
porcentajes.productoId = lpl.productoId AND -- si el actor tiene un porcentaje del producto, participó en la producción de ese producto
lpl.productoId != 2 -- si la venta no involucra el producto 2.
GROUP BY facturas.facturaId, facturas.fecha, actores.genericId, productores.productorId, productores.nombre
EXCEPT
SELECT facturas.facturaId, facturas.fecha, productores.productorId, productores.nombre, SUM(items.cantidadProductos) cantidadProductosTotal,
SUM((items.montoTotal/tiposDeCambio.conversion) * porcentajes.porcentaje / 100) dineroProductor
FROM facturas LEFT JOIN itemsFactura ON facturas.facturaId = itemsFactura.facturaId
INNER JOIN itemsProductos items ON items.itemProdId = itemsFactura.itemId
INNER JOIN lotesProduccionLogs lpl ON items.loteId = lpl.loteId
INNER JOIN contratosProduccion contProd ON contProd.prodContratoId = lpl.prodContratoId
INNER JOIN actoresContratoProd actores ON actores.prodContratoId = contProd.prodContratoId
INNER JOIN porcentajesActores porcentajes ON porcentajes.actorId = actores.actorId
RIGHT JOIN productores ON productores.productorId = actores.genericId
INNER JOIN tiposDeCambio ON tiposDeCambio.monedaCambioId = items.monedaId
WHERE items.fecha BETWEEN '2022-01-01 00:00:00' AND GETDATE() AND
itemsFactura.tipoItemId = 3 AND
actores.objectTypeId = 7 AND
porcentajes.productoId = lpl.productoId AND
lpl.productoId != 2 AND
productores.productorId = 5 -- excluimos el productor 5
GROUP BY facturas.facturaId, facturas.fecha, actores.genericId, productores.productorId, productores.nombre
ORDER BY [Factura.Id]
FOR JSON PATH, ROOT ('Facturas')
```



	Factura.Id	Factura.Fecha	Productor.Id	Productor.Nombre	Productor.CantidadProductosTotal	Productor.DineroTotalProductor
1	1	2022-01-23 07:16:00.000	6	Rolls Royce	2290.00	241.6020868800
2	1	2022-01-23 07:16:00.000	2	Amazon	3336.00	0.5135302100
3	1	2022-01-23 07:16:00.000	4	Burger King	4900.00	17.1288000000
4	2	2022-01-22 15:58:00.000	2	Amazon	8143.00	144.4854000000
5	2	2022-01-22 15:58:00.000	3	Acer	7240.00	282.8595000000
6	2	2022-01-22 15:58:00.000	6	Rolls Royce	2003.00	325.8303108900
7	2	2022-01-22 15:58:00.000	4	Burger King	12022.00	214.0076479200
8	3	2022-11-03 05:03:00.000	4	Burger King	6017.00	147.9011250000
9	3	2022-11-03 05:03:00.000	2	Amazon	3163.00	18.6230398100
10	3	2022-11-03 05:03:00.000	3	Acer	7949.00	161.7395277300
11	4	2022-02-22 18:46:00.000	2	Amazon	16969.00	967.1172500000
12	4	2022-02-22 18:46:00.000	6	Rolls Royce	4611.00	3.4911695000
13	4	2022-02-22 18:46:00.000	3	Acer	4257.00	0.0284531100
14	4	2022-02-22 18:46:00.000	1	La Trattoria	14570.00	66.3996720000
15	5	2022-09-10 14:16:00.000	6	Rolls Royce	9088.00	101.5099812000
16	5	2022-09-10 14:16:00.000	1	La Trattoria	5975.00	783.1458000000
17	5	2022-09-10 14:16:00.000	3	Acer	5045.00	349.5350040000
18	6	2022-11-22 22:13:00.000	6	Rolls Royce	8646.00	857.8820138900
19	6	2022-11-22 22:13:00.000	4	Burger King	8251.00	93.7764000000
20	7	2022-03-06 07:48:00.000	1	La Trattoria	308.00	154.1925000000
21	7	2022-03-06 07:48:00.000	3	Acer	18348.00	216.0907392700
22	7	2022-03-06 07:48:00.000	6	Rolls Royce	6820.00	100.5300000000
23	8	2022-08-18 05:34:00.000	4	Burger King	2492.00	245.0727000000

Query executed successfully. | PREDATOR (16.0 RTM) | PREDATOR\dandi (54) | evtest | 00:00:01 | 24,956 rows

	JSON_F52E2B61-18A1-11d1-B105-00805F49916B
1	{"Facturas":[{"Factura":{"Id":1,"Fecha":"2022-01-23T...

Rendimiento del query original: Se ejecutó en 442 ms.

```
(24956 rows affected)

(1 row affected)

 SQL Server Execution Times:
   CPU time = 328 ms,  elapsed time = 442 ms.

Completion time: 2023-04-28T18:40:49.1646303-06:00
```

Unidad de workload	Explicación	Norma
Clustered Index Scan (Clustered) Scanning a clustered index, entirely or only a range. **Physical Operation** Clustered Index Scan **Logical Operation** Clustered Index Scan **Actual Execution Mode** Batch **Estimated Execution Mode** Batch **Storage** RowStore **Actual Number of Rows Read** 200000 **Actual Number of Rows for All Executions** 200000 **Actual Number of Batches** 223 **Estimated I/O Cost** 1.72535 **Estimated Operator Cost** 1.9455 (12%) **Estimated Subtree Cost** 1.9455 **Estimated CPU Cost** 0.220157 **Estimated Number of Executions** 1 **Number of Executions** 1 **Estimated Number of Rows for All Executions** 200000 **Estimated Number of Rows Per Execution** 200000 **Estimated Number of Rows to be Read** 200000 **Estimated Row Size** 48 B **Actual Rebinds** 0 **Actual Rewinds** 0 **Ordered** False **Node ID** 39 **Predicate** [evtest].[dbo].[itemsProductos].[fecha] as [items].[fecha]<=getdate() AND [evtest].[dbo].[itemsProductos].[fecha] as [items].[fecha]>='2022-01-01 00:00:00.000' **Object** [evtest].[dbo].[itemsProductos].[PK_ventasProductosLogs] [items] **Output List** [evtest].[dbo].[itemsProductos].itemProdId, [evtest].[dbo].[itemsProductos].loteId, [evtest].[dbo].[itemsProductos].cantidadProductos, [evtest].[dbo].[itemsProductos].montoTotal, [evtest].[dbo].[itemsProductos].monedaId	El motor recorre la tabla de itemsProductos con base en el primary key PK_ventasProductosLogs y a partir de eso extrae los registros cuyas fechas estén entre el intervalo establecido. Se relaciona con el S7, donde se recorre a partir de un índice individual y se filtra por un conjunctive selection.	- Ordenar la tabla de itemsProductos en cuanto a la fecha para que sea más fácil encontrar las fechas que están en un intervalo sin tener que revisar toda la tabla. Para cuando las fechas se salgan del intervalo. - Cuando se hace un Clustered Index Scan y se filtra por una llave non-key, Revisar el campo non-key que está en el Predicate y agregar un NonClustered Index. Incluso, se pueden agregar las columnas que aparecen en el Output List, mientras no sean muchas, para extraer la información directamente. Se podría poner un nonclustered index en itemsProductos.fecha para ordenar los registros en cuanto a las fechas y sea más fácil acceder a ellos directamente.

V2: Se aplicará la norma de crear un nonclustered index en itemsProductos.fecha. Como son muchas columnas, no se incluyen como included columns.

El rendimiento mejoró a 425 ms, una mejora de 17ms.

```
(24956 rows affected)

(1 row affected)

 SQL Server Execution Times:
   CPU time = 297 ms,  elapsed time = 425 ms.

Completion time: 2023-04-28T18:42:20.6443534-06:00
```

Clustered Index Scan (Clustered)
Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	RowStore
Actual Number of Rows Read	200000
Actual Number of Rows for All Executions	200000
Actual Number of Batches	223
Estimated I/O Cost	1.72535
Estimated Operator Cost	1.9455 (12%)
Estimated Subtree Cost	1.9455
Estimated CPU Cost	0.220157
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows for All Executions	200000
Estimated Number of Rows Per Execution	200000
Estimated Number of Rows to be Read	200000
Estimated Row Size	48 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	39

Predicate
[evtest].[dbo].[itemsProductos].[fecha] as [items].[fecha]<=getdate() AND [evtest].[dbo].[itemsProductos].[fecha] as [items].[fecha]>='2022-01-01 00:00:00.000'
Object
[evtest].[dbo].[itemsProductos].[PK_ventasProductosLogs] [items]
Output List
[evtest].[dbo].[itemsProductos].itemProdId, [evtest].[dbo].[itemsProductos].loteId, [evtest].[dbo].[itemsProductos].cantidadProductos, [evtest].[dbo].[itemsProductos].montoTotal, [evtest].[dbo].[itemsProductos].monedaId

Sin embargo, este índice no contribuyó a disminuir el porcentaje en el costo de la operación ni la estrategia que utiliza.

Esto es porque el optimizador todavía determina que hacer el Clustered Index Scan es la forma más eficiente para realizar la operación. Debemos darle más herramientas para que lo haga más eficientemente.

Se intentó aplicar la norma de ordenar la tabla por fecha, pero al usar un EXCEPT y aggregate functions, el campo de itemsProductos.fecha debería estar en el output list y en el GROUP BY, lo cual retornaría resultados diferentes a lo que se busca con el query.

V3: En la misma operación anterior, se aplica la norma de agregar un nonclustered index con los campos que extrae en el output list.

```
SQL Server parse and compile time:
   CPU time = 94 ms, elapsed time = 98 ms.

(24956 rows affected)

(1 row affected)

 SQL Server Execution Times:
   CPU time = 281 ms,  elapsed time = 407 ms.

Completion time: 2023-04-28T18:47:35.9230894-06:00
```

Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	RowStore
Actual Number of Rows Read	200000
Actual Number of Rows for All Executions	200000
Actual Number of Batches	223
Estimated Operator Cost	1.08624 (8%)
Estimated I/O Cost	0.866088
Estimated Subtree Cost	1.08624
Estimated CPU Cost	0.220157
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows for All Executions	200000
Estimated Number of Rows to be Read	200000
Estimated Number of Rows Per Execution	200000
Estimated Row Size	40 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	39

Object
[evtest].[dbo].[itemsProductos].[IX_itemsProductos_fecha] [items]

Output List
[evtest].[dbo].[itemsProductos].itemProdId, [evtest].[dbo].[itemsProductos].loteId, [evtest].[dbo].[itemsProductos].cantidadProductos, [evtest].[dbo].[itemsProductos].montoTotal, [evtest].[dbo].[itemsProductos].monedaId

Seek Predicates
Seek Keys[1]: Start: [evtest].[dbo].[itemsProductos].fecha >= Scalar Operator('2022-01-01 00:00:00.000'), End: [evtest].[dbo].[itemsProductos].fecha <= Scalar Operator(getdate())

El rendimiento mejoró de 442ms a 407ms, una mejora de 35ms.

Esto también mejoró la estrategia y el costo de la operación en el plan de ejecución. Pasó de un Clustered Index Scan con un peso de 12%, a un Index Seek (NonClustered) con un peso de 8%.

Esto es porque puede usar la estructura del árbol B para llegar más directo al campo por donde se realiza la consulta. También, al almacenar las otras columnas en el índice, no tiene que realizar operaciones de I/O para extraer la información, por lo que lo hace más rápida la consulta

Unidad de workload	Explicación	Norma
Table Scan (Heap) Scan rows from a table. **Physical Operation**: Table Scan **Logical Operation**: Table Scan **Actual Execution Mode**: Batch **Estimated Execution Mode**: Batch **Storage**: RowStore **Actual Number of Rows Read**: 200000 **Actual Number of Rows for All Executions**: 200000 **Actual Number of Batches**: 223 **Estimated I/O Cost**: 1.34979 **Estimated Operator Cost**: 1.56995 (12%) **Estimated Subtree Cost**: 1.56995 **Estimated CPU Cost**: 0.220157 **Estimated Number of Executions**: 1 **Number of Executions**: 1 **Estimated Number of Rows for All Executions**: 200000 **Estimated Number of Rows Per Execution**: 200000 **Estimated Number of Rows to be Read**: 200000 **Estimated Row Size**: 24 B **Actual Rebinds**: 0 **Actual Rewinds**: 0 **Ordered**: False **Node ID**: 40 **Predicate** [evtest].[dbo].[itemsFactura].[tipoItemId]=(3) **Object** [evtest].[dbo].[itemsFactura] **Output List** [evtest].[dbo].[itemsFactura].facturaId, [evtest].[dbo].[itemsFactura].itemId	Aquí hace una búsqueda lineal (S1) sobre toda la tabla de itemsFactura y selecciona los registros donde el tipoItemId=3. Retorna los campos de facturaId y itemId.	Cuando hay que hacer una búsqueda secuencial para buscar dónde se cumple la condición en una columna, indexar la columna problema con un nonclustered index para hacer una búsqueda S5 donde a partir de la condición, extrae todos los records que la cumplen. En este caso, pondríamos un nonclustered index en tipoItemId. Adicionalmente, las columnas que tiene que extraer esos registros pueden incluirse como non-key included columns en el índice para obtenerlas directamente con el índice y así reducir las operaciones de I/O.

Se aplica la norma y agregamos un NonClustered Index con Included Columns en itemsFactura.tipoItemId y ponemos el output list (facturaId y itemId) como los included Columns.

```
(24956 rows affected)

(1 row affected)

 SQL Server Execution Times:
   CPU time = 313 ms,  elapsed time = 385 ms.

Completion time: 2023-04-28T19:00:51.0137170-06:00
```

De la versión anterior a esta, mejoró de 407ms a 385ms.

Aquí, el método pasó de un Table Scan a un Index Seek (NonClustered)

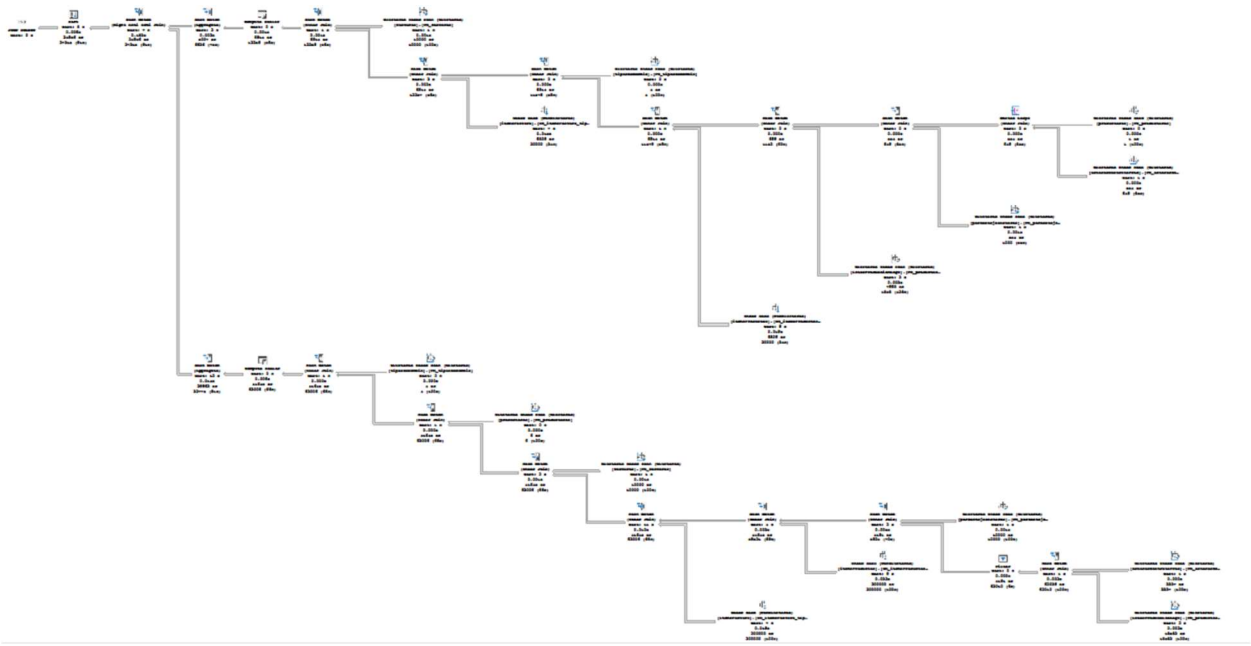
El porcentaje pasó de un 12% a un 7%.

Index Seek (NonClustered)
Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	RowStore
Actual Number of Rows Read	200000
Actual Number of Rows for All Executions	200000
Actual Number of Batches	223
Estimated Operator Cost	0.790689 (7%)
Estimated I/O Cost	0.570532
Estimated Subtree Cost	0.790689
Estimated CPU Cost	0.220157
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows for All Executions	200000
Estimated Number of Rows to be Read	200000
Estimated Number of Rows Per Execution	200000
Estimated Row Size	23 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	40

Object
[evtest].[dbo].[itemsFactura].[IX_itemsFactura_tipoItemId]
Output List
[evtest].[dbo].[itemsFactura].facturaId, [evtest].[dbo].[itemsFactura].itemId
Seek Predicates
Seek Keys[1]: Prefix: [evtest].[dbo].[itemsFactura].tipoItemId = Scalar Operator((3))

Diagrama de plan de ejecución:



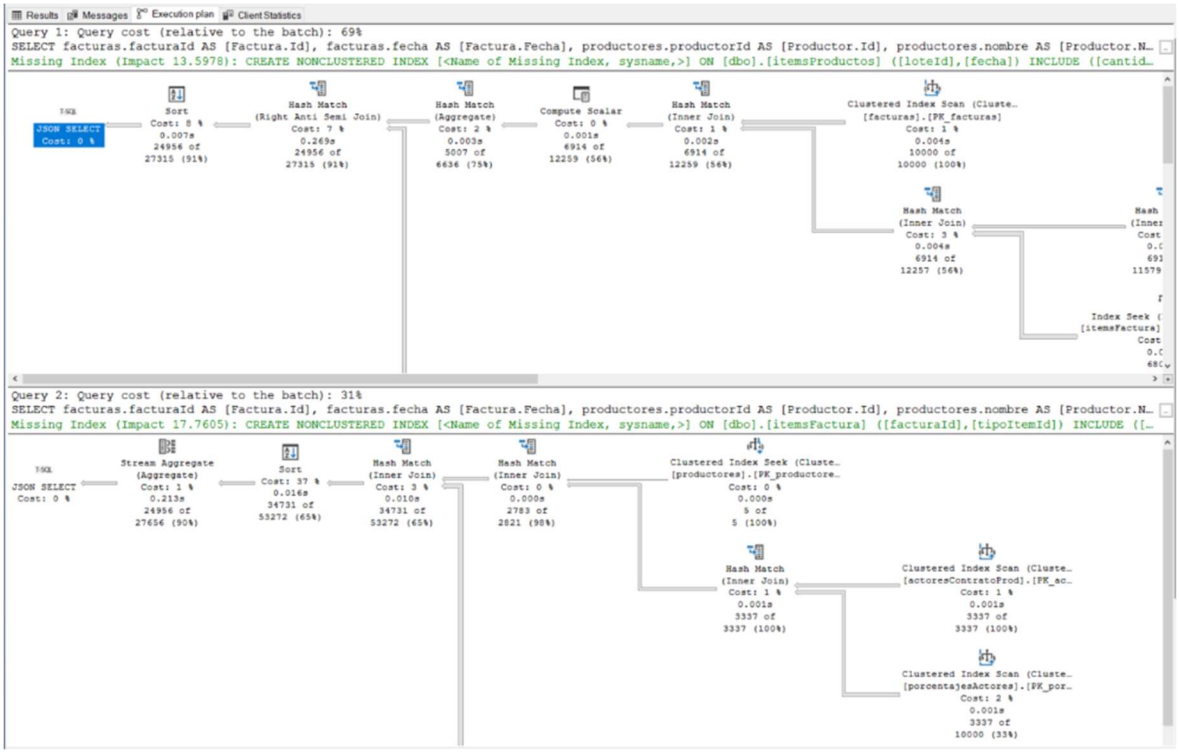
Explicación

Nuestro query contiene dos SELECTS, una para resolver los resultados generales y la otra para resolver los resultados que se deben excluir del primer SELECT por medio del EXCEPT. Por esta razón, el plan de ejecución se divide en dos ramas, las cuales son esencialmente iguales.

Norma

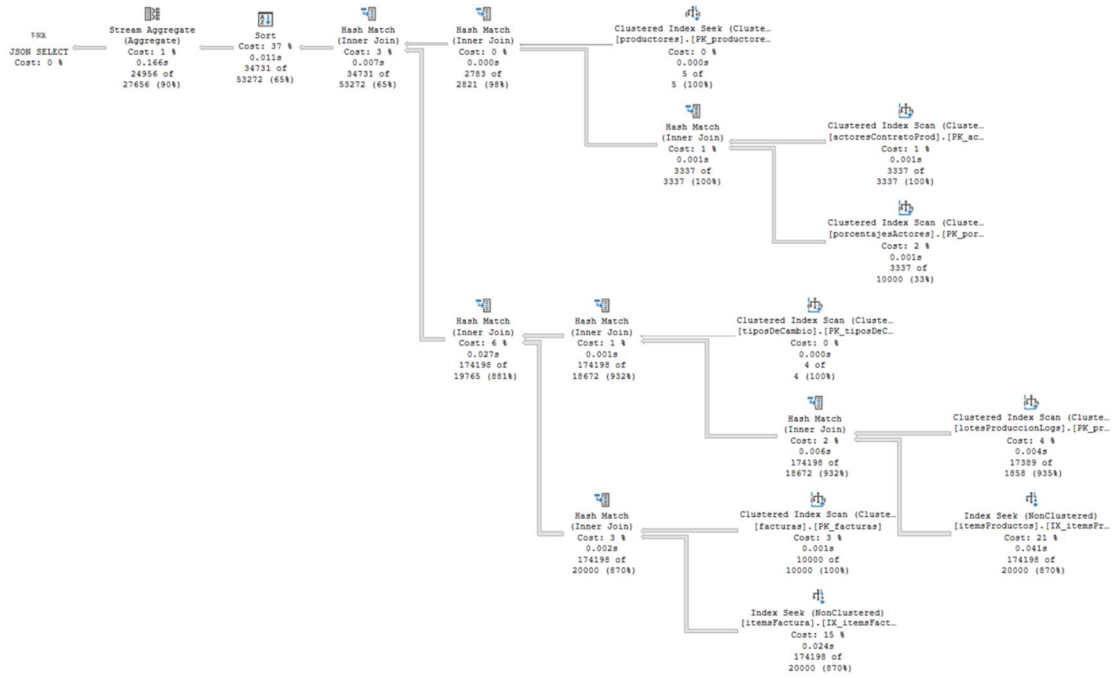
Si la consulta contiene un EXCEPT o INTERSECT que opera sobre consultas muy similares, intentar eliminar ese EXCEPT por medio de igualdades, desigualdades, o algún otro medio.

Resultados: Norma con el EXCEPT vs. Norma sin el EXCEPT



```
(24956 rows affected)

(1 row affected)

 SQL Server Execution Times:
   CPU time = 235 ms,  elapsed time = 341 ms.

Completion time: 2023-04-29T17:09:06.7297275-06:00
```

El rendimiento mejoró de 385ms a 341ms. El diagrama del plan de ejecución se redujo significativamente:



Esto ocurre porque solo tiene que resolver un query. No tiene que repetir operaciones, ya que el filtro del productorId se hace en el Clustered Index Seek en la tabla de productores.

Clustered Index Seek (Clustered)
Scanning a particular range of rows from a clustered index.

Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	RowStore
Actual Number of Rows Read	5
Actual Number of Rows for All Executions	5
Actual Number of Batches	2
Estimated Operator Cost	0.0032875 (0%)
Estimated I/O Cost	0.003125
Estimated Subtree Cost	0.0032875
Estimated CPU Cost	0.0001625
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows for All Executions	5
Estimated Number of Rows to be Read	5
Estimated Number of Rows Per Execution	5
Estimated Row Size	51 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	4

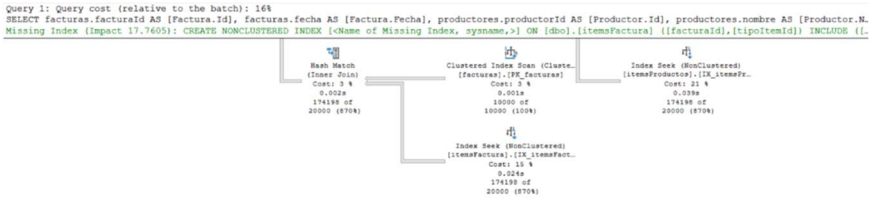
Object
[evtest].[dbo].[productores].[PK_productores]
Output List
[evtest].[dbo].[productores].productorId, [evtest].[dbo].[productores].nombre
Seek Predicates
[1] Seek Keys[1]: End: [evtest].[dbo].[productores].productorId < Scalar Operator((5)), [2] Seek Keys[1]: Start: [evtest].[dbo].[productores].productorId > Scalar Operator((5))

Esto se nota en las cantidades de lectura en las tablas. Se reducen a la mitad.

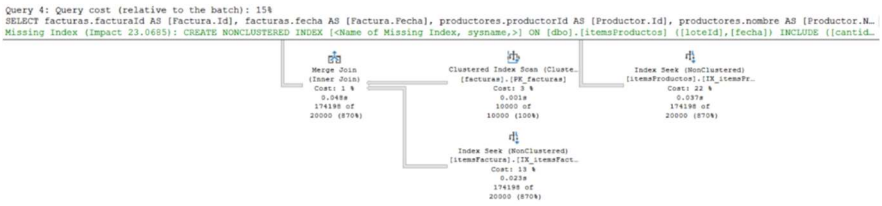
```
(24956 rows affected)
Table 'itemsFactura'. Scan count 2, logical reads 1542, physical reads 0, p
Table 'itemsProductos'. Scan count 2, logical reads 2350, physical reads 0,
Table 'lotesProduccionLogs'. Scan count 2, logical reads 492, physical read
Table 'actoresContratoProd'. Scan count 2, logical reads 168, physical read
Table 'porcentajesActores'. Scan count 2, logical reads 258, physical reads
Table 'facturas'. Scan count 2, logical reads 430, physical reads 0, page s
Table 'productores'. Scan count 1, logical reads 4, physical reads 0, page
Table 'tiposDeCambio'. Scan count 2, logical reads 4, physical reads 0, pag
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page se
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, page ser

(1 row affected)

 SQL Server Execution Times:
   CPU time = 297 ms,  elapsed time = 455 ms.

(24956 rows affected)
Table 'itemsFactura'. Scan count 1, logical reads 771, physical reads 0, pa
Table 'facturas'. Scan count 1, logical reads 215, physical reads 0, page s
Table 'itemsProductos'. Scan count 1, logical reads 1175, physical reads 0,
Table 'lotesProduccionLogs'. Scan count 1, logical reads 246, physical read
Table 'tiposDeCambio'. Scan count 1, logical reads 2, physical reads 0, pag
Table 'porcentajesActores'. Scan count 1, logical reads 129, physical reads
Table 'actoresContratoProd'. Scan count 1, logical reads 84, physical reads
Table 'productores'. Scan count 2, logical reads 4, physical reads 0, page
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page se

(1 row affected)

 SQL Server Execution Times:
   CPU time = 266 ms,  elapsed time = 371 ms.
```

Unidad de workload	Explicación	Norma
Hash Match Use each row from the top input to build a hash table, and each row from the bottom input to probe into the hash table, outputting all matching rows. **Physical Operation** Hash Match **Logical Operation** Inner Join **Actual Execution Mode** Batch **Estimated Execution Mode** Batch **Actual Number of Rows for All Executions** 174198 **Actual Number of Batches** 194 **Estimated Operator Cost** 0.139242 (3%) **Estimated I/O Cost** 0 **Estimated Subtree Cost** 1.10105 **Estimated CPU Cost** 0.139235 **Estimated Number of Executions** 1 **Number of Executions** 1 **Estimated Number of Rows for All Executions** 20000 **Estimated Number of Rows Per Execution** 20000 **Estimated Row Size** 31 B **Actual Rebinds** 0 **Actual Rewinds** 0 **Node ID** 17 **Output List** [evtest].[dbo].[facturas].facturaId, [evtest].[dbo].[facturas].fecha, [evtest].[dbo].[itemsFactura].itemId **Hash Keys Probe** [evtest].[dbo].[itemsFactura].facturaId **Probe Residual** [evtest].[dbo].[itemsFactura].[facturaId]=[evtest].[dbo].[facturas].[facturaId]	Aquí hace un Hash Match (J4) con base en el campo de facturaId en las tablas de itemsFactura y facturas.	Buscar que las columnas involucradas en la igualdad para el Hash Match estén ordenadas, ya sea por un Sort, un clustered index o un nonclustered index para intentar convertirlo en un Merge Join (J3)

Se agregó un Clustered Index en las columnas de facturaId y itemId de la tabla itemsFactura. La tabla de facturas ya tiene un clustered index en facturaId.



En comparación con la versión anterior, el query tiene un porcentaje de costo menor en general.



El método para el join pasó de un Hash Match a un Merge Join. Esto ocurrió porque las dos tablas están ordenadas con base en el clustered index en facturaId, por lo que puede utilizar el método J3.

Merge Join
Match rows from two suitably sorted input tables exploiting their sort order.

Physical Operation	Merge Join
Logical Operation	Inner Join
Actual Execution Mode	Row
Estimated Execution Mode	Row
Actual Number of Rows for All Executions	174198
Actual Number of Batches	194
Estimated Operator Cost	0.070643 (1%)
Estimated I/O Cost	0
Estimated Subtree Cost	0.885785
Estimated CPU Cost	0.0706
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows for All Executions	20000
Estimated Number of Rows Per Execution	20000
Estimated Row Size	31 B
Actual Rebinds	0
Actual Rewinds	0
Many to Many	False
Node ID	17

Where (join columns)
([evtest].[dbo].[itemsFactura].facturaId) = ([evtest].[dbo].[facturas].facturaId)
Output List
[evtest].[dbo].[facturas].facturaId, [evtest].[dbo].[facturas].fecha, [evtest].[dbo].[itemsFactura].itemId
Residual
[evtest].[dbo].[itemsFactura].[facturaId]=[evtest].[dbo].[facturas].[facturaId]

No obstante, en cuanto al rendimiento, no observamos una mejoría en el tiempo de ejecución.

Sin índice:

```
(24956 rows affected)

(1 row affected)

 SQL Server Execution Times:
   CPU time = 312 ms,  elapsed time = 438 ms.

Completion time: 2023-04-30T17:12:26.5609010-06:00
```

Con índice:

```
(24956 rows affected)

(1 row affected)

 SQL Server Execution Times:
   CPU time = 359 ms,  elapsed time = 462 ms.

Completion time: 2023-04-30T17:11:55.3108765-06:00
```

Sin embargo, es posible que, con una mayor cantidad de datos, la eficiencia del Merge Join nos beneficie y haga la consulta más rápidamente.

En la siguiente comparación, se aprecia que con el índice se requieren menos operaciones de I/O en la tabla de itemsFactura, de 574 versus 771.

```
(24956 rows affected)
Table 'itemsFactura'. Scan count 1, logical reads 771, physical reads 0, page server reads 0,
Table 'facturas'. Scan count 1, logical reads 215, physical reads 0, page server reads 0, read
Table 'itemsProductos'. Scan count 1, logical reads 1175, physical reads 0, page server reads
Table 'lotesProduccionLogs'. Scan count 1, logical reads 246, physical reads 0, page server re
Table 'tiposDeCambio'. Scan count 1, logical reads 2, physical reads 0, page server reads 0, r
Table 'porcentajesActores'. Scan count 1, logical reads 129, physical reads 0, page server rea
Table 'actoresContratoProd'. Scan count 1, logical reads 84, physical reads 0, page server rea
Table 'productores'. Scan count 2, logical reads 4, physical reads 0, page server reads 0, rea
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-

(1 row affected)

 SQL Server Execution Times:
   CPU time = 234 ms,  elapsed time = 366 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 40 ms, elapsed time = 40 ms.

(24956 rows affected)
Table 'itemsFactura'. Scan count 1, logical reads 574, physical reads 0, page server reads 0,
Table 'itemsProductos'. Scan count 1, logical reads 1175, physical reads 0, page server reads
Table 'lotesProduccionLogs'. Scan count 1, logical reads 246, physical reads 0, page server re
Table 'tiposDeCambio'. Scan count 1, logical reads 2, physical reads 0, page server reads 0, r
Table 'porcentajesActores'. Scan count 1, logical reads 129, physical reads 0, page server rea
Table 'actoresContratoProd'. Scan count 1, logical reads 84, physical reads 0, page server rea
Table 'productores'. Scan count 2, logical reads 4, physical reads 0, page server reads 0, rea
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-
Table 'facturas'. Scan count 1, logical reads 215, physical reads 0, page server reads 0, read

(1 row affected)

 SQL Server Execution Times:
   CPU time = 266 ms,  elapsed time = 557 ms.

Completion time: 2023-04-30T17:34:26.7436232-06:00
```

Unidad de workload	Explicación	Norma
Clustered Index Scan (Clustered) Scanning a clustered index, entirely or only a range. **Physical Operation**: Clustered Index Scan **Logical Operation**: Clustered Index Scan **Actual Execution Mode**: Batch **Estimated Execution Mode**: Batch **Storage**: RowStore **Actual Number of Rows Read**: 20000 **Actual Number of Rows for All Executions**: 17389 **Actual Number of Batches**: 21 **Estimated I/O Cost**: 0.183125 **Estimated Operator Cost**: 0.205262 (4%) **Estimated Subtree Cost**: 0.205262 **Estimated CPU Cost**: 0.022137 **Estimated Number of Executions**: 1 **Number of Executions**: 1 **Estimated Number of Rows for All Executions**: 1858.3 **Estimated Number of Rows Per Execution**: 1858.3 **Estimated Number of Rows to be Read**: 20000 **Estimated Row Size**: 19 B **Actual Rebinds**: 0 **Actual Rewinds**: 0 **Ordered**: False **Node ID**: 14 **Predicate** [evtest].[dbo].[lotesProduccionLogs].[productoId] as [lpl].[productoId] <> (2) AND PROBE([Opt_Bitmap1038],[evtest].[dbo].[lotesProduccionLogs].[productoId] as [lpl].[productoId]) AND PROBE([Opt_Bitmap1038],[evtest].[dbo].[lotesProduccionLogs].[prodContratoId] as [lpl].[prodContratoId]) **Object** [evtest].[dbo].[lotesProduccionLogs].[PK_productosFabricados] [lpl] **Output List** [evtest].[dbo].[lotesProduccionLogs].loteId, [evtest].[dbo].[lotesProduccionLogs].productoId, [evtest].[dbo].[lotesProduccionLogs].prodContratoId	Aquí hace un Clustered Index Scan (S7) sobre la tabla de lotesProduccionLogs usando el primary key PK_productosFabricados. Luego usa los otros elementos de la condición conjuntiva para seleccionar registros específicos.	Revisar cuáles son las columnas involucradas en el predicate. Agregar un nonclustered index en las columnas del PROBE o igualdades para que las pueda filtrar más rápidamente. Poner columnas diferentes en el output list como included columns en el index mientras no sean muchas.

Index Seek (NonClustered)
Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	RowStore
Actual Number of Rows Read	18583
Actual Number of Rows for All Executions	17389
Actual Number of Batches	20
Estimated Operator Cost	0.0540751 (1%)
Estimated I/O Cost	0.0334954
Estimated Subtree Cost	0.0540751
Estimated CPU Cost	0.0205797
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows for All Executions	1858.3
Estimated Number of Rows to be Read	18583
Estimated Number of Rows Per Execution	1858.3
Estimated Row Size	19 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	14

Predicate
PROBE([Opt_Bitmap1040],[evtest].[dbo].[lotesProduccionLogs].[productoId] as [lpl].[productoId]) AND PROBE([Opt_Bitmap1040],[evtest].[dbo].[lotesProduccionLogs].[prodContratoId] as [lpl].[prodContratoId])
Object
[evtest].[dbo].[lotesProduccionLogs].[IX_lotesProduccionLogs_productoIdcontratoId] [lpl]
Output List
[evtest].[dbo].[lotesProduccionLogs].loteId, [evtest].[dbo].[lotesProduccionLogs].productoId, [evtest].[dbo].[lotesProduccionLogs].prodContratoId
Seek Predicates
[1] Seek Keys[1]: End: [evtest].[dbo].[lotesProduccionLogs].productoId < Scalar Operator((2)), [2] Seek Keys[1]: Start: [evtest].[dbo].[lotesProduccionLogs].productoId > Scalar Operator((2))

Aplicando la norma, se agrega un nonclustered index en las columnas de productoId y prodContratoId, las cuales están en las cláusulas de PROBE en el Predicate. Esto hace que pueda recorrer el índice para encontrar las filas que ocupa en lugar de recorrer todos los registros de la tabla. Por esta razón, el operador cambia a un Index Seek (Nonclustered) y lo hace más eficientemente.

Esto hace por ejemplo, que las lecturas en la tabla de lotesProduccionLogs bajen de 246 a 47. El tiempo de ejecución también es menor.

```
(24956 rows affected)
Table 'itemsFactura'. Scan count 1, logical reads 574, physical reads 1, page server reads 0
Table 'itemsProductos'. Scan count 1, logical reads 1175, physical reads 3, page server read
Table 'lotesProduccionLogs'. Scan count 1, logical reads 246, physical reads 1, page server
Table 'tiposDeCambio'. Scan count 1, logical reads 2, physical reads 1, page server reads 0,
Table 'porcentajesActores'. Scan count 1, logical reads 129, physical reads 1, page server r
Table 'actoresContratoProd'. Scan count 1, logical reads 84, physical reads 1, page server r
Table 'productores'. Scan count 2, logical reads 4, physical reads 1, page server reads 0, r
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, rea
Table 'facturas'. Scan count 1, logical reads 215, physical reads 1, page server reads 0, re

(1 row affected)

 SQL Server Execution Times:
   CPU time = 266 ms,  elapsed time = 416 ms.
SQL Server parse and compile time:
   CPU time = 46 ms, elapsed time = 46 ms.

(24956 rows affected)
Table 'itemsFactura'. Scan count 1, logical reads 574, physical reads 0, page server reads 0
Table 'itemsProductos'. Scan count 1, logical reads 1175, physical reads 0, page server read
Table 'lotesProduccionLogs'. Scan count 2, logical reads 47, physical reads 0, page server r
Table 'tiposDeCambio'. Scan count 1, logical reads 2, physical reads 0, page server reads 0,
Table 'porcentajesActores'. Scan count 1, logical reads 129, physical reads 0, page server r
Table 'actoresContratoProd'. Scan count 1, logical reads 84, physical reads 0, page server r
Table 'productores'. Scan count 2, logical reads 4, physical reads 0, page server reads 0, r
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, rea
Table 'facturas'. Scan count 1, logical reads 215, physical reads 0, page server reads 0, re

(1 row affected)

 SQL Server Execution Times:
   CPU time = 265 ms,  elapsed time = 410 ms.

Completion time: 2023-04-30T18:11:46.3241591-06:00
```

Unidad de workload	Explicación	Norma
Clustered Index Scan (Clustered) Scanning a clustered index, entirely or only a range. **Physical Operation**: Clustered Index Scan **Logical Operation**: Clustered Index Scan **Actual Execution Mode**: Batch **Estimated Execution Mode**: Batch **Storage**: RowStore **Actual Number of Rows Read**: 10000 **Actual Number of Rows for All Executions**: 3337 **Actual Number of Batches**: 4 **Estimated I/O Cost**: 0.063125 **Estimated Operator Cost**: 0.074282 (2%) **Estimated Subtree Cost**: 0.074282 **Estimated CPU Cost**: 0.011157 **Estimated Number of Executions**: 1 **Number of Executions**: 1 **Estimated Number of Rows for All Executions**: 3337 **Estimated Number of Rows Per Execution**: 3337 **Estimated Number of Rows to be Read**: 10000 **Estimated Row Size**: 20 B **Actual Rebinds**: 0 **Actual Rewinds**: 0 **Ordered**: False **Node ID**: 6 **Predicate** [evtest].[dbo].[actoresContratoProd].[objectTypeId] as [actores].[objectTypeId]=(7) **Object** [evtest].[dbo].[actoresContratoProd].[PK_actoresContratoProd] [actores] **Output List** [evtest].[dbo].[actoresContratoProd].prodContratoId, [evtest].[dbo].[actoresContratoProd].actorId, [evtest].[dbo].[actoresContratoProd].genericId	Aquí hace un Clustered Index Scan (S7) sobre la tabla de actoresContratoProd usando el primary key PK_actoresContratoProd. Luego usa los otros elementos de la condición conjuntiva (si el objectTypeId = 7) para seleccionar registros específicos.	Revisar cuáles son las columnas involucradas en el predicate. Agregar un nonclustered index en las columnas del PROBE o que están en una condición y son non-key para que las pueda filtrar más rápidamente. Poner columnas diferentes en el output list como included columns en el index, mientras no sean muchas.

Index Seek (NonClustered)
Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	RowStore
Actual Number of Rows Read	3337
Actual Number of Rows for All Executions	3337
Actual Number of Batches	4
Estimated Operator Cost	0.0121444 (0%)
Estimated I/O Cost	0.0083167
Estimated Subtree Cost	0.0121444
Estimated CPU Cost	0.0038277
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows for All Executions	3337
Estimated Number of Rows to be Read	3337
Estimated Number of Rows Per Execution	3337
Estimated Row Size	19 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	6

Object
[evtest].[dbo].[actoresContratoProd].[IX_actoresContratoProd_objectTypeId] [actores]
Output List
[evtest].[dbo].[actoresContratoProd].prodContratoId, [evtest].[dbo].[actoresContratoProd].actorId, [evtest].[dbo].[actoresContratoProd].genericId
Seek Predicates
Seek Keys[1]: Prefix: [evtest].[dbo].[actoresContratoProd].objectTypeId = Scalar Operator((7))

Se agregó un nonclustered index en la tabla actoresContratoProd sobre objectTypeId con prodContratoId, actorId y genericId como included columns. Esto convirtió el operador a un Index Seek (Nonclustered). Como en los casos anteriores, puede usar el índice directamente para encontrar los que tengan un objectTypeId = 7 y extraer los valores que ocupa directo del índice. Así, el peso del operador es menor.

Con este índice, pasa de realizar 84 lecturas lógicas a solo 10 lecturas en la tabla de actoresContratoProd. El tiempo de ejecución también es menor con el índice.

```
(24956 rows affected)
Table 'itemsFactura'. Scan count 1, logical reads 574, physical reads 1, page server reads 0, read-a
Table 'itemsProductos'. Scan count 1, logical reads 1175, physical reads 3, page server reads 0, rea
Table 'lotesProduccionLogs'. Scan count 2, logical reads 47, physical reads 1, page server reads 0,
Table 'tiposDeCambio'. Scan count 1, logical reads 2, physical reads 1, page server reads 0, read-ah
Table 'porcentajesActores'. Scan count 1, logical reads 129, physical reads 1, page server reads 0,
Table 'actoresContratoProd'. Scan count 1, logical reads 84, physical reads 1, page server reads 0,
Table 'productores'. Scan count 2, logical reads 4, physical reads 1, page server reads 0, read-ahea
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead
Table 'facturas'. Scan count 1, logical reads 215, physical reads 1, page server reads 0, read-ahead

(1 row affected)

 SQL Server Execution Times:
   CPU time = 313 ms,  elapsed time = 464 ms.
SQL Server parse and compile time:
   CPU time = 42 ms, elapsed time = 42 ms.

(24956 rows affected)
Table 'itemsFactura'. Scan count 1, logical reads 574, physical reads 0, page server reads 0, read-a
Table 'itemsProductos'. Scan count 1, logical reads 1175, physical reads 0, page server reads 0, rea
Table 'lotesProduccionLogs'. Scan count 2, logical reads 47, physical reads 0, page server reads 0,
Table 'tiposDeCambio'. Scan count 1, logical reads 2, physical reads 0, page server reads 0, read-ah
Table 'porcentajesActores'. Scan count 1, logical reads 129, physical reads 0, page server reads 0,
Table 'actoresContratoProd'. Scan count 1, logical reads 10, physical reads 0, page server reads 0,
Table 'productores'. Scan count 2, logical reads 4, physical reads 0, page server reads 0, read-ahea
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead
Table 'facturas'. Scan count 1, logical reads 215, physical reads 0, page server reads 0, read-ahead

(1 row affected)

 SQL Server Execution Times:
   CPU time = 281 ms,  elapsed time = 393 ms.

Completion time: 2023-04-30T18:20:20.7206914-06:00
```

Unidad de workload	Explicación	Norma
Clustered Index Scan (Clustered) Scanning a clustered index, entirely or only a range. **Physical Operation**: Clustered Index Scan **Logical Operation**: Clustered Index Scan **Actual Execution Mode**: Batch **Estimated Execution Mode**: Batch **Storage**: RowStore **Actual Number of Rows Read**: 10000 **Actual Number of Rows for All Executions**: 3337 **Actual Number of Batches**: 4 **Estimated I/O Cost**: 0.0964583 **Estimated Operator Cost**: 0.107615 (2%) **Estimated Subtree Cost**: 0.107615 **Estimated CPU Cost**: 0.011157 **Estimated Number of Executions**: 1 **Number of Executions**: 1 **Estimated Number of Rows for All Executions**: 10000 **Estimated Number of Rows Per Execution**: 10000 **Estimated Number of Rows to be Read**: 10000 **Estimated Row Size**: 20 B **Actual Rebinds**: 0 **Actual Rewinds**: 0 **Ordered**: False **Node ID**: 8 **Predicate** PROBE([Opt_Bitmap1035],[evtest].[dbo].[porcentajesActores].[actorId] as [porcentajes].[actorId]) **Object** [evtest].[dbo].[porcentajesActores].[PK_porcentajesActores] [porcentajes] **Output List** [evtest].[dbo].[porcentajesActores].actorId, [evtest].[dbo].[porcentajesActores].porcentaje, [evtest].[dbo].[porcentajesActores].productoId	Aquí hace un Clustered Index Scan (S7) sobre la tabla de porcentajesActores usando el primary key PK_porcentajesActores. Luego extrae la columna de actorId como Probe para el Hash Match que va a hacer después.	Revisar cuáles son las columnas involucradas en el predicate. Agregar un nonclustered index en las columnas del PROBE o que están en una condición y son non-key para que las pueda filtrar más rápidamente. Poner columnas diferentes en el output list como included columns en el index, mientras no sean muchas.

Index Scan (NonClustered)
Scan a nonclustered index, entirely or only a range.

Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows Read	10000
Actual Number of Rows for All Executions	10000
Actual Number of Batches	0
Estimated I/O Cost	0.0238657
Estimated Operator Cost	0.0350227 (1%)
Estimated CPU Cost	0.011157
Estimated Subtree Cost	0.0350227
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows for All Executions	10000
Estimated Number of Rows Per Execution	10000
Estimated Number of Rows to be Read	10000
Estimated Row Size	20 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	7

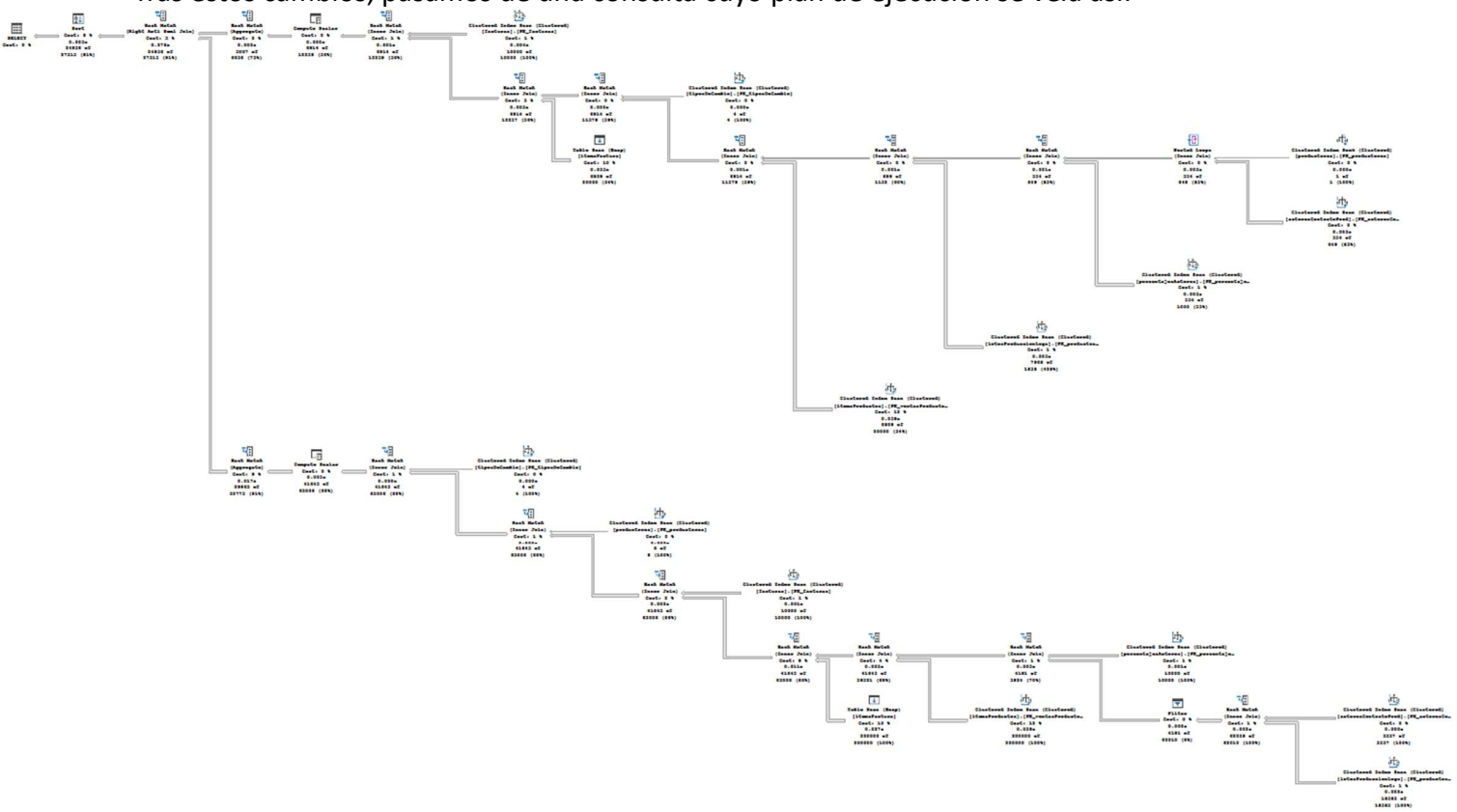
Object
[evtest].[dbo].[porcentajesActores].[IX_porcentajesActores_actorId] [porcentajes]
Output List
[evtest].[dbo].[porcentajesActores].actorId, [evtest].[dbo].[porcentajesActores].porcentaje, [evtest].[dbo].[porcentajesActores].productoId

Se agregó un nonclustered index en la tabla porcentajesActores sobre actorId con porcentaje y productoId como included columns. Esto cambió el operador a un Index Scan (NonClustered). No es un index seek porque no tenemos una condición de igualdad. No obstante, el resultado es una operación más rápida porque la búsqueda en el árbol B es más rápida. Adicionalmente, tras esta conversión, el Hash Match que venía después se transformó en un Merge Join debido a que el campo donde se hace el join, el cual es actorId, está ordenado en ambas tablas por medio de índices.

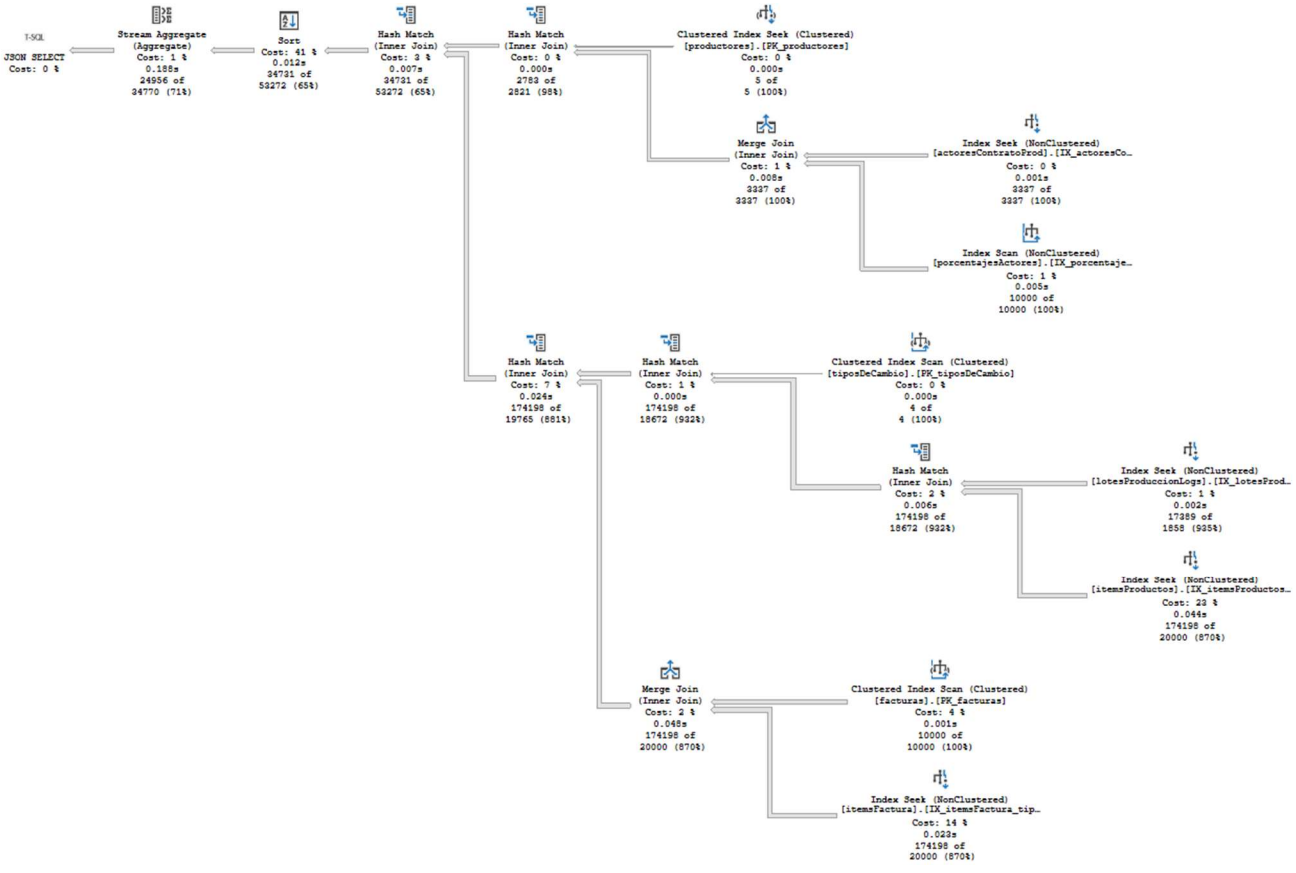
Este índice permite que las lecturas de la tabla porcentajesActores bajaran de 129 a 31. El tiempo de ejecución también disminuyó.

```
(24956 rows affected)
Table 'itemsFactura'. Scan count 1, logical reads 574, physical reads 1, page server reads
Table 'itemsProductos'. Scan count 1, logical reads 1175, physical reads 3, page server re
Table 'lotesProduccionLogs'. Scan count 2, logical reads 47, physical reads 1, page serve
Table 'tiposDeCambio'. Scan count 1, logical reads 2, physical reads 1, page server reads
Table 'porcentajesActores'. Scan count 1, logical reads 129, physical reads 1, page serve
Table 'actoresContratoProd'. Scan count 1, logical reads 10, physical reads 1, page serve
Table 'productores'. Scan count 2, logical reads 4, physical reads 1, page server reads 0,
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0,
Table 'facturas'. Scan count 1, logical reads 215, physical reads 1, page server reads 0,

(1 row affected)

 SQL Server Execution Times:
   CPU time = 297 ms,  elapsed time = 453 ms.
SQL Server parse and compile time:
   CPU time = 46 ms, elapsed time = 46 ms.

(24956 rows affected)
Table 'itemsFactura'. Scan count 1, logical reads 574, physical reads 0, page server reads
Table 'itemsProductos'. Scan count 1, logical reads 1175, physical reads 0, page server re
Table 'lotesProduccionLogs'. Scan count 2, logical reads 47, physical reads 0, page serve
Table 'tiposDeCambio'. Scan count 1, logical reads 2, physical reads 0, page server reads
Table 'productores'. Scan count 2, logical reads 4, physical reads 0, page server reads 0,
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0,
Table 'facturas'. Scan count 1, logical reads 215, physical reads 0, page server reads 0,
Table 'porcentajesActores'. Scan count 1, logical reads 31, physical reads 0, page server
Table 'actoresContratoProd'. Scan count 1, logical reads 10, physical reads 0, page serve

(1 row affected)

 SQL Server Execution Times:
   CPU time = 281 ms,  elapsed time = 406 ms.

Completion time: 2023-04-30T18:45:40.1030410-06:00
```

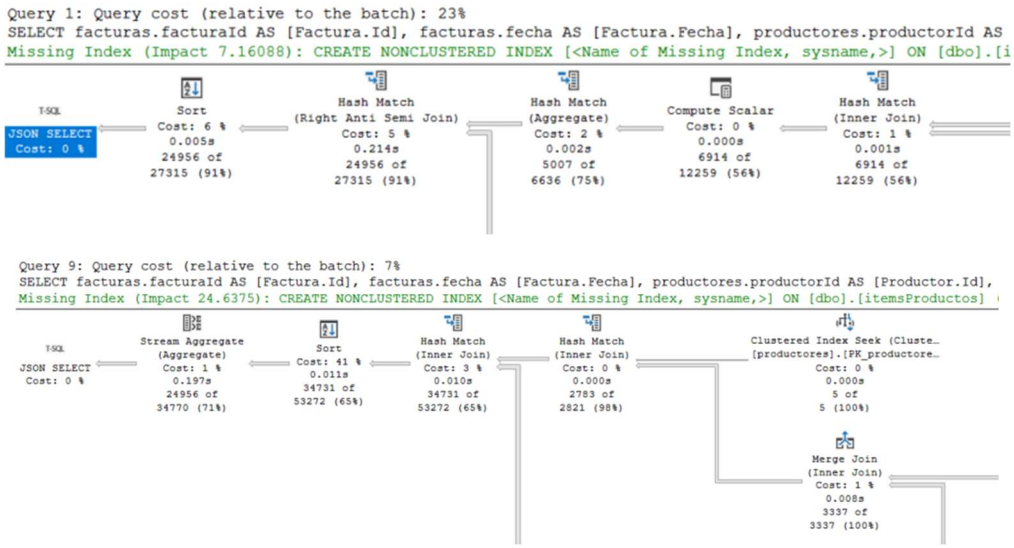
Tras estos cambios, pasamos de una consulta cuyo plan de ejecución se veía así:



A una consulta cuyo plan de ejecución se ve así:



Comparando los dos: tenemos que la primera versión tiene un costo 3 veces mayor que la segunda.



En cuanto al tiempo de ejecución y operaciones de I/O, la segunda versión es más rápida y tiene muchas menos operaciones de I/O:

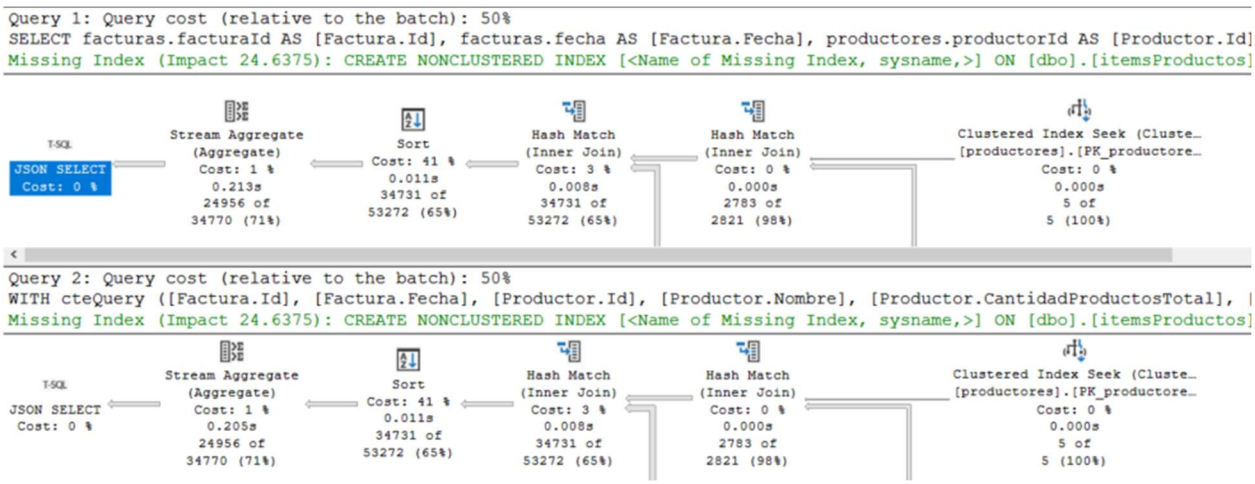
```
(24956 rows affected)
Table 'itemsFactura'. Scan count 2, logical reads 3578, physical reads 0, page se
Table 'itemsProductos'. Scan count 2, logical reads 4676, physical reads 2, page
Table 'lotesProduccionLogs'. Scan count 2, logical reads 492, physical reads 1, p
Table 'actoresContratoProd'. Scan count 2, logical reads 168, physical reads 1, p
Table 'porcentajesActores'. Scan count 2, logical reads 258, physical reads 1, pa
Table 'facturas'. Scan count 2, logical reads 430, physical reads 1, page server
Table 'productores'. Scan count 1, logical reads 4, physical reads 1, page server
Table 'tiposDeCambio'. Scan count 2, logical reads 4, physical reads 1, page serv
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server r
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, page server re

(1 row affected)

 SQL Server Execution Times:
   CPU time = 281 ms,  elapsed time = 544 ms.

Completion time: 2023-05-02T18:53:08.9927705-06:00


(24956 rows affected)
Table 'itemsFactura'. Scan count 1, logical reads 574, physical reads 0, page s
Table 'itemsProductos'. Scan count 1, logical reads 1175, physical reads 0, pag
Table 'lotesProduccionLogs'. Scan count 2, logical reads 47, physical reads 0,
Table 'tiposDeCambio'. Scan count 1, logical reads 2, physical reads 0, page se
Table 'productores'. Scan count 2, logical reads 4, physical reads 0, page serv
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server
Table 'facturas'. Scan count 1, logical reads 215, physical reads 0, page serve
Table 'porcentajesActores'. Scan count 1, logical reads 31, physical reads 0, p
Table 'actoresContratoProd'. Scan count 1, logical reads 10, physical reads 0,

(1 row affected)

 SQL Server Execution Times:
   CPU time = 329 ms,  elapsed time = 426 ms.

Completion time: 2023-05-02T18:54:17.5761389-06:00
```

Common Table Expression

Encapsulando el query optimizado en un CTE:

```
WITH cteQuery ([Factura.Id], [Factura.Fecha], [Productor.Id], [Productor.Nombre], [Productor.CantidadProductosTotal],
[Productor.DineroTotalProductor]) AS
(
    SELECT facturas.facturaId AS [Factura.Id], facturas.fecha AS [Factura.Fecha],
    productores.productorId AS [Productor.Id], productores.nombre AS [Productor.Nombre],
    SUM(items.cantidadProductos) AS [Productor.CantidadProductosTotal],
    SUM((items.montoTotal/tiposDeCambio.conversion) * porcentajes.porcentaje / 100) AS [Productor.DineroTotalProductor]
    FROM facturas LEFT JOIN itemsFactura ON facturas.facturaId = itemsFactura.facturaId
    INNER JOIN itemsProductos items ON items.itemProdId = itemsFactura.itemId
    INNER JOIN lotesProduccionLogs lpl ON items.loteId = lpl.loteId
    INNER JOIN contratosProduccion contProd ON contProd.prodContratoId = lpl.prodContratoId
    INNER JOIN actoresContratoProd actores ON actores.prodContratoId = contProd.prodContratoId
    INNER JOIN porcentajesActores porcentajes ON porcentajes.actorId = actores.actorId
    RIGHT JOIN productores ON productores.productorId = actores.genericId
    INNER JOIN tiposDeCambio ON tiposDeCambio.monedaCambioId = items.monedaId
    WHERE items.fecha BETWEEN '2022-01-01 00:00:00' AND GETDATE() AND
    itemsFactura.tipoItemId = 3 AND -- si es un item de venta de producto
    actores.objectTypeId = 7 AND -- si el actor es un productor
    porcentajes.productoId = lpl.productoId AND -- si el actor tiene un porcentaje del producto, participó en la producción de ese producto
    lpl.productoId != 2 AND-- si la venta no involucra el producto 2.
    productores.productorId != 5 -- se sustitye el except por una desigualdad
    GROUP BY facturas.facturaId, facturas.fecha, actores.genericId, productores.productorId, productores.nombre
)
SELECT * FROM cteQuery
    ORDER BY [Factura.Id]
    FOR JSON PATH, ROOT ('Facturas')
```

No se percibe una mejora en rendimiento encapsulando la consulta en un CTE. Más bien, a veces el tiempo de ejecución es mayor, pero esto es muy variable. Las operaciones de lectura y escritura son iguales. Los porcentajes de costos son los mismos:




```
(24956 rows affected)
Table 'itemsFactura'. Scan count 1, logical reads 574, physical reads 0, p
Table 'itemsProductos'. Scan count 1, logical reads 1175, physical reads (
Table 'lotesProduccionLogs'. Scan count 2, logical reads 47, physical reac
Table 'tiposDeCambio'. Scan count 1, logical reads 2, physical reads 0, p
Table 'productores'. Scan count 2, logical reads 4, physical reads 0, page
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page
Table 'facturas'. Scan count 1, logical reads 215, physical reads 0, page
Table 'porcentajesActores'. Scan count 1, logical reads 31, physical reads
Table 'actoresContratoProd'. Scan count 1, logical reads 10, physical reac

(1 row affected)

 SQL Server Execution Times:
   CPU time = 312 ms,  elapsed time = 425 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 45 ms, elapsed time = 45 ms.
```

← Query sin CTE

```
(24956 rows affected)
Table 'itemsFactura'. Scan count 1, logical reads 574, physical reads 0, p
Table 'itemsProductos'. Scan count 1, logical reads 1175, physical reads (
Table 'lotesProduccionLogs'. Scan count 2, logical reads 47, physical reac
Table 'tiposDeCambio'. Scan count 1, logical reads 2, physical reads 0, p
Table 'productores'. Scan count 2, logical reads 4, physical reads 0, page
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page
Table 'facturas'. Scan count 1, logical reads 215, physical reads 0, page
Table 'porcentajesActores'. Scan count 1, logical reads 31, physical reads
Table 'actoresContratoProd'. Scan count 1, logical reads 10, physical reac

(1 row affected)

 SQL Server Execution Times:
   CPU time = 313 ms,  elapsed time = 436 ms.

Completion time: 2023-05-02T19:53:39.4780150-06:00
```

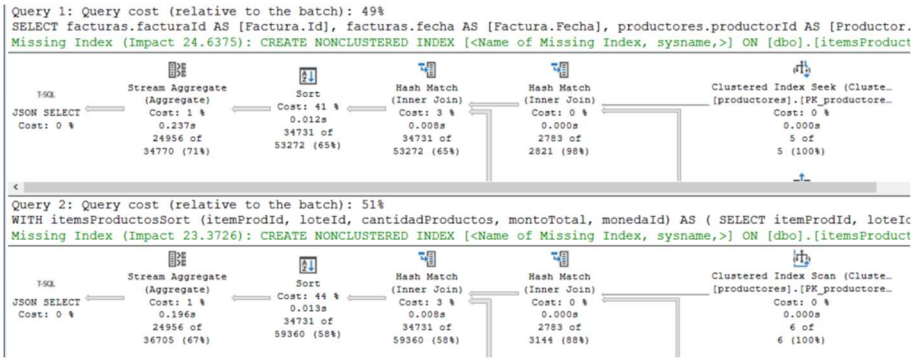
← Query con CTE

Query simplificada con CTEs:

```
WITH itemsProductosSort (itemProdId, loteId, cantidadProductos, montoTotal, monedaId) AS
(
    SELECT itemProdId, loteId, cantidadProductos, montoTotal, monedaId FROM itemsProductos
    WHERE fecha BETWEEN '2022-01-01 00:00:00' AND GETDATE()
),
ventasFacturas (facturaId, itemId) AS
(
    SELECT facturaId, itemId FROM itemsFactura
    WHERE itemsFactura.tipoItemId = 3
),
lotesProductos (loteId, prodContratoId, productoId) AS
(
    SELECT loteId, prodContratoId, productoId FROM lotesProduccionLogs
    WHERE lotesProduccionLogs.productoId != 2
),
actores (actorId, prodContratoId, genericId) AS
(
    SELECT actorId, prodContratoId, genericId FROM actoresContratoProd
    WHERE objectTypeId = 7 AND genericId != 5
)
SELECT facturas.facturaId AS [Factura.Id], facturas.fecha AS [Factura.Fecha],
productores.productorId AS [Productor.Id], productores.nombre AS [Productor.Nombre],
SUM(itemsProductosSort.cantidadProductos) AS [Productor.CantidadProductosTotal],
SUM((itemsProductosSort.montoTotal/tiposDeCambio.conversion) * porcentajes.porcentaje / 100) AS [Productor.DineroTotalProductor]
FROM facturas
INNER JOIN ventasFacturas ON ventasFacturas.facturaId = facturas.facturaId
INNER JOIN itemsProductosSort ON itemsProductosSort.itemProdId = ventasFacturas.itemId
INNER JOIN lotesProductos ON lotesProductos.loteId = itemsProductosSort.loteId
INNER JOIN contratosProduccion ON contratosProduccion.prodContratoId = lotesProductos.prodContratoId
INNER JOIN actores ON actores.prodContratoId = contratosProduccion.prodContratoId
INNER JOIN porcentajesActores porcentajes ON porcentajes.actorId = actores.actorId
INNER JOIN productores ON productores.productorId = actores.genericId
INNER JOIN tiposDeCambio ON tiposDeCambio.monedaCambioId = itemsProductosSort.monedaId
WHERE porcentajes.productoId = lotesProductos.productoId
GROUP BY facturas.facturaId, facturas.fecha, actores.genericId, productores.productorId, productores.nombre
ORDER BY [Factura.Id]
FOR JSON PATH, ROOT ('Facturas');
```

En este query, realizamos el filtro antes de hacer los joins para trabajar directamente con la información correcta. Así se simplifica las conjunciones en el WHERE del SELECT principal.

En este caso, el costo del query con CTEs es ligeramente mayor:




```
(24956 rows affected)
Table 'itemsFactura'. Scan count 1, logical reads 574, physical reads
Table 'itemsProductos'. Scan count 1, logical reads 1175, physical rea
Table 'lotesProduccionLogs'. Scan count 2, logical reads 47, physical
Table 'tiposDeCambio'. Scan count 1, logical reads 2, physical reads
Table 'productores'. Scan count 2, logical reads 4, physical reads 1,
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, pa
Table 'facturas'. Scan count 1, logical reads 215, physical reads 1, p
Table 'porcentajesActores'. Scan count 1, logical reads 31, physical r
Table 'actoresContratoProd'. Scan count 1, logical reads 10, physical

(1 row affected)

 SQL Server Execution Times:
   CPU time = 375 ms,  elapsed time = 496 ms.

(24956 rows affected)
Table 'itemsFactura'. Scan count 1, logical reads 574, physical reads
Table 'itemsProductos'. Scan count 1, logical reads 1175, physical rea
Table 'lotesProduccionLogs'. Scan count 2, logical reads 47, physical
Table 'tiposDeCambio'. Scan count 1, logical reads 2, physical reads (
Table 'porcentajesActores'. Scan count 1, logical reads 31, physical r
Table 'actoresContratoProd'. Scan count 1, logical reads 10, physical
Table 'productores'. Scan count 1, logical reads 2, physical reads 0,
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, pa
Table 'facturas'. Scan count 1, logical reads 215, physical reads 0, p
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, pa

(1 row affected)

 SQL Server Execution Times:
   CPU time = 265 ms,  elapsed time = 407 ms.

Completion time: 2023-05-02T20:14:52.2966687-06:00
```

La cantidad de operaciones de lectura en las tablas se mantiene igual.

No obstante, sí se percibe una ligera mejora en cuanto al tiempo de ejecución, por lo que sí es recomendado intentar simplificar el query por medio de CTEs. Adicionalmente, pueden ayudar a mejorar la legibilidad y comprensión del query:

Resumen de la norma:

1. Cuando se hace un Clustered Index Scan y se filtra por una llave non-key, revisar el campo non-key que está en el Predicate y agregar un NonClustered Index. Incluso, se pueden agregar las columnas que aparecen en el Output List, mientras no sean muchas, para extraer la información directamente.

2. Cuando hay que hacer una búsqueda secuencial para buscar dónde se cumple la condición en una columna, indexar la columna problema con un nonclustered index para hacer una búsqueda S5 donde a partir de la condición, extrae todos los records que la cumplen. Adicionalmente, las columnas que tiene que extraer esos registros pueden incluirse como non-key included columns en el índice para obtenerlas directamente con el índice y así reducir las operaciones de I/O.

3. Si la consulta contiene un EXCEPT o INTERSECT que opera sobre consultas muy similares, intentar eliminar ese EXCEPT por medio de igualdades, desigualdades, o algún otro medio.
4. Buscar que las columnas involucradas en la igualdad para el Hash Match estén ordenadas, ya sea por un Sort, un clustered index o un nonclustered index para intentar convertirlo en un Merge Join (J3)
5. Revisar cuáles son las columnas involucradas en el predicate. Agregar un nonclustered index en las columnas del PROBE o que están en una condición y son non-key para que las pueda filtrar más rápidamente. Poner columnas diferentes del output list como included columns en el index, mientras no sean muchas.
6. Intentar simplificar el query por medio de CTEs para obtener una ligera mejora en el rendimiento.
7. En cuanto a vistas, si la tabla va a cambiar mucho, es mejor usar una vista dinámica para recalcular el SELECT cada vez que se usa la vista. Si no ocupo consultar la misma versión de los datos frecuentemente, es mejor usar una vista indexada o estática para facilitar el acceso a la información, ya que esta se guarda.

Bibliografía:

Yaseen, A. (2019). SQL Server indexed views. SQL Shack - articles about database auditing, server performance, data recovery, and more. https://www.sqlshack.com/sql-server-indexed-views/