



# UNIVERSIDAD NACIONAL DEL NORDESTE



FACULTAD DE CIENCIAS  
EXACTAS Y NATURALES  
Y AGRIMENSURA

## Tema de Investigación:

Indices Columnares en SQL  
Server

### Grupo: 3

Escobar, Esteban E.

L.U: 44523

Fernández, R. Oscar

L.U: 33539

Micheloud, Matias A.

L.U: 56606

Romero, R. Antonio

L.U:44212381

# ÍNDICE

ÍNDICE.....	1
CAPÍTULO I: INTRODUCCIÓN.....	2
CAPÍTULO II: MARCO CONCEPTUAL O REFERENCIAL.....	3
CAPÍTULO III: METODOLOGÍA.....	4
CAPÍTULO IV: DESARROLLO DEL TEMA/RESULTADOS.....	5
Generación de lotes.....	7
Generación de Índice columnar.....	8
Preparaciones previas a las pruebas:.....	8
Prueba 1:.....	9
Prueba 2:.....	13
Optimización de Consultas a través de Índices.....	15
CAPÍTULO V: CONCLUSIONES.....	17
CAPÍTULO VI: BIBLIOGRAFÍA.....	18

## **CAPÍTULO I: INTRODUCCIÓN**

El presente trabajo se encargará de abordar el tema de índices columnares en SQL Server y como los mismos están diseñados para mejorar el rendimiento de las consultas en tablas de gran tamaño.

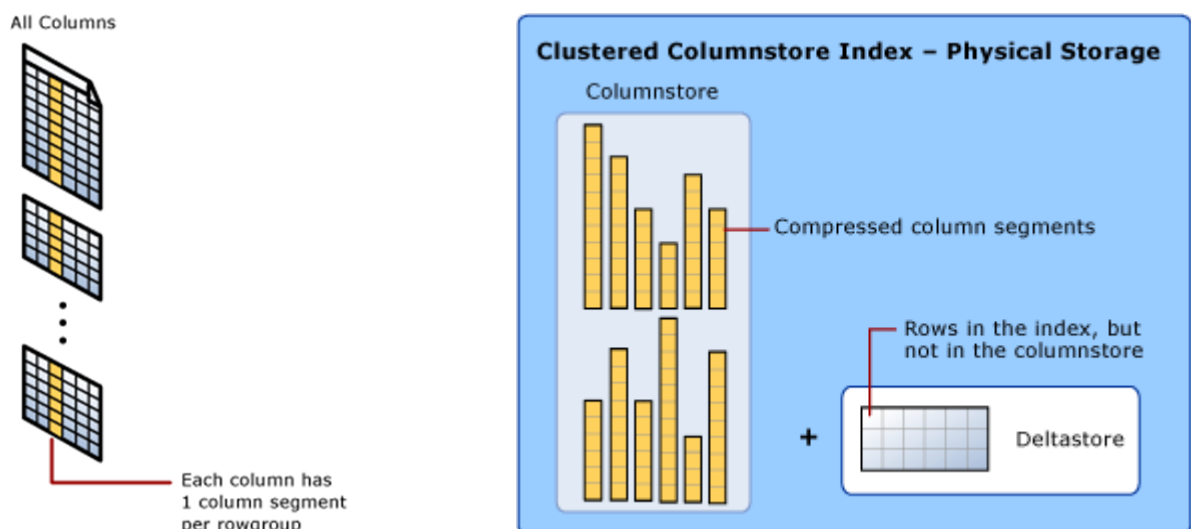
El principal propósito de este trabajo práctico es brindar una comprensión sólida y conceptual de los índices columnares. Además, se desarrollarán pruebas y ejemplos prácticos sobre cómo estos índices usan el almacenamiento de datos basado en columnas y el procesamiento de consultas para lograr mejoras en el rendimiento de las consultas hasta 10 veces más eficientes que en el almacenamiento de datos tradicional orientado a filas.

## CAPÍTULO II: MARCO CONCEPTUAL O REFERENCIAL

Los índices de almacén de columnas son el estándar para almacenar y consultar las tablas de hechos de almacenamiento de datos de gran tamaño. Este índice usa el almacenamiento de datos basado en columnas y el procesamiento de consultas para lograr ganancias de hasta 10 veces el rendimiento de las consultas en el almacenamiento de datos sobre el almacenamiento tradicional orientado a filas. También puede lograr ganancias de hasta 10 veces la compresión de datos sobre el tamaño de los datos sin comprimir.

Teniendo en cuenta que los índices columnares son un tipo de índice que se utilizan para mejorar el rendimiento de consultas en bases de datos con grandes cantidades de datos. Podemos brindar una visión general acerca de los mismos.

- **Almacenamiento Eficiente:** Almacenan cada columna de forma independiente, permitiendo un almacenamiento más eficiente (especialmente en tablas con muchas columnas).
- **Compresión de Datos:** Generalmente aplican técnicas de compresión de datos, reduciendo espacio de almacenamiento, e indirectamente, acelera la lectura de los mismos.
- **Mejora del Rendimiento:** Los índices columnares están diseñados para consultas analíticas y de agregación, como consultas que involucran sumas, promedios o filtrado de datos en una o varias columnas. Estos índices suelen acelerar significativamente estas consultas.
- **No Adecuados para Operaciones de Actualización Frecuente:** Los índices columnares pueden no ser la mejor opción para tablas con muchas operaciones de inserción, actualización o eliminación, ya que pueden ralentizar estas operaciones.



### **CAPÍTULO III: METODOLOGÍA**

Para la consecución del presente trabajo se siguieron una serie de pasos descritos a continuación:

- 1) Se realizó la creación de una nueva tabla, tomando como modelo la tabla gasto tomando como nombre gastoNew.
- 2) Se llevó a cabo una carga masiva de datos (al menos 1 millón de registros) sobre la tabla recién creada. Pudiendo repetir los registros ya existentes.
- 3) Se definió un índice columnar sobre la tabla gastoNew.
- 4) Se ejecutó una consulta sobre las tablas gasto y gastoNew, a fin de evaluar los tiempos de respuestas entre ambas (una con índice columnar y otra sin el mismo).

Para la realización de este trabajo, el sistema de administración de base de datos (DBMS) que utilizamos es Microsoft SQL server versión 2022 (RTM).

Además, cabe mencionar que los datos fueron obtenidos a través de consultas realizadas en internet, en el sitio oficial de microsoft y otras consultas de material bibliográfico que nos sirvió de guía para obtener los conocimientos elementales para llevar a la práctica todo lo aprendido.

Adicionalmente podemos agregar como herramienta, las distintas entrevistas y consultas realizadas a los profesores, quienes nos brindaron las pautas y lograron transmitir una idea conceptual de los requerimientos. También realizamos investigaciones en línea, revisamos libros y documentos especializados.

Distribuimos las responsabilidades de investigación y redacción de manera equitativa. Cada miembro del grupo se enfocó en investigar un aspecto específico de los índices columnares y redactar su parte correspondiente.

Para mantenernos coordinados, utilizamos aplicaciones de mensajería y herramientas de colaboración en línea, como Whatsapp y Google Docs. Esto nos permitió compartir información y avances de manera efectiva, a pesar de nuestras diferencias de horario.

## CAPÍTULO IV: DESARROLLO DEL TEMA/RESULTADOS

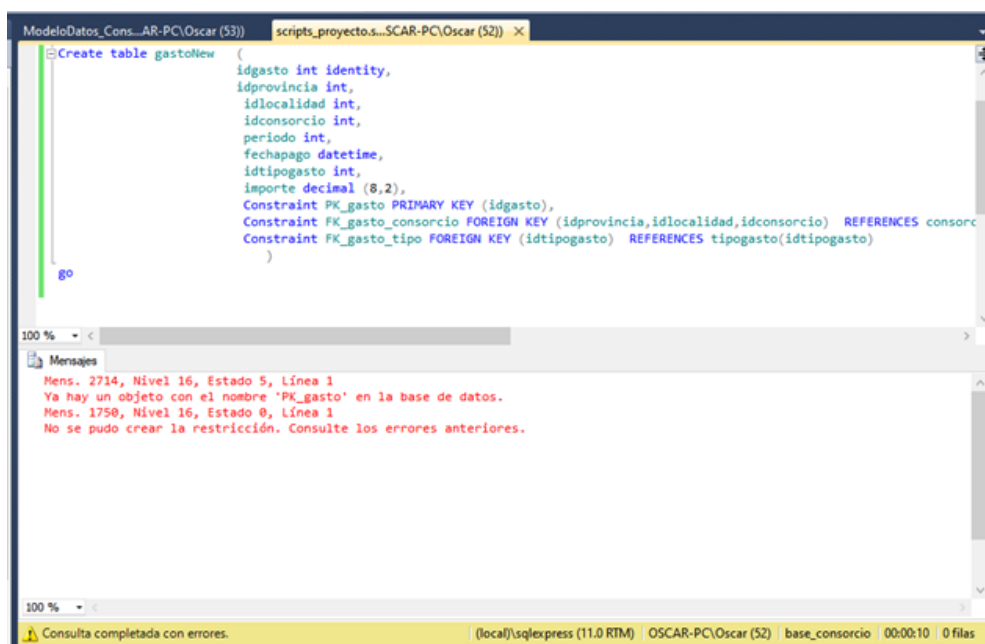
Como primera instancia realizamos la creación de la tabla gastoNew reutilizando la estructura física y lógica brindada en clases prácticas y adaptándola a nuestras necesidades.

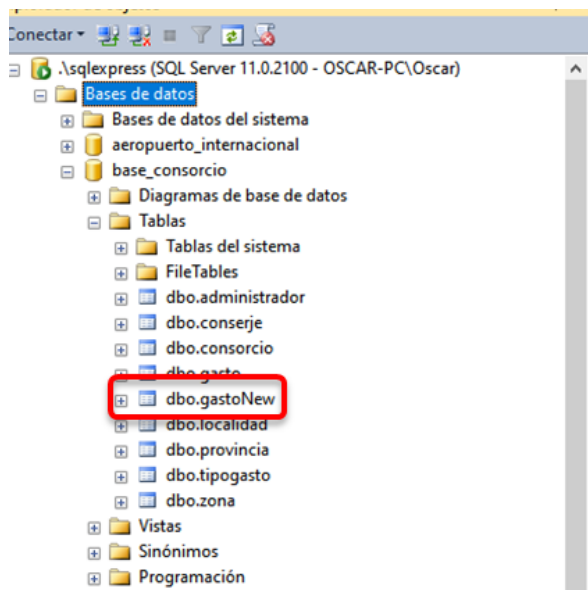
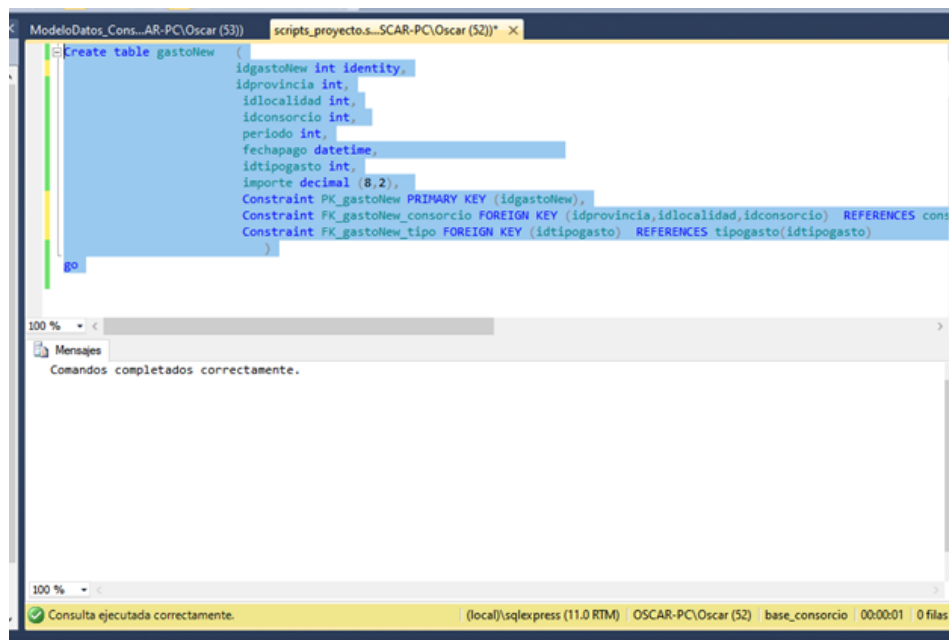
El script utilizado para ello es el que se describe a continuación:

```
use base_consortio;
```

```
Create table gastoNew    (  
    idgastoNew int identity,  
    idprovincia int,  
    idlocalidad int,  
    idconsorcio int,  
    periodo int,  
    fechapago datetime,  
    idtipogasto int,  
    importe decimal (8,2),  
    constraint PK_gastoNew PRIMARY KEY (idgastoNew),  
    constraint FK_gastoNew_consortio FOREIGN KEY  
(idprovincia,idlocalidad,idconsorcio) REFERENCES  
consorcio(idprovincia,idlocalidad,idconsorcio),  
    constraint FK_gastoNew_tipo FOREIGN KEY (idtipogasto) REFERENCES  
tipogasto(idtipogasto)  
)  
go
```

En nuestro caso, en la primera ejecución del primer script tuvimos errores de PK repetidas, a lo cual solucionando este inconveniente pudimos continuar.





## Generación de lotes

Para la generación del lote de pruebas con un millón de registros en la tabla gastoNew utilización un script, el cual, mediante la generación de tablas temporales #LOCALIDADES1 y #TempLocalidades. La tabla #LOCALIDADES1 tendrá los datos de la localidad con la provincia asociada y posee más de 900 registros. Al tratarse de tablas temporales, las inserciones de los registros deben estar en el mismo script.

```
CREATE TABLE #LOCALIDADES1
(
    idrow int identity PRIMARY KEY,
    idprovincia int,
    idlocalidad int,
    descripcion varchar(50),
)
```

Mientras que la tabla #TempLocalidades tendrá un tamaño de un millón de registros en el cual cada uno tendrá un campo localidad1 que se usará para asociar a la tabla #LOCALIDADES1 mediante un JOIN. El valor de localidad1 será un valor aleatorio entre 1 a 966 (localidades existentes).

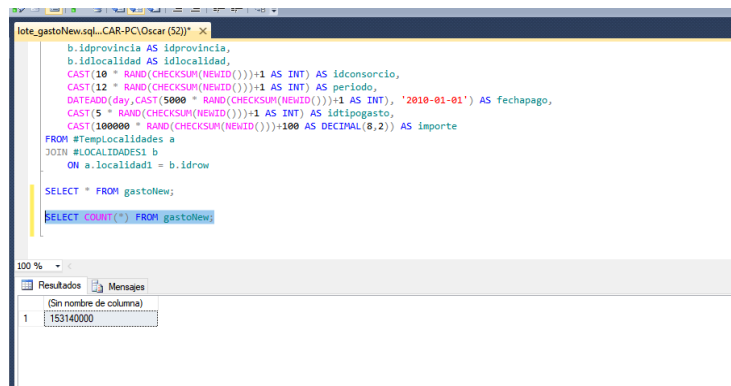
```
SELECT TOP 1000000
    CAST(966 * RAND(CHECKSUM(NEWID()))+1 AS INT) AS localidad1
INTO #TempLocalidades
FROM sys.all_objects a
CROSS JOIN sys.all_objects b;
```

Luego, nos queda insertar los datos en la tabla gastoNew. Para ello, asociamos las tablas temporales mencionadas anteriormente e insertamos sus valores en la tabla. Además, generamos valores aleatorios para las columnas de idconsorcio, periodo, fechapago, idtipogasto e importe.

```
INSERT INTO gastoNew(idprovincia, idlocalidad, idconsorcio, periodo, fechapago, idtipogasto, importe)
SELECT TOP 1000000
    b.idprovincia AS idprovincia,
    b.idlocalidad AS idlocalidad,
    CAST(10 * RAND(CHECKSUM(NEWID()))+1 AS INT) AS idconsorcio,
    CAST(12 * RAND(CHECKSUM(NEWID()))+1 AS INT) AS periodo,
    DATEADD(day, CAST(5000 * RAND(CHECKSUM(NEWID()))+1 AS INT), '2010-01-01') AS fechapago,
    CAST(5 * RAND(CHECKSUM(NEWID()))+1 AS INT) AS idtipogasto,
    CAST(100000 * RAND(CHECKSUM(NEWID()))+100 AS DECIMAL(8,2)) AS importe
FROM #TempLocalidades a
JOIN #LOCALIDADES1 b
    ON a.localidad1 = b.idrow
```

Así ya tendríamos la tabla de gastoNew con un millón de registros generada y lista para generar los índices que necesitamos.





## Generación de Índice columnar

A continuación creamos los índices columnares que utilizaremos en la tabla gastoNew con todas las columnas de la tabla para que todas las consultas se beneficien del índice.

```

CREATE NONCLUSTERED COLUMNSTORE INDEX IDX_gastoNew
ON gastoNew (idprovincia, idlocalidad, idconsorcio, periodo, fechapago, idtipogasto, importe);

```

## Preparaciones previas a las pruebas:

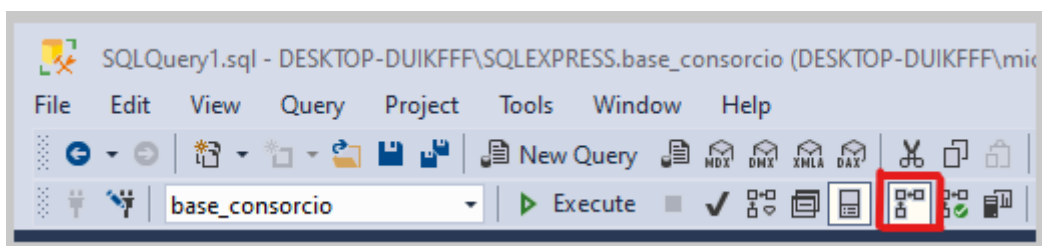
Para la medición de las pruebas utilizamos la siguiente sentencia para ver las operaciones que entrada/salida y el tiempo de ejecución de las consultas:

```

--Para ver las operaciones de entrada/salida y el tiempo de ejecucion de las consultas.
SET STATISTICS IO, TIME ON;

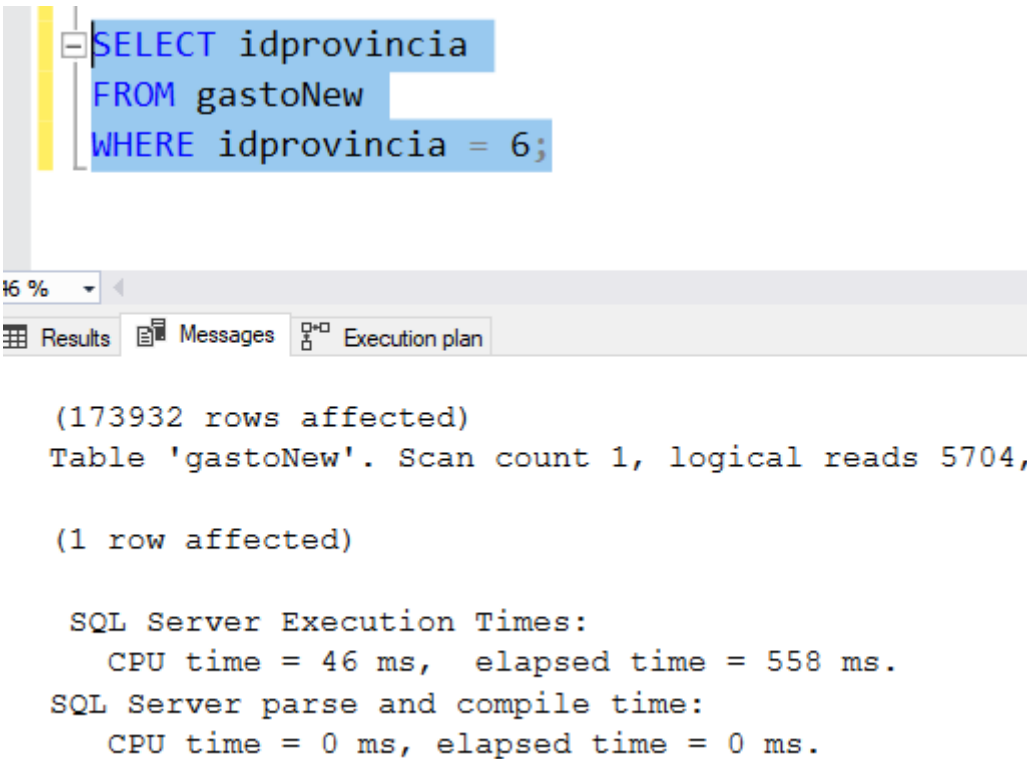
```

Además, activamos el plan de ejecución que nos mostrará qué índice utiliza el motor de base de datos para la consulta. Esta función se encuentra en la barra de tareas en la parte superior de SQL server management:



## Prueba 1:

Realizamos una consulta de idprovincia donde idprovincia es 6 en la tabla gastoNew sin índice columnar:



The screenshot shows a SQL query window with the following text:

```
SELECT idprovincia
FROM gastoNew
WHERE idprovincia = 6;
```

Below the query window, the 'Results' tab is active, displaying the following output:

```
(173932 rows affected)
Table 'gastoNew'. Scan count 1, logical reads 5704,

(1 row affected)

SQL Server Execution Times:
    CPU time = 46 ms,  elapsed time = 558 ms.
SQL Server parse and compile time:
    CPU time = 0 ms,  elapsed time = 0 ms.
```

Como podemos observar, al realizar esta consulta la base de datos encontró 173932 coincidencias tras realizar 5704 lecturas lógicas y con un tiempo de ejecución de 558 ms.

Además, en el plan de ejecución, observamos que la consulta se realiza a través del índice de almacenamiento en filas de la clave primaria que se genera por defecto, debido a que esta no posee un orden adecuado para la consulta la base de datos debe buscar en todas las filas de la tabla, en decir en el millón de registros.

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	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows Read	1000000
Actual Number of Rows for All Executions	173932
Actual Number of Batches	0
Estimated I/O Cost	4,21127
Estimated Operator Cost	5,31143 (100%)
Estimated Subtree Cost	5,31143
Estimated CPU Cost	1,10016
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows for All Executions	174348
Estimated Number of Rows Per Execution	174348
Estimated Number of Rows to be Read	1000000
Estimated Row Size	11 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0
<b>Predicate</b>	
[base_consorcio].[dbo].[gastoNew].[idprovincia]=CONVERT_IMPLICIT(int,[@1],0)	
<b>Object</b>	
[base_consorcio].[dbo].[gastoNew].[PK_gastoNew]	
<b>Output List</b>	
[base_consorcio].[dbo].[gastoNew].idprovincia	

Debido a esto, la misma base de datos nos brinda como recomendación crear un índice para mejorar el rendimiento en un 99,07 % sobre la columna de idprovincia.

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 100%

```
SELECT [idprovincia] FROM [gastoNew] WHERE [idprovincia]=@1
```

Missing Index (Impact 99.0763): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[gastoNew] ([idprovincia])

Clustered Index Scan (Clustered)

[gastoNew].[PK\_gastoNew]

Cost: 100 %

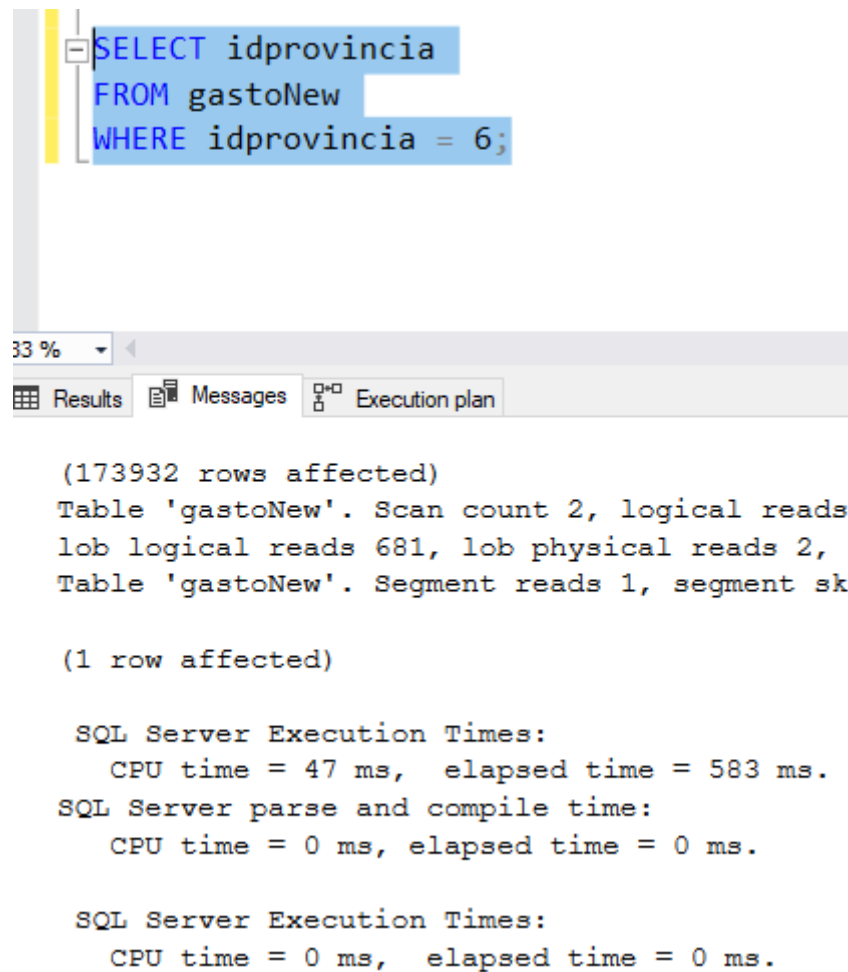
0.219s

173932 of 174348 (99%)

SELECT

Cost: 0 %

Ahora realicemos la misma consulta pero con un índice columnar creado en la tabla gastoNew sobre todas las columnas.



The screenshot shows the SQL Server Enterprise Manager interface. At the top, a query is entered in the query editor:

```
SELECT idprovincia
FROM gastoNew
WHERE idprovincia = 6;
```

Below the query editor, the 'Results' tab is selected, displaying the execution results. The results show that 173932 rows were affected by the first part of the query (the table scan), and 1 row was affected by the filter condition. The execution times are also displayed:

```
(173932 rows affected)
Table 'gastoNew'. Scan count 2, logical reads
lob logical reads 681, lob physical reads 2, :
Table 'gastoNew'. Segment reads 1, segment sk:

(1 row affected)

SQL Server Execution Times:
  CPU time = 47 ms,  elapsed time = 583 ms.
SQL Server parse and compile time:
  CPU time = 0 ms,  elapsed time = 0 ms.

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

Como vemos, en este ejemplo si bien no existe una diferencia significativa en cuanto al tiempo de ejecución, la base de datos ahora realiza solo 681 lecturas lógicas mejorando la eficacia de la consulta.

Además, se observa en el plan de ejecución como ya no debe leer todos los registros de la tabla y un aumento de la eficiencia al bajar los costos de la consulta.

**Columnstore Index Scan (NonClustered)**

Scan a columnstore index, entirely or only a range.

<b>Physical Operation</b>	Columnstore Index Scan
<b>Logical Operation</b>	Index Scan
<b>Actual Execution Mode</b>	Batch
<b>Estimated Execution Mode</b>	Batch
<b>Storage</b>	ColumnStore
<b>Actual Number of Rows for All Executions</b>	173932
<b>Actual Number of Batches</b>	194
<b>Estimated Operator Cost</b>	0,113141 (100%)
<b>Estimated I/O Cost</b>	0,003125
<b>Estimated CPU Cost</b>	0,110016
<b>Estimated Subtree Cost</b>	0,113141
<b>Estimated Number of Executions</b>	1
<b>Number of Executions</b>	1
<b>Estimated Number of Rows for All Executions</b>	174348
<b>Estimated Number of Rows Per Execution</b>	174348
<b>Estimated Number of Rows to be Read</b>	1000000
<b>Estimated Row Size</b>	23 B
<b>Actual Rebinds</b>	0
<b>Actual Rewinds</b>	0
<b>Ordered</b>	False
<b>Node ID</b>	2

**Predicate**

[base\_consorcio].[dbo].[gastoNew].[idprovincia]=CONVERT\_IMPLICIT(int,  
[@1],0)

**Object**

[base\_consorcio].[dbo].[gastoNew].[IDX\_gastoNew]

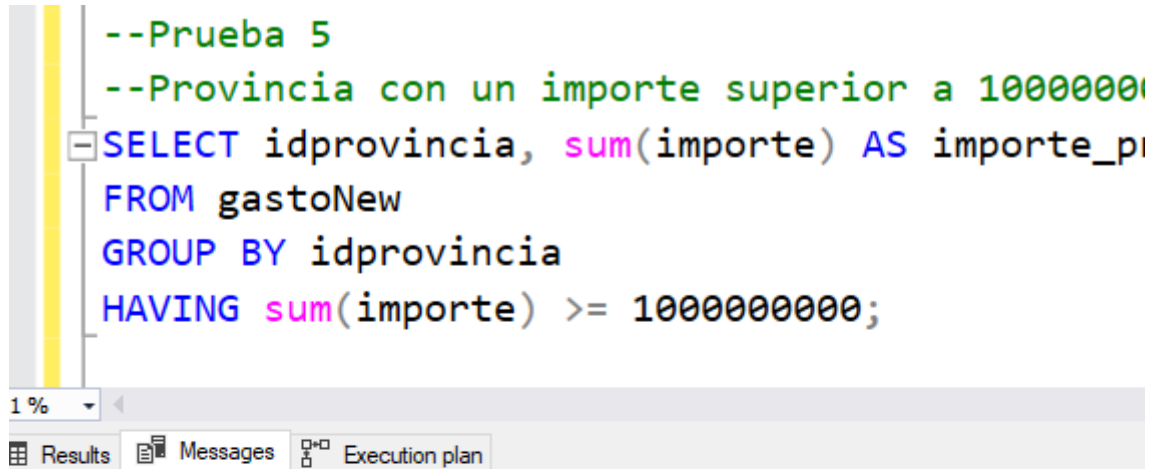
**Output List**

[base\_consorcio].[dbo].[gastoNew].idgastoNew; [base\_consorcio].[dbo].  
[gastoNew].idprovincia; Generation1003

## Prueba 2:

Probemos ahora con un test más complejo. En este caso vamos a buscar aquellas provincias que tengan un importe total mayor a mil millones.

Primero ejecutamos la sentencia sin índice columnar:



```
--Prueba 5
--Provincia con un importe superior a 1000000000
SELECT idprovincia, sum(importe) AS importe_pi
FROM gastoNew
GROUP BY idprovincia
HAVING sum(importe) >= 1000000000;
```

1 %

Results Messages Execution plan

```
(17 rows affected)
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0,
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0,
Table 'gastoNew'. Scan count 1, logical reads 5704, physical reads 0,
```

```
(1 row affected)
```

```
SQL Server Execution Times:
    CPU time = 265 ms,  elapsed time = 295 ms.
SQL Server parse and compile time:
    CPU time = 0 ms, elapsed time = 0 ms.
```

Esta realiza la búsqueda y encuentra 17 coincidencias en 295 ms. y 5704 lecturas lógicas.

Ahora, comprobamos los resultados con un índice columnar:

```
--Prueba 5
--Provincia con un importe superior a 1000000000
SELECT idprovincia, sum(importe) AS importe_provincia
FROM gastoNew
GROUP BY idprovincia
HAVING sum(importe) >= 1000000000;
```

161 %

Results Messages Execution plan

```
(17 rows affected)
Table 'gastoNew'. Scan count 2, logical reads 0, physical reads 0, lob logical reads 1686, lob physical reads 2, lob page server reads 0.
Table 'gastoNew'. Segment reads 1, segment skipped 0.
```

```
(1 row affected)
```

```
SQL Server Execution Times:
    CPU time = 0 ms,  elapsed time = 31 ms.
SQL Server parse and compile time:
    CPU time = 0 ms,  elapsed time = 0 ms.
```

En este caso obtenemos las 17 coincidencias en solo 31 ms. y 1686 lecturas lógicas. Por lo que se obtiene una mejora significativa en el tiempo de ejecución en comparación a la misma sentencia sin un índice columnar para la tabla.

## Optimización de Consultas a través de Índices

Al igual que en nuestro tema de investigación, este hace uso de índices pero con la diferencia de que estos no son índices columnares.

Para iniciar la integración y realizar las consultas de prueba sobre la tabla gastoNew primero debemos eliminar el índice columnar IDX\_gastoNew existente.

```
--Elimina el índice columnar creado en la tabla gastoNew  
DROP INDEX IDX_gastoNew ON gastoNew;
```

Ahora realizamos una consulta para ver cuanto tiempo tarda y cuantos procesos de IO realiza.

```
--Consulta: Gastos del periodo 8  
SELECT g.idgastoNew, g.periodo, g.fechapago, t.descripcion  
FROM gastoNew g  
INNER JOIN tipogasto t ON g.idtipogasto = t.idtipogasto  
WHERE g.periodo = 8 ;
```

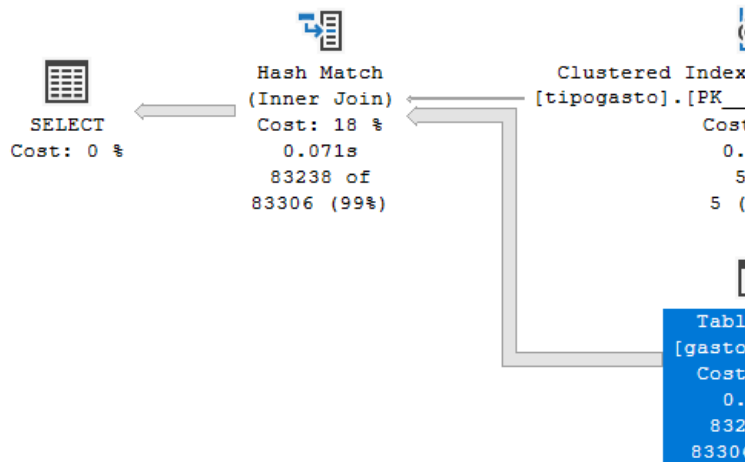
Resultados primera consulta sin índices:

```
(83238 rows affected)  
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, physical reads in page file 0  
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, physical reads in page file 0  
Table 'gastoNew'. Scan count 1, logical reads 6803, physical reads 0, physical reads in page file 0  
Table 'tipogasto'. Scan count 1, logical reads 2, physical reads 0, physical reads in page file 0  
  
(1 row affected)  
  
SQL Server Execution Times:  
    CPU time = 62 ms,  elapsed time = 433 ms.  
SQL Server parse and compile time:  
    CPU time = 0 ms,  elapsed time = 0 ms.
```



En el plan de ejecución, el motor nos recomienda crear un índice no agrupado en la tabla periodo de gastoNew que incluya las columnas 'idgastoNew', 'fechapago' y 'idtipogasto' para optimizar la consulta en un 85,97%:

```
Query 1: Query cost (relative to the batch): 100%
SELECT g.idgastoNew, g.periodo, g.fechapago, t.descripcion FROM gasto
Missing Index (Impact 85.9746): CREATE NONCLUSTERED INDEX [<Name of I
```



Creamos el índice recomendado por el motor (es importante el orden de las columnas, ya que, se ordenarán en esa prioridad):

```
--Creamos un nuevo índice NONCLUSTERED en periodo, idtipogasto, idgastoNew y fechapago en ese orden.
CREATE NONCLUSTERED INDEX IDX_gastoNew_periodo_idtipogasto_idgastoNew_fechapago
ON gastoNew (periodo, idtipogasto, idgastoNew, fechapago);
```

y observamos los resultados:

```
(83238 rows affected)
Table 'gastoNew'. Scan count 1, logical reads 355,
Table 'tipogasto'. Scan count 1, logical reads 2, 1
```

```
(1 row affected)
```

```
SQL Server Execution Times:
    CPU time = 16 ms,  elapsed time = 377 ms.
SQL Server parse and compile time:
    CPU time = 0 ms,  elapsed time = 0 ms.
```

Como conclusión, el uso del índice disminuyó las lecturas lógicas y el tiempo de ejecución.

## **CAPÍTULO V: CONCLUSIONES**

Pudimos observar en el desarrollo de este trabajo que el uso de índices columnares en las tablas mejoran significativamente el rendimiento de las consultas en comparación con las tablas sin índices columnares, especialmente en conjuntos de datos masivos.

Es importante destacar que los índices columnares son efectivos en consultas analíticas y de agregación en lugar de consultas de búsqueda o filtrado de registros individuales. Por lo tanto, la elección de usar índices columnares en las tablas depende de las necesidades específicas y el tipo de consultas que se realicen en las mismas.

Otra cuestión importante a tener en cuenta es que la creación de un índice columnar puede requerir más recursos de almacenamiento, por lo que debemos considerar los beneficios de rendimiento con costos de almacenamiento.

En la integración de trabajos nos permitió ver que nuestro proyecto no solo se centre en la creación de índices columnares, sino que también ilustre cómo estos índices se relacionan con otras áreas cruciales de la administración de bases de datos en SQL Server.

En la optimización de Consultas a través de Índices nos permitió profundizar en cómo los índices columnares pueden optimizar consultas específicas y demostrar casos de estudio donde la creación de índices específicos ha mejorado significativamente el rendimiento de consultas clave.

En el tema de Backup y Restore nos mostró cómo realizar respaldos antes y después de crear o modificar índices columnares importantes. Explicar cómo los respaldos son esenciales al realizar cambios significativos en la estructura de la base de datos, como agregar o eliminar índices.

Y sobre todo resaltar cómo la implementación adecuada de índices columnares no solo mejora el rendimiento de las consultas, sino que también impacta la seguridad, la integridad de los datos y la administración general de la base de datos.

## CAPÍTULO VI: BIBLIOGRAFÍA

- <https://learn.microsoft.com/es-es/sql/relational-databases/indexes/columnstore-indexes-overview?view=sql-server-ver16>
- <https://learn.microsoft.com/es-es/sql/relational-databases/indexes/columnstore-indexes-design-guidance?view=sql-server-ver16#choose-the-best-columnstore-index-for-our-needs>
- <https://learn.microsoft.com/es-es/sql/relational-databases/indexes/get-started-with-columnstore-for-real-time-operational-analytics?view=sql-server-ver16>
- <https://datoptim.com/millones-de-datos-aleatorios-en-sql-server/>