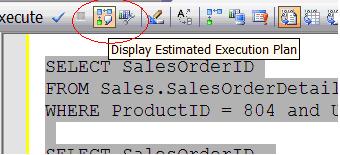
**SQL Server Indexes: The Basics**

Indexes directly affect the performance of database applications. This article uses analogies to describe how indexes work. The estimated execution plan feature of the Query Window is utilized to compare the performance of two queries in a batch.

Most developers and DBAs understand that indexes on database tables enable applications and reports to run more efficiently. But, do they really know whether the database system will take advantage of an index to process a particular query or update statement? Does the order of columns specified in the index matter? What is the difference between a clustered index and a non-clustered index? To understand some index concepts, I have found it useful to use some analogies.

To follow along with this article, you will need an instance of SQL Server 2005 with the AdventureWorks database installed. Each example will compare the performance of two queries in the same batch using the Graphical Estimated Execution Plan. We will focus on the "Query Cost (relative to the batch)" to compare the performance of the two queries. To see the Graphical Estimated Execution Plan, click the button shown below (Figure 1) instead of running the query.

   
**Figure 1: The button for the Estimated Execution Plan**

**Clustered Indexes**

A printed phone directory is a great example of a clustered index. Each entry in the directory represents one row of the table. A table can have only one clustered index. That is because a clustered index is the actual table sorted in order of the cluster key. At first glance, you might think that inserting a new row into the table will require all the rows after the inserted row to be moved on the disk. Luckily, this is not the case. The row will have to be inserted into the correct data page, and this might require a page split if there is not enough room on the page for the new row. A list of pointers maintains the order between the pages, so the rows in other pages will not have to actually move.

The primary key of the phone directory is the phone number. Usually the primary key is used as the clustering key as well, but this is not the case in our example. The cluster key in the phone directory is a combination of last name and first name. How would you find a friend's phone number if you knew the last and first name? Easy, open the book approximately to the section of the book that contains the entry. If your friend's last name starts with an "F", you will search near the beginning of the book, if an "S", you will search towards the back. You can use the names printed at the top of the page to quickly locate the page with the listing. You then drill down to the section of the correct page till you find the last name you are looking for. Now you can use the first name to choose the correct listing. The phone number is right there next to the name. It probably takes more time describe the process than to actually do it. Using the last name plus first name to find the number is called a clustered index seek.

Lets say you wanted to find all the people who have the last name of "Smith" for a family reunion. How quickly could you locate all the names? Of course, it would take a matter of seconds to find all of them grouped together, possibly over a few pages. What if you wanted to locate everyone with the first name of "Jeff" in the book? Could it be done? Of course it could, but you would have to look at every entry in the entire book because the first name is the second column in our cluster key. This is called a clustered index scan, a very expensive operation on a big table.

Here is an example using one of the tables in AdventureWorks. The Sales.SalesOrderDetail table has a clustered index on SalesOrderID plus SalesOrderDetailID. Take a look at the graphical estimated execution plan (Figure 2) of the batch paying particular attention to the Query cost when either the first or second column in the cluster key is used.

SELECT SalesOrderID, SalesOrderDetailID

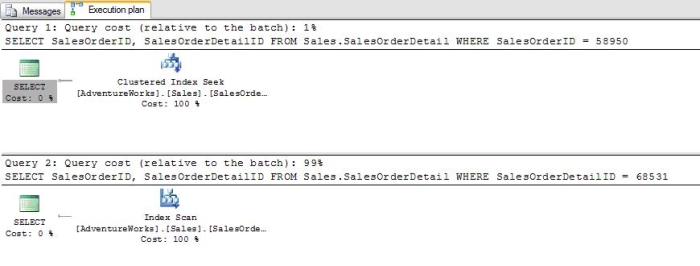
FROM Sales.SalesOrderDetail

WHERE SalesOrderID = 58950

SELECT SalesOrderID, SalesOrderDetailID

FROM Sales.SalesOrderDetail

WHERE SalesOrderDetailID = 68531

  
**Figure 2: Compare the query costs when either the first or second column is searched.**

The first query in the batch will utilize a "Clustered Index Seek" while the second will use a "Clustered Index Scan". Notice that the first query will use 1% of the cost of the batch while the second query will use 99% of the cost. These results make sense when you think about the fact that the first column of the index is SalesOrderID. When trying to find a particular row based on the SalesOrderDetailID in the second query, the entire table is searched.

Another search you probably wouldn't attempt with the phone directory is looking for every entry with a particular last name or a particular first name. If you wanted to find all of the entries with the first name of "Jeff" or the last name of "Smith" you would have to search every name in the book. Here is a batch illustrating this point:

SELECT SalesOrderID, SalesOrderDetailID

FROM Sales.SalesOrderDetail

WHERE SalesOrderID = 43683

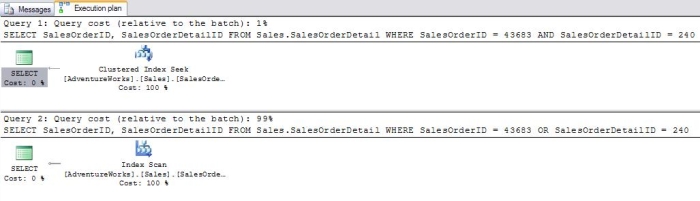
**AND**SalesOrderDetailID = 240

SELECT SalesOrderID, SalesOrderDetailID

FROM Sales.SalesOrderDetail

WHERE SalesOrderID = 43683

**OR**SalesOrderDetailID = 240

  
**Figure 3: Using "OR" causes a scan.**

The second query using "OR" in the WHERE clause uses 99% of the resources of the batch because the entire clustered index must be scanned.

**Non-Clustered Indexes**

The index in the back of a book is an example of a non-clustered index. A non-clustered index has the indexed columns and a pointer or bookmark pointing to the actual row. In the case of our example it contains a page number. Another example could be a search done on Google or another of the search engines. The results on the page contain links to the original web pages. The thing to remember about non-clustered indexes is that you may have to retrieve part of the required information from the rows in the table. When using a book index, you will probably have to turn to the page of the book. When searching on Google, you will probably have to click the link to view the original page. If all of the information you need is included in the index, you have no need to visit the actual data.

In SQL Server 2000 the term "Bookmark Lookup" was used to describe the process of retrieving some of the columns from the actual row. In my experience, this is a very expensive operation when performed on large tables. Now "Clustered Index Seek" for tables with a clustered index, and "RID Lookup" for tables without a clustered index are the terms used. I find this very confusing since "Clustered Index Seek" is usually the preferred behavior. In Books Online, it states that when the keyword "LOOKUP" appears, it is actually a bookmark lookup. I haven't found the term displayed in the Graphical Execution plan. To see the keyword, you have to use the SHOWPLAN\_TEXT option.

Here is a batch and the graphical execution plan (Figure 4) when using the non-clustered index to search on the ProductID column. The second query retrieves a column that is not included in the index key:

SELECT ProductID, SalesOrderID, SalesOrderDetailID

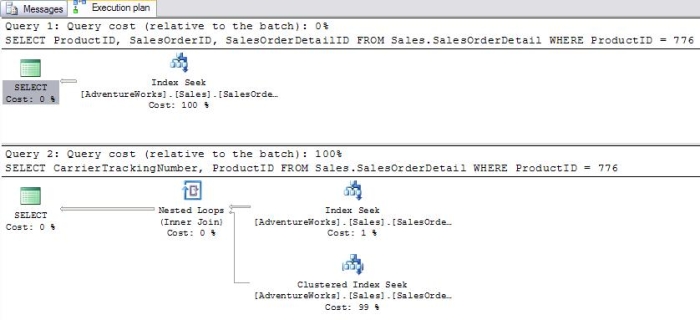
FROM Sales.SalesOrderDetail

WHERE ProductID = 776

SELECT CarrierTrackingNumber, ProductID

FROM Sales.SalesOrderDetail

WHERE ProductID = 776

   
**Figure 4: Retrieving a column not part of the index.**

The first query cost is 0% of the batch. In this case the ProductID is retrieved from the index -- no need to look at the actual table. The SalesOrderID and SalesOrderDetailID are not defined in the index, but are automatically included since they comprise the primary key. The second query in the batch costs 100% of the resources. Even though we are searching on an indexed column, we must retrieve the CarrierTrackingNumber from the actual table. Surprisingly, a "Clustered Index Seek" is the most expensive part of the query. To see why, run the following:

SET SHOWPLAN\_TEXT ON

GO

SELECT CarrierTrackingNumber, ProductID

FROM Sales.SalesOrderDetail

WHERE ProductID = 776

Instead of executing the query, the execution plan is returned in text format (results abbreviated). The "LOOKUP" keyword designates that a bookmark lookup was used.

|--Clustered Index Seek

...

**LOOKUP** ORDERED FORWARD)

In the previous example, if the CarrierTrackingNumber was part of the index, the performance issue would be solved. Run this script to modify the index:

USE [AdventureWorks]

GO

DROP INDEX [IX\_SalesOrderDetail\_ProductID] ON [Sales].[SalesOrderDetail] WITH ( ONLINE = OFF )

GO

USE [AdventureWorks]

GO

CREATE NONCLUSTERED INDEX [IX\_SalesOrderDetail\_ProductID] ON [Sales].[SalesOrderDetail]

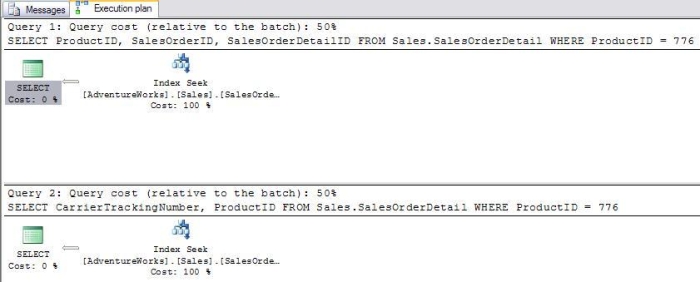
(

[ProductID] ASC,

[CarrierTrackingNumber] ASC

)

Now when the query batch is run, the second query has the same cost as the first query because the CarrierTrackingNumber is now part of the index. The table doesn't have to be accessed. Below is the graphical execution plan (Figure 5):

  
**Figure 5: Using a covering index**

When all of the required columns are part of the index, it is called a "covering index". An index key can contain up to 16 columns and can be up to 900 bytes wide. SQL Server 2005 has a new feature to create indexes that surpass these limits called "included columns". This allows you to include additional columns in the index over the 16 column limit or columns that would be too large to include.

While you can only have one clustered index per table, you can have up to 249 non-clustered indexes per table. If you ever have that many, rest assured, you have a design problem! An important thing to keep in mind is that while indexes can improve the performance of queries, indexes take up disk space and require resources to keep updated. If a table has four non-clustered indexes, every write to that table requires four additional writes to keep the indexes up to date.

One more interesting thing about non-clustered indexes is that SQL Server can use them in combination or along with the clustered index. The SalesOrderDetail table has a non-clustered index on the ModifiedDate column. This query batch shows the difference when a WHERE clause has two conditions. In the first query, there is no index on the second column used in the WHERE clause. In the second query, there is a separate index on each index in the WHERE clause. In this case, the SQL Server uses the two indexes in combination to process the query, and the performance of the second query is much better (Figure 6).

SELECT SalesOrderID

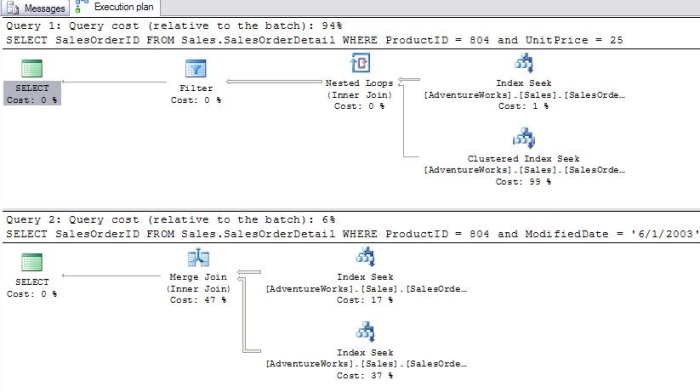
FROM Sales.SalesOrderDetail

WHERE ProductID = 804 and UnitPrice = 25

SELECT SalesOrderID

FROM Sales.SalesOrderDetail

WHERE ProductID = 804 and ModifiedDate = '6/1/2003'

  
**Figure 6: Using two indexes in combination.**

Recall how using the "OR" operator with a clustered index caused a clustered index scan. If there are individual indexes defined on two columns used in a WHERE clause with "OR", the performance is about the same as when "AND" is used. Here is an example comparing "AND" and "OR", and the costs are almost evenly divided (Figure 7):

SELECT SalesOrderID

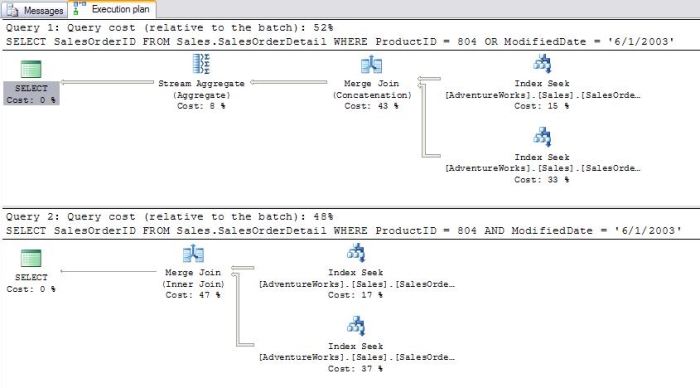
FROM Sales.SalesOrderDetail

WHERE ProductID = 804 OR ModifiedDate = '6/1/2003'

SELECT SalesOrderID

FROM Sales.SalesOrderDetail

WHERE ProductID = 804 AND ModifiedDate = '6/1/2003'

   
**Figure 7: Using "OR" with two indexes.**

There are two more issues to consider concerning indexes and WHERE clauses. What happens when a function is used in the WHERE clause? In that case, the index will not be used because the function will have to be applied to every row in the table. What happens when a wildcard is used in the WHERE clause? In this case it depends on where the wildcard is located. If the first character of the search term is replaced with a wildcard, the index will not be used and a table scan will result. Think about our phone directory example for a minute. How easy would it be to find an entry if you did not know the first letter of the last name?

# Getting Started with the New Column Store Index of SQL Server 2012

## Introduction

Column Store Index is a new feature in SQL Server 2012 that improves performance of data warehouse queries several folds. Unlike regular indexes or heaps, which store data in B-Tree structure (in row-wise fashion), the column store index stores data in columnar fashion and uses compression aggressively to reduce the disk I/O needed to serve the query request along with the newly introduced batch mode processing.

A column store index can be created as a non-clustered index only and only one column store index can be created on a table, though we can include all the columns of the table (except those which are not supported as part of column store index

CREATENONCLUSTEREDCOLUMNSTOREINDEX<IndexName>

ON<TableName>(<Column1>,<Column2>,<Column3>,...<ColumnN)

ON<FilegroupName>

Let me show an example of creating column store index. For that, I want to first create two empty tables, which will mimic the schema/structure of the Sales.SalesOrderDetail table of the AdventureWorks database.

CREATETABLESalesOrderDetailWithRegularIndex(

       [SalesOrderID][int]NOTNULL,

       [SalesOrderDetailID][int]IDENTITY(1,1)NOTNULL,

       [CarrierTrackingNumber][nvarchar](25)NULL,

       [OrderQty][smallint]NOTNULL,

       [ProductID][int]NOTNULL,

       [SpecialOfferID][int]NOTNULL,

       [UnitPrice][money]NOTNULL,

       [UnitPriceDiscount][money]NOTNULL,

       [LineTotal][money],

       [rowguid][uniqueidentifier]ROWGUIDCOL  NOTNULL,

       [ModifiedDate][datetime]NOTNULL,

 CONSTRAINT[PK\_SalesOrderDetailWithRegularIndex\_SalesOrderID\_SalesOrderDetailID]

PRIMARYKEYCLUSTERED

(

       [SalesOrderID]ASC,

       [SalesOrderDetailID]ASC

))ON[PRIMARY]

GO

CREATETABLESalesOrderDetailWithColumnStoreIndex(

       [SalesOrderID][int]NOTNULL,

       [SalesOrderDetailID][int]IDENTITY(1,1)NOTNULL,

       [CarrierTrackingNumber][nvarchar](25)NULL,

       [OrderQty][smallint]NOTNULL,

       [ProductID][int]NOTNULL,

       [SpecialOfferID][int]NOTNULL,

       [UnitPrice][money]NOTNULL,

       [UnitPriceDiscount][money]NOTNULL,

       [LineTotal][money],

       [rowguid][uniqueidentifier]ROWGUIDCOL  NOTNULL,

       [ModifiedDate][datetime]NOTNULL,

 CONSTRAINT[PK\_SalesOrderDetailWithColumnStoreIndex\_SalesOrderID\_SalesOrderDetailID]

PRIMARYKEYCLUSTERED

(

       [SalesOrderID]ASC,

       [SalesOrderDetailID]ASC

))ON[PRIMARY]

GO

Next I want to load a good amount of data (20 times data of AdventureWorks.Sales.SalesOrderDetail, which is close to 2.4 millions) into these tables so that performance differences will be evident:

INSERTINTOSalesOrderDetailWithRegularIndex(SalesOrderID,CarrierTrackingNumber,

OrderQty,ProductID,SpecialOfferID,UnitPrice,UnitPriceDiscount,LineTotal,rowguid,

ModifiedDate)

SELECTSalesOrderID,CarrierTrackingNumber,OrderQty,ProductID,

SpecialOfferID,UnitPrice,UnitPriceDiscount,LineTotal,rowguid,ModifiedDate

FROM[AdventureWorks2012].Sales.[SalesOrderDetail]

GO 20

INSERTINTOSalesOrderDetailWithColumnStoreIndex(SalesOrderID,CarrierTrackingNumber,

OrderQty,ProductID,SpecialOfferID,UnitPrice,UnitPriceDiscount,LineTotal,rowguid,

ModifiedDate)

SELECTSalesOrderID,CarrierTrackingNumber,OrderQty,ProductID,SpecialOfferID,

UnitPrice,UnitPriceDiscount,LineTotal,rowguid,ModifiedDate

FROM[AdventureWorks2012].Sales.[SalesOrderDetail]

GO 20

SELECTCount(\*)FROM[AdventureWorks2012].Sales.[SalesOrderDetail]

SELECTCount(\*)FROMSalesOrderDetailWithRegularIndex

SELECTCount(\*)FROMSalesOrderDetailWithColumnStoreIndex

And finally I want to create a regular non-cluster index (on ProductId and LineTotal columns) on the first table, and column store index on the second table, which will include ProductId and LineTotal columns.

CREATENONCLUSTEREDINDEX[XI\_SalesOrderDetailWithRegularIndex\_ProductID\_LineTotal]

ONSalesOrderDetailWithRegularIndex

(ProductID,LineTotal)

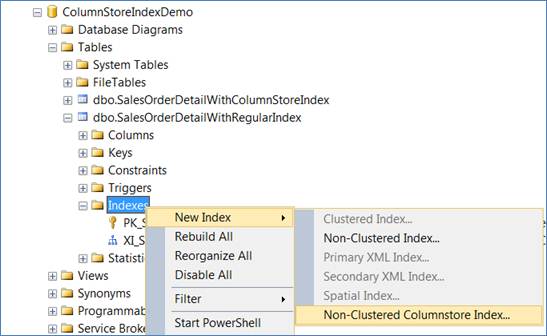
CREATENONCLUSTEREDCOLUMNSTOREINDEX

[XI\_SalesOrderDetailWithColumnStoreIndex\_ProductID\_LineTotal]

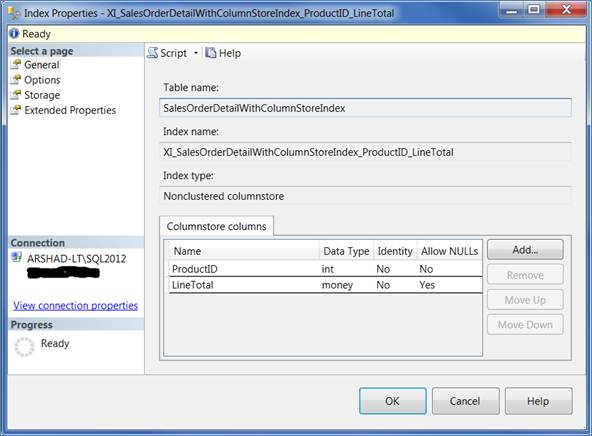
ONSalesOrderDetailWithColumnStoreIndex

(ProductID,LineTotal)

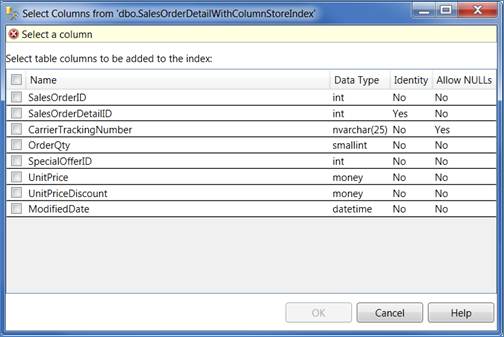
Not only you can create column store index using T-SQL script, but you can also take advantage of the wizard available in SSMS (SQL Server Management Studio) to create column store index; Right click on Indexes node under the desired table and then click on New Index -> Non-Clustered Columnstore Index as shown below:

  
Click on New Index -> Non-Clustered Columnstore Index

You can also modify the existing column store index using the wizard; just double click on the index name and you will see the Index Properties dialog box, as shown below:

  
Index Properties dialog box

Using the wizard is quite helpful when you have a large number of columns in the table. You can exclude or include columns using the wizard, and when you click on OK the column store index will be re-created.

  
Select a column

## Analyzing the Performance Benefits of Using Column Store Index

Column store index has been designed to substantially accelerate common data warehouse queries, which require scanning, aggregation and filtering of large amounts of data or joining multiple tables like a star schema. With column store index, you can get almost interactive response time for queries against billions of rows on an economical SMP server with enough RAM to hold your frequently accessed data. So let’s see running these kinds of queries against both tables (one that doesn’t have column store index and one that has column store index) that we created earlier.

SELECTProductID,SUM(LineTotal)AS'ProductWiseSale'FROM

SalesOrderDetailWithRegularIndex

GROUPBYProductID

ORDERBYProductID

SELECTProductID,SUM(LineTotal)AS'ProductWiseSale'FROM

SalesOrderDetailWithColumnStoreIndex

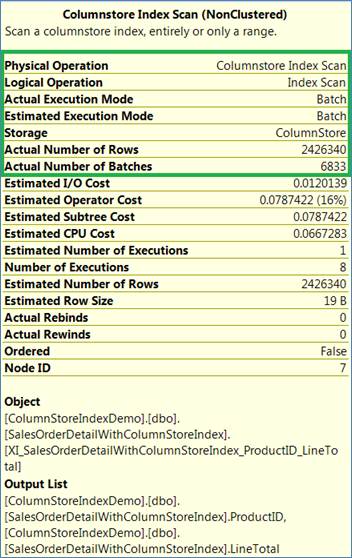
GROUPBYProductID

ORDERBYProductID

And wow! The relative cost of the second query (which uses column store index) is just 7% as opposed to the relative cost of first query (which uses regular index) which is 93%.

  
The relative cost of the second query

For column store index exclusively, SQL Server 2012 introduces a new execution mode called Batch Mode, which processes batches of rows (as opposed to the row by row processing in case of regular index) that is optimized for multicore CPUs and increased memory throughput of modern hardware architecture. It also introduced a new operator for column store index processing as shown below:

  
Columnstore Index Scan

When I ran the query with STATISTICS IO ON, I found stunning results (with significant performance) of using column store index vs regular index, as you can see below:

--Clear the procedure cache and buffer cache

--CAUTION - Please don't run these clean script on live/production system

DBCCDROPCLEANBUFFERS

DBCCFREEPROCCACHE

SETSTATISTICSIOON

SELECTProductID,SUM(LineTotal)AS'ProductWiseSale'FROMSalesOrderDetailWithRegularIndex

GROUPBYProductID

ORDERBYProductID

SELECTProductID,SUM(LineTotal)AS'ProductWiseSale'FROMSalesOrderDetailWithColumnStoreIndex

GROUPBYProductID

ORDERBYProductID

SETSTATISTICSIOOFF

(266 row(s) affected)

Table 'SalesOrderDetailWithRegularIndex'. Scan count 9, logical reads 7945, physical reads 1, read-ahead reads 7879, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

(266 row(s) affected)

Table 'SalesOrderDetailWithColumnStoreIndex'. Scan count 8, logical reads 141, physical reads 2, read-ahead reads 179, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Even the time required to run these two queries greatly varied, the queries with regular index took 625 ms for CPU cycle and 2238 ms as elapsed time vs just 46 ms for CPU cycle and 57 ms as elapsed time for the second query, which uses column store index:

--Clear the procedure cache and buffer cache

--CAUTION - Please don't run these clean script on live/production system

DBCCDROPCLEANBUFFERS

DBCCFREEPROCCACHE

SETSTATISTICSTIMEON

SELECTProductID,SUM(LineTotal)AS'ProductWiseSale'FROMSalesOrderDetailWithRegularIndex

GROUPBYProductID

ORDERBYProductID

SELECTProductID,SUM(LineTotal)AS'ProductWiseSale'FROMSalesOrderDetailWithColumnStoreIndex

GROUPBYProductID

ORDERBYProductID

SETSTATISTICSTIMEOFF

(266 row(s) affected)

 SQL Server Execution Times:

   CPU time = 625 ms,  elapsed time = 2238 ms.

(266 row(s) affected)

 SQL Server Execution Times:

   CPU time = 46 ms,  elapsed time = 57 ms.

Sometimes when you have column store index on a table, but for some reason you don’t want that index to be used as part of your query execution, you can use the “OPTION (IGNORE\_NONCLUSTERED\_COLUMNSTORE\_INDEX)” clause to exclude column store index as part of execution plan:

SELECTProductID,SUM(LineTotal)AS'ProductWiseSale'FROMSalesOrderDetailWithColumnStoreIndex

GROUPBYProductID

ORDERBYProductID

OPTION (IGNORE\_NONCLUSTERED\_COLUMNSTORE\_INDEX)

On a different note, as per a performance improvement study done by Microsoft on a 32-logical processor machine with 256GB of RAM on a table with 1 TB of data and 1.44 billion rows; the queries gained a 16X speed-up in CPU time and a whopping 455X improvement in elapsed time. In real terms it means a query that took 501 seconds originally (when using no column store index) was reduced to merely 1.1 seconds (when using column store index).