**Chapter 3 – Sales Use Case – Analytical Functions**

Now that we have created and familiarized ourselves with the Sales Data Warehouse and created queries that use aggregate window functions we can move to our next step, learning and creating queries using the analytical functions available with SQL Server.

In the last chapter we touched upon performance tuning the queries. We will use the same approach in this chapter, but we will look at other performance tools such as client statistics to give us a more complete picture of how the queries run.

We will make heavy use of execution statistics, that is we compare statistics before recommended indexes are created and after, to see if they improve performance or not.

Lastly, we will look at setting up pre-loaded reporting tables and memory optimized tables to see if they can yield higher query performance.

The functions we are studying are processing intensive, so we want to make sure we write efficient, accurate and fast executing queries and scripts. Query tuning, index creation and performance analysis are all part of the process for delivering useful and actionable data to your business analysts.

**Analytical Functions**

Below are the eight functions for this category sorted by name. They can be used in queries together with the OVER() clause to generate valuable reports for our business analysts.

• CUME\_DIST()

• FIRST\_VALUE()

• LAST\_VALUE()

• LAG()

• LEAD()

• PERCENT\_RANK()

• PERCENTILE\_CONT()

• PERCENTILE\_DISC()

The same approach as the prior chapter will be taken. I describe what the function does, present the code and the results. Next, some performance analysis is performed so we can see what improvements, if any, are required.

As we saw in the examples in chapter 2, improvements can be adding indexes or even creating a denormalized report table so that when the query using the analytical window functions is executed, it goes against preloaded data avoiding joins, calculations, and complex logic.

**CUME\_DIST() Function**

This function calculates the relative position of a value within a data set like a table, partition or a table variable loaded with test data.

How does it work?

First, calculate how many values come before it or are equal to it (call this value C1). Next calculate the number of values or rows in the data set (call this C2). C1 is then divided by C2 to deliver the cumulative distribution result. The values returned are a float data type so you need to use the FORMAT() function to convert it to a percentage..

By the way, this function is similar to the PERCENT\_RANK() function which works as follows:

Given a set of values in a data set, this function calculates the rank of each individual value relative to the entire data set. This function also returns a percentage.

Back to the CUME\_DIST() function. We will look at two examples, an easy one and then one that will query our sales data warehouse.

Let’s look at a simple example first so we can see how the prior description works on a small data set. Please refer to listing 3.1a below:

***Listing 3.1a – a Simple Example***

USE TEST

GO

DECLARE @CumDistDemo TABLE (

Col1 VARCHAR(8),

ColValue INTEGER

);

INSERT INTO @CumDistDemo VALUES

('AAA',1),

('BBB',2),

('CCC',3),

('DDD',4),

('EEE',5),

('FFF',6),

('GGG',7),

('HHH',8),

('III',9),

('JJJ',10)

SELECT Col1,ColValue,

CUME\_DIST() OVER(

ORDER BY ColValue

) AS CumeDistValue,

A.RowCountLE,

B.TotalRows,

CONVERT(DECIMAL(10,2),A.RowCountLE)

/ CONVERT(DECIMAL(10,2),B.TotalRows) AS MyCumeDist

FROM @CumDistDemo CDD

CROSS APPLY (

SELECT COUNT(\*) AS RowCountLE FROM @CumDistDemo

WHERE ColValue <= CDD.ColValue

) A

CROSS APPLY (

SELECT COUNT(\*) AS TotalRows FROM @CumDistDemo

) B

GO

A simple table variable called @CumeDistDemo is declared and loaded with 10 rows of data. Only two columns are created. The first is called Col1 and acts as a key column while the ColValue column holds the data. In our case these are the numbers 1 through 10. Really simple.

The query utilizes the CUME\_DIST() function and an OVER( ) clause. No PARTITION BY clause is included but the ORDER BY clause is used to order the partition rows by the ColValue columns in ascending order.

We need to include the formula for calculating cumulative distribution so we can see how it works and compare it to the SQL Server version. Here is the formula:

Count of rows with values less than or equal to the current column value divided by the total number of rows.

To get the values for this formula two CROSS APPLY operators are used. The first one calculates the number of rows with values less than or equal to the current column value and the second calculates the total number of rows in the data set.

This query can probably be optimized but for our purposes it should illustrate how the CUME\_DIST() function works.

The values returned by the queries in the CROSS APPLY block are used in the formula in the select clause:

CONVERT(DECIMAL(10,2),A.RowCountLE)

/ CONVERT(DECIMAL(10,2),B.TotalRows) AS MyCumeDist

Let’s execute the query and see what we get. Please refer to figure 3.1 below:

Table

Description automatically generated

***Figure 3.1 – Two Ways of Calculating Cumulative Distribution.***

Looks good and predictable, I think! The MyCumeDist column has the values calculated by the formula we created, and the column called CumeDistValue contains the values the SQL Server function calculated. Maybe a little formatting is needed so they look the same but the values match.

Now that we understand how this function works let’s create a query that goes against our sales data. We will use the strategy we used in the prior chapter. That is, we will use a CTE to set up the data and then the functions we are learning are in the query that accesses the CTE.

Please refer to Listing 3.1b below:

***Listing 3.1b – The CUME\_DIST() function in action***

WITH CustSales (

SalesYear,SalesQuarter,SalesMonth,CustomerNo,StoreNo,CalendarDate,SalesTotal

)

AS

(

SELECT YEAR(CalendarDate) AS SalesYear

,DATEPART(qq,CalendarDate) AS SalesQuarter

,MONTH(CalendarDate) AS SalesMonth

,ST.CustomerNo,

ST.StoreNo,

ST.CalendarDate,

SUM(ST.UnitRetailPrice \* ST.TransactionQuantity) AS SalesTotal

FROM StagingTable.SalesTransaction ST

GROUP BY ST.CustomerNo

,ST.StoreNo

,ST.CalendarDate

,ST.UnitRetailPrice

,ST.TransactionQuantity

)

SELECT SalesYear

,SalesQuarter

,SalesMonth

,CustomerNo

,SUM(SalesTotal) AS MonthlySalesTotal

,CUME\_DIST() OVER (

PARTITION BY SalesYear

ORDER BY SUM(SalesTotal)

) AS CumeDist

FROM CustSales

WHERE SalesYear IN(2010,2011)

AND CustomerNo = 'C00000001'

GROUP BY SalesYear

,SalesQuarter

,SalesMonth

,CustomerNo

GO

Our usual CTE is set up to pull transaction data from the SalesTransaction table so we can pull out the individual date objects: year, quarter, and month. The sum of the sales total is also calculated by summing the results of multiplying the UnitRetailPrice by the TransactionQuantity columns. (Not efficient I have to admit, I could have used the TotalSalesAmount column and avoided the formula, oh well.)

The CUME\_DIST() function uses an OVER() clause which includes a PARTITION BY clause and an ORDER BY clause. A partition is set up over the SalesYear column and the row set of the partition is sorted by the sum of the sum SalesTotal column.

Results are summed up again because we eliminated the store number and only want to see totals by year, quarter, month, and customer. A WHERE clause is included to filter out the results for only one customer and for only two years. I did this to keep the result set small, but you can remove them to see what you get.

Please refer to the results in figure 3.2 below:

Graphical user interface, application, table, Excel

Description automatically generated

***Figure 3.2 – Query Results for the Test Script***

The results were sorted by the sum of the Sales Total column in the OVER() clause in order to see the ascending values of the sales totals and cumulative distribution results so you can see how the formula was applied. Sorting by the sales amounts in ascending order lines things up nicely.

Try out the example by sorting results by sales year, quarter, and month which is what a business analyst would normally want to see.

**Performance Considerations**

Let’s generate an estimated query execution plan in the usual manner. The results are long, so we only look at the right half of the display as this is where the most interesting statistics for the tasks are displayed. The other tasks or steps in the plan are mostly 0 cost so we ignore them in the interest of space constraints.

Please refer to figure 3.3 below:

Graphical user interface

Description automatically generated with low confidence

***Figure 3.3 – Estimated Query Plan for this Script***

Yikes! Look at the three sort steps. At least there is an index seek and no other indexes were suggested by the estimated performance tool.

The index seek tasks costs 25% of the total execution plan and the sort step next to it costs 28%. A second sort step at 25% can be seen so a lot of sorting seems to occur with this query. A third sort step costs 17%!

This is an indication that we could improve performance by staging the initial data generated by the CTE in some sort of report or staging table and building indexes to assist when the main query is executed. A large benefit would be that you load this table once a day and assuming that users are happy with 24-hour data refresh, the processing of the loading is not included when the users execute the main query that has the window function.

By the way, if you position your mouse over each sort step, you will see that the following columns are sorted:

* First sort step: store and calendar date.
* Second sort step: sort by customer.
* Third sort step: sort by customer again.

Now we are second-guessing the query plan estimator. On your own, experiment by creating one or two indexes to support the required sorting based on the above information. See if the sort steps are eliminated or at least reduced. Keep in mind that the more indexes you create on a table the slower performance will be when loading or editing the table rows.

I have included a script based on the CTE logic in the listings that stages the data in a table. An index is also created to support the sorting (the index is suggested by the estimated query plan tool). The results are that the index seek costs a little less, but the sorting still exists and is a bit higher.

Where the improvement appears is in the query statistics. In the first query, when accessing the sales transaction table, the logical reads are 23 but this is reduced to 10 when using the staging table. The physical reads in the first query are 1 while they are 0 in the query accessing the staging table.

This strategy seems to work. Once again, try it on your own before looking at my example.

Please refer to table 3.1 below for some statistics:

***Table 3.1 – Execution Statistics – No Staging Table Versus Staging Table***

| **SQL Server Parse & Compile Time** | **Existing Index** | **New Index** |
| --- | --- | --- |
| CPU Time (ms) | 0 | 0 |
| elapsed time (ms) | **77** | **6** |
|  |  |  |
| **Statistic (work table)** | **Existing Index** | **New Index** |
| Scan Count | 26 | 26 |
| Logical Read | **198** | **196** |
| Physical Reads | 0 |  |
| read-ahead reads | 0 |  |
|  |  |  |
| **Statistic (SalesTransaction)** | **Existing Index** | **New Index** |
| Scan Count | **1** | **2** |
| Logical Read | **23** | **10** |
| Physical Reads | **0** | **1** |
| read-ahead reads | **0** | **3** |
|  |  |  |
| **SQL Server Execution Times** | **Existing Index** | **New Index** |
| CPU Time (ms) | 0 | 0 |
| elapsed time (ms) | 35 | 2 |

I reran the original query and a query that accesses the staging table. No changes were made to the original query (that contains the CUME\_DIST() function) except that it now accesses the staging table instead of the CTE. Also, some new indexes were created.

The CTE in the original query was modified so it loads the staging table once. Assume in a real production environment it would be run nightly at off hours.

As can be seen, the first set of elapsed time went from 77 to 6. Good start. The scan counts remained the same and the logical read went down a bit (198 versus 196) for the work table.

Looking at the SalesTransaction table statistics the scan count went from 1 to 2 while the logical reads went down to 10. This is also good!

Now we get to physical reads and read-ahead reads. In the original query they went from 0 to 1 and 3 respectively. Not so good. Lastly, CPU time was zero in both cases, but it went from 35 to 2 so some major improvement here.

Some final comments, when you perform this analysis, you will get different results due to your hardware configuration. You need to run these tests several times to get a complete picture of what works and does not work. Do not forget to run DBCC so you always start off with clean buffers. Not executing this command before your run each test will give you misleading results!

**Note:** by the way, DBCC stands for database consistency check. A very powerful and useful tool. I suggest you research the Microsoft SQL Server documentation on this.

Let’s check out the client statistics for the first query and the query that uses the staging table. .The Client Statics button can be found on the menu bar to the right of the button with the little green checkbox. Click on it to turn it on or off.

Please refer to figure 3.4 below:

Graphical user interface, table

Description automatically generated

***Figure 3.4 – Client Statistics for the First Query***

This is interesting, the top report is for the first query. I am only showing client processing time, total execution time and wait time on server replies. They are respectively 25, 126 and 101.

The second client statistics report shows client processing time at 11 (good) but check out the total execution time at 5028 and wait on server replies at 5017. These went up significantly.

So, there are a few takeaways from our discussion. The first is you have a set of statistics and plans generated by some valuable tools that you should use when performing query design and tuning.

Second, you need to run the tests several times to get an “average” view of the results. Creating indexes seems to be a good idea but most of the time they will give you the same results or worse results than the indexes suggested by the query estimated execution plan tool. If you forget to run the DBCC command to clear the buffer, you will get misleading results that make you think performance is improving.

Lastly, working on small sets of data like less than 200,000 will not show much difference when you are running tests. You need to see how everything works when dealing with millions of rows or more! (Look over the load scripts for the Sales data warehouse and see if you can load 5 – 10 million rows in the fact tables we are using by modifying my load script. Make sure you make a backup).

Our next function in our bag of tricks is the PERCENT\_RANK() function.

**PERCENT\_RANK() Function**

Next on deck is the PERCENT\_RANK() function. What does it do?

Using a data set like results from a query, partition or a table variable, this function calculates the relative rank of each individual value relative to the entire data set (as a percentage). You need to multiply results (the value returned is a float) by 100.00 or use the FORMAT()function to convert the results to a percentage.

Also, it works a lot like the CUME\_DIST() function, at least that’s what the Microsoft documentation says. We discussed this function at the beginning of the chapters so go back and take a look at what it does if you forgot.

You will also see that it works a lot like the RANK()function (in theory anyway). The RANK() function returns the position of the value while PERCENT\_RANK()returns a percentage value. We will use this function together with the CUME\_DIST() function just discussed so we can compare the results and also see what effect it will have on the query plan we will generate shortly.

But first let’s examine another simple example, like the prior discussion so we can clearly see the results generated from a simple data set.

I will also include a homegrown version of the percent rank calculation based on the following formula:

(current row column value - 1) / (total row count of the data set - 1)

Simple! We will use the same table variable data as the CUME\_DIST() function.

For our homegrown code, we need to simulate the RANK() function which will be introduced in the next chapter. I also include the RANK() function in the SELECT clause so we can compare its results with our custom code.

Please refer to Listing 3.2 below:

***Listing 3.2 – A Simple Percent Rank Example***

DECLARE @CumeDistDemo TABLE (

Col1 VARCHAR(8),

ColValue DECIMAL(10,2)

);

INSERT INTO @CumeDistDemo VALUES

('AAA',1.0),

('BBB',2.0),

('CCC',3.0),

('DDD',4.0),

('EEE',5.0),

('FFF',6.0),

('GGG',7.0),

('HHH',8.0),

('III',9.0),

('JJJ',10.0)

SELECT Col1,ColValue,A.RowCountLTE AS MyRank,

RANK() OVER(

ORDER BY ColValue

) AS SQLRank,

PERCENT\_RANK() OVER(

ORDER BY ColValue

) AS PCTRank,

/\* current value rank - 1 /data sample total row count - 1 \*/

(RANK() OVER(

ORDER BY ColValue

) - 1.0) / CONVERT(DECIMAL(10,2),(

SELECT COUNT(\*) AS SampleRowCount

FROM @CumeDistDemo) - 1.0

) AS MyPctRank

FROM @CumeDistDemo CDD

CROSS APPLY (

SELECT COUNT(\*) AS RowCountLTE FROM @CumeDistDemo

WHERE ColValue <= CDD.ColValue

) A

GO

Notice I repurposed the @CumeDistDemo table variable (Too lazy to rename it!).

As before, the data values are incremented from 1.0 to 10.0 sequentially. This will make it easy to correlate the values to the rank results. A single CROSS APPLY block is used in the FROM clause so we can calculate the total row count of the data set less than or equal to the current row:

CROSS APPLY (

SELECT COUNT(\*) AS RowCountLTE

FROM @CumeDistDemo WHERE ColValue <= CDD.ColValue

) A

Applying the simple formula, the following code delivers the percent rank:

(RANK() OVER(

ORDER BY ColValue

) - 1.0) / CONVERT(DECIMAL(10,2),(SELECT COUNT(\*) AS SampleRowCount FROM @CumeDistDemo) - 1.0) AS MyPctRank

Notice that we are using the RANK() function to calculate the ranking assignments (we will discuss it in the next chapter). If we sort the data being ranked, namely the ColValue column in ascending order, as a trick, we could also use the ROW\_NUMBER() function as shown below (we will also discuss this in the next chapter):

ROW\_NUMBER() OVER(

ORDER BY ColValue

) AS RowNumberAsRank,

One final alternative is to just use the column RowCountLTE as the rank:

/\* current value rank - 1 /sample total row count - 1 \*/

FORMAT(

(RowCountLTE - 1.0) /

CONVERT(DECIMAL(10,2),(

SELECT COUNT(\*) AS SampleRowCount

FROM @CumeDistDemo

) - 1.0),'P'

) AS MyPctRank

Try to modify the code by using all versions of percentile rank so you can verify the results are the same and the formula works. Try it on your own before looking at the bonus script.

Let’s look at the results. Please refer to Figure 3.5 below:

Table

Description automatically generated

***Figure 3.5 – Calculating Percent Rank with SQL Server Function & Home-Grown script***

Here are the sorted results. Both percent rank results match and so do both the rank calculations used with the SQL Server function and the homegrown function. You can use the FORMAT() function to convert the results to percentages if you like. For example:

FORMAT(

PERCENT\_RANK() OVER(

ORDER BY ColValue

),'P'

) AS PCTRank,

Just wrap the PERCENT\_RANK() code block with parenthesis, use the FORMAT() function and include the ‘P’ parameter before the last parenthesis. Very easy to use.

**Tip:** in my opinion, creating your own version of a function based on the formula is a great way to understand what the function does.

Let’s try this function out on our Sales data warehouse.

Please refer to Listing 3.3 below:

***Listing 3.3 – The PERCENT\_RANK() Function in Action***

WITH CustSales (

SalesYear,SalesQuarter,SalesMonth,CustomerNo,

StoreNo,CalendarDate,SalesTotal

)

AS

(

SELECT YEAR(CalendarDate) AS SalesYear

,DATEPART(qq,CalendarDate) AS SalesQuarter

,MONTH(CalendarDate) AS SalesMonth

,ST.CustomerNo

,ST.StoreNo

,ST.CalendarDate

,SUM(ST.TotalSalesAmount) AS SalesTotal

FROM StagingTable.SalesTransaction ST

GROUP BY ST.CustomerNo

,ST.StoreNo

,ST.CalendarDate

)

SELECT SalesYear

,SalesQuarter

,SalesMonth

,CustomerNo

,SUM(SalesTotal) AS MonthlySalesTotal

,CUME\_DIST() OVER (

PARTITION BY SalesYear

ORDER BY SUM(SalesTotal)

) AS CumeDist

,PERCENT\_RANK() OVER (

PARTITION BY SalesYear

ORDER BY SUM(SalesTotal)

) AS PctRank

FROM CustSales

WHERE SalesYear IN(2010,2011)

AND CustomerNo = 'C00000001'

GROUP BY SalesYear

,SalesQuarter

,SalesMonth

,CustomerNo

GO

I do not think we need to go over the CTE again except to highlight that we are summing up sales by store. We could use the staging table we created for the prior example but let’s work with the CTE again. Also, I included the CUME\_DIST() function in the SELECT clause so we can compare the results between the two functions.

The OVER() clause uses the same ORDER BY and PARTITION by clause. We want to sum up values by year, quarter month and customer. We are not using the store number column so we need to apply the SUM() function across the year, quarter, month and customer number columns. This means we need to use a GROUP BY clause.

Once again, we use a WHERE clause to filter by years 2010 and 2011 and for a single customer (“C0000001”) in order to keep the result set small. Feel free to download the script and modify the WHERE clause once you are comfortable with how the query works so you load more years (make sure your laptop or desktop is powerful).

Let’s check out the results. Please refer to the partial results in figure 3.6 below:

Graphical user interface, application, table

Description automatically generated

***Figure 3.6 – Query Results for the Test Script***

I changed the query a bit by eliminating the WHERE clause filter and including an ORDER BY clause. This change resulted in 16,380 rows being returned once the query was executed.

Lastly, there is a bonus query after this example in the script that uses the homegrown version of the percent rank logic and compares it to the SQL Server function using the formula we discussed.

The performance was still under 1 second. Let’s see what the estimated query plan and statistics look like next.

**Performance Considerations**

Let’s run an estimated query plan and execute the query so it generates IO and TIME statistics. Once again, no new index was suggested so the indexes we created earlier seem to be utilized.

Please refer to figure 3.7 below for the estimated query plan:

Graphical user interface, text

Description automatically generated

***Figure 3.7 – Estimated Execution Plan Results***

Working from right to left we encounter an index seek at a cost of 17% and a sort task at a cost of 47%. The only other expensive step is a second sort task at 15%. The remaining tasks are either low value or zero, so we ignore them. Usually, in a real production environment you want to at least be aware of them but mainly focus on the expensive steps to determine if you eliminate them or at least reduce the cost.

Sometimes creating indexes based on the columns of the dimensions might help. Check out the sort steps and see if an index based on the columns referenced in the sort steps are needed.

There does not seem much we can do for now so let’s look at the performance statistics to see what they tell us.

Please refer to table 3.2 below:

***Table 3.2 – Performance Analysis***

| **SQL Server Parse & Compile Time** | **Existing Index** | **New Index** |
| --- | --- | --- |
| CPU Time (ms) | 0 |  |
| elapsed time (ms) | 37 |  |
|  |  |  |
| **Statistic (work table)** | **Existing Index** | **New Index** |
| Scan Count | 26 |  |
| Logical Read | 196 |  |
| Physical Reads | 0 |  |
| read-ahead reads | 0 |  |
|  |  |  |
| **Statistic (SalesTransaction)** | **Existing Index** | **New Index** |
| Scan Count | 1 |  |
| Logical Read | 17 |  |
| Physical Reads | 1 |  |
| read-ahead reads | 14 |  |
|  |  |  |
| **SQL Server Execution Times** | **Existing Index** | **New Index** |
| CPU Time (ms) | 0 |  |
| elapsed time (ms) | 30 |  |

The scan count step was high on the worktable at 26. The logical read statistic also seems a bit high for both the worktable and the SalesTransactionTable, at 196 and 17 respectively. Lastly, the SalesTransaction table has a read-ahead at 14. Looks like some improvement might be called for. Let’s look at the client statistics next.

Please refer to figure 3.8 below:

Graphical user interface, table

Description automatically generated

***Figure 3.8 – Client Statistics Analysis.***

Let’s just look at the time statistics at the bottom. The others are important also, but these tell us some performance stats we need to know. Client processing time was 11, total execution time was a bit high at 103 and wait on server replies was at 92.

Sometimes creating indexes is not enough. Other solutions could include modifying the query, utilizing temporary or report tables or even creating memory optimized tables. This last option loads all table rows to memory if you have enough, that is. We will discuss this next.

**High Performance Strategy**

In this section we create a memory optimized table and load it with the base data (this replaces the CTE query in the prior query). This strategy assumes that the table is loaded nightly once a day and the users query the memory optimized table the next day.

This strategy also replaces the strategy of the staging or report table we discussed earlier as we are assuming tables that exist in memory will give faster performance than tables that exist on physical disk. Granted, if one runs out of memory, SQL Server will have to access data on physical disk anyway. That is why the steps to create a memory optimized table include creating a dedicated file group and a physical file (or two).

Below are the steps to take to create a memory optimized table.

**Step 1: check compatibility level**.

First things first. Make sure your computer or laptop supports an optimized memory table by checking the compatibility level of SQL Server by running the simple query below.

Please see listing 3.4 below:

***Listing 3.4 – Check Compatibility Level***

SELECT d.compatibility\_level

FROM sys.databases as d

WHERE d.name = Db\_Name();

GO

You should be at level 130 or greater. If you are, we need to set a couple of system parameters. If not, read this section anyway so you know what it is all about!

**Step 2: set parameter MEMORY\_OPTIMIZED\_ELEVATE\_TO\_SNAPSHOT to ON.**

Please see listing 3.5 below:

***Listing 3.5 – Set Memory Optimized Elevate to Snapshot***

ALTER DATABASE [APSAles]

SET MEMORY\_OPTIMIZED\_ELEVATE\_TO\_SNAPSHOT = ON;

GO

**Step 3 -Next, add a dedicated file group for memory optimized data by running the following command.**

Please see listing 3.6 below:

***Listing 3.6 – Add File Group for Memory Optimized Data***

ALTER DATABASE APSales

ADD FILEGROUP APSalesMemOptimized CONTAINS MEMORY\_OPTIMIZED\_DATA;

GO

Notice that a directive needs to be included that specifies the file group will contain memory optimized data.

**Step 4: create a dedicated file for memory optimized tables.**

Next, we add a file to the file group in the usual manner. Please see listing 3.7 below:

***Listing 3.7 – Add a file to the file group***

ALTER DATABASE APSales

ADD FILE (

name='APSalesMemoOptData',

filename=N'D:\APRESS\_DATABASES\AP\_SALES\MEMORYOPT\AP\_SALES\_MEMOPT.mdf'

)

TO FILEGROUP APSAlesMemOptimized

GO

**Step 5: create the memory optimized table.**

Next, we create a table in the usual manner but need to add a bit of code that defines it as a memory optimized table. Please see listing 3.8 below:

***Listing 3.8 – Create Memory Optimized Table***

CREATE TABLE [SalesReports].[MemorySalesTotals](

[SalesTotalKey] INTEGER NOT NULL IDENTITY PRIMARY KEY NONCLUSTERED,

[SalesYear] [int] NOT NULL,

[SalesQuarter] [int] NOT NULL,

[SalesMonth] [int] NOT NULL,

[CustomerNo] [nvarchar](32) NOT NULL,

[StoreNo] [nvarchar](32) NULL,

[CalendarDate] [date] NOT NULL,

[SalesTotal] [decimal](21, 2) NULL

)

WITH (

MEMORY\_OPTIMIZED = ON,

DURABILITY = SCHEMA\_AND\_DATA

);

GO

We need to make sure to identify a primary key and set two parameters at the end of the CREATE TABLE command:

MEMORY\_OPTIMIZED = ON,

DURABILITY = SCHEMA\_AND\_DATA

**Step 6: check that it was created.**

Next, we need to query two system tables to make sure the table is registered as a memory optimized table. Please see listing 3.9 below:

***Listing 3.9 – Check the Filegroups and Database\_files Tables***

SELECT g.name, g.type\_desc, f.physical\_name

FROM sys.filegroups g JOIN sys.database\_files f ON g.data\_space\_id = f.data\_space\_id

WHERE g.type = 'FX' AND f.type = 2

GO

The sys.filegroups and sys.database\_files tables have a lot of interesting columns and I suggest you check them out. For our little discussion I will only list the basic columns to save space.

**Step 7: Load the Memory Optimized Table.**

Now it’s time to load the table. Please see listing 3.10 below:

***Listing 3.10 – Load the Memory Optimized Table***

INSERT INTO [SalesReports].[MemorySalesTotals]

SELECT YEAR(CalendarDate) AS SalesYear

,DATEPART(qq,CalendarDate) AS SalesQuarter

,MONTH(CalendarDate) AS SalesMonth

,ST.CustomerNo

,ST.StoreNo

,ST.CalendarDate

,SUM(ST.UnitRetailPrice \* ST.TransactionQuantity) AS SalesTotal

FROM StagingTable.SalesTransaction ST

GROUP BY ST.CustomerNo

,ST.StoreNo

,ST.CalendarDate

,ST.UnitRetailPrice

,ST.TransactionQuantity

GO

Remember, this was the query used in the CTE from the prior example we discussed. Now we are ready for our performance analysis.

**Step 8: Check estimated query plan.**

Before we run the query, we need to check the estimated query plan in the usual manner. See if an index is suggested. If it is, create the index and re-run the query plan estimator tool. Once the index is created, set the parameters to display performance statistics.

**Step 9: Run the Query.**

Clear buffers with DBCC, make sure IO and TIME statistics are set and run the query.

Please see listing 3.11 below:

***Listing 3.11 – Query the Memory Optimized Table***

DBCC dropcleanbuffers;

CHECKPOINT;

GO

SET STATISTICS IO ON

GO

SET STATISTICS TIME ON

GO

SELECT SalesYear

,SalesQuarter

,SalesMonth

,CustomerNo

,SUM(SalesTotal) AS MonthlySalesTotal

,CUME\_DIST() OVER (

PARTITION BY SalesYear

ORDER BY SUM(SalesTotal)

) AS CumeDist

,PERCENT\_RANK() OVER (

PARTITION BY SalesYear

ORDER BY SUM(SalesTotal)

) AS PctRank

FROM [SalesReports].[MemorySalesTotals]

WHERE SalesYear IN(2010,2011)

AND CustomerNo = 'C00000001'

GROUP BY SalesYear

,SalesQuarter

,SalesMonth

,CustomerNo

GO

Always clear the buffers by running the DBCC command. Next run the two commands to set the IO and TIME statistics on and then run the query.

***Step 10: - create the suggested index.***

Here is the code for the suggested index. I placed it here, but you should run it after you check out the first estimated query plan. Come to think of it, run the query prior to creating the index so you can record the performance statistics.

Please see listing 3.12 below:

***Listing 3.12 – Create the Suggested Index***

ALTER TABLE SalesReports.MemorySalesTotals

ADD INDEX ieCustNoSaleYearMemTable

NONCLUSTERED (CustomerNo,SalesYear)

GO

**Step 11: - Create a Second Estimated Index Plan**

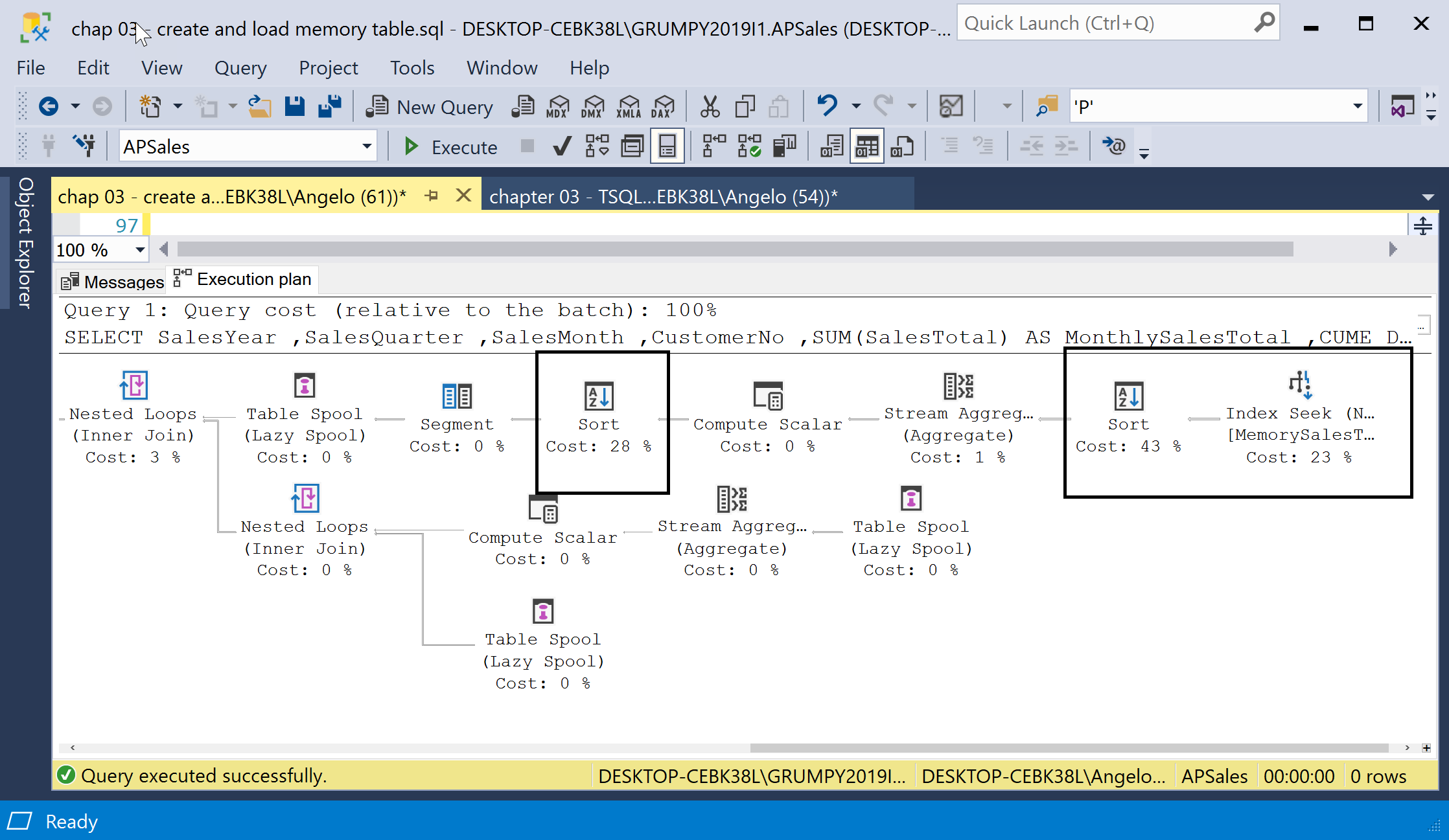
After the index is created, create a second estimated query plan. Maybe you can do this in two horizontally placed query panes. The top one would contain the query plan without the suggested index and the bottom one would contain the query plan after the suggested index is created so that you can compare the two.

**Step 12: - Re-run the query and make sure all statistics are turned on**

Make sure you clear buffers with **DBCC** when checking estimated query plans and running queries with statistics on.

I ran the estimated query plan twice. The first time was to see what index the estimator suggested. The second time was after I created the suggested index.

Please refer to figure 3.9 below for the estimated query plan after the index was created:



***Figure 3.9 – The Estimated Query Plan***

Reading the index plan right to left, we notice an index seek at a cost of 23%. Following it is a sort step at 43%. Ignoring the tasks with zero or minimal costs, we see a second sort step at 28%. There is a nested loop join at 3%. Not shown are more tasks to the left of the nested loop join but they were all zero.

Download the code yourself and check out the plans and statistics. It is worthwhile learning how to do this performance analysis activity.

Let’s wrap things up by comparing statistics generated for the CTE version of the query and the query that utilizes a memory optimized table.

Please refer to table 3.x below for CTE versus Memory Table Statistics Analysis

***Table 3.3 – Comparing CTE versus Memory Table Statistics***

| **SQL Server Parse & Compile Time** | **Using a CTE** | **Memory Table** |
| --- | --- | --- |
| CPU Time (ms) | **46** | **0** |
| elapsed time (ms) | **83** | **0** |
|  |  |  |
| **Statistic (work table)** | **Existing Index** | **New Index** |
| Scan Count | 26 | 26 |
| Logical Read | 196 | 198 |
| Physical Reads | 0 | 0 |
| read-ahead reads | 0 | 0 |
|  |  | 0 |
| **Statistic (SalesTransaction)**  **& MemorySalesTotals** | **Existing Index** | **Memory Table** |
| Scan Count | 1 | N/A |
| Logical Read | **23** | N/A |
| Physical Reads | 1 | N/A |
| read-ahead reads | 20 | N/A |
|  |  |  |
| **SQL Server Execution Times** | **Existing Index** | New Index |
| CPU Time (ms) | **0** | **0** |
| elapsed time (ms) | **34** | **0** |

This is interesting. For the memory optimized table strategy, we only have statistics for the worktable. No statistics for the MemorySalesTotal table which is now loaded in memory.

We see significant improvement in both sets of CPU time and elapsed time for parse and compile and for execution times. With the memory optimized table, they are all zero. Lastly, for the worktable the results are about the same.

Looks like the memory optimized table yields significant performance results.

One last comment, while performing this type of analysis, it might be a good idea to store statistics in a spreadsheet so you can analyze and compare. Run several tests that:

* Generate query plans
* Generate IO and TIME statistics
* Generate client statistics.

And another last comment, as stated earlier, sometimes you need to go beyond index creation and look at other modifications like report and memory enhanced tables (or even the query itself).

The last resort is getting management to upgrade your servers with more and/or faster memory and fast disks or SAN solutions. (A nice solid-state drive for TEMPDB would also help!)

Let’s forge ahead with our next two analytical functions.

**LAST\_VALUE() & FIRST\_VALUE()**

So, what do these related pairs of functions do for us? The names give us a hint.

Given a set of sorted values, the FIRST\_VALUE() function will return the first value.

Given a set of sorted values, the LAST\_VALUE() function will return the last value.

Pretty simple, let’s dive right into an example. This time we eliminated the CTE and utilized the memory enhanced table we created earlier.

Please refer to Listing 3.13 below:

***Listing 3.13 – The FIRST\_VALUE() & LAST\_VALUE() Function in Action***

SELECT SalesYear

,SalesQuarter

,SalesMonth

,CustomerNo

,SUM(SalesTotal) AS MonthlySalesTotal

,FIRST\_VALUE(SUM(SalesTotal)) OVER (

PARTITION BY SalesYear

ORDER BY SalesMonth

) AS SalesTotalFirstValue

,LAST\_VALUE(SUM(SalesTotal)) OVER (

PARTITION BY SalesYear

ORDER BY SalesMonth

) AS SalesTotalLastValue

,FIRST\_VALUE(SUM(SalesTotal)) OVER (

PARTITION BY SalesYear

ORDER BY SalesMonth

) -

LAST\_VALUE(SUM(SalesTotal)) OVER (

PARTITION BY SalesYear

ORDER BY SalesMonth

) AS Change

,CASE

WHEN (

FIRST\_VALUE(SUM(SalesTotal)) OVER (

PARTITION BY SalesYear

ORDER BY SalesMonth

) - -

LAST\_VALUE(SUM(SalesTotal)) OVER (

PARTITION BY SalesYear

ORDER BY SalesMonth

)

) > 0 THEN 'Sales Increase'

WHEN (

FIRST\_VALUE(SUM(SalesTotal)) OVER (

PARTITION BY SalesYear

ORDER BY SalesMonth

) -

LAST\_VALUE(SUM(SalesTotal)) OVER (

PARTITION BY SalesYear

ORDER BY SalesMonth

)

) < 0 THEN 'Sales Decrease'

ELSE 'No change'

END AS [Sales Performance]

FROM SaleReports.MemorySalesTotals

WHERE SalesYear IN(2010,2011)

AND CustomerNo = 'C00000001'

GROUP BY SalesYear

,SalesQuarter

,SalesMonth

,CustomerNo

GO

The usual OVER() structure for these functions is used. A PARTITION BY clauses sets up partitions by sales year. The ORDER BY clause sorts the partition rows by the Sales Month column.

We sum a second time as we want the results by customer only as the memory enhanced table rows are summed by store and customer plus the usual year, quarter, and month columns.

Please refer to the partial results in figure 3.10 below:

Graphical user interface, table, Excel

Description automatically generated

***Figure 3.10 – Query Results for the Test Script***

Results are returned for two years. Notice the first and last values for the first row. They are equal as the window frame consists of only one row. When we move to the second row the first value remains the same, but the second value now reflects the monthly sales totals for February.

As the window frames get larger the last value is adjusted to reflect the last month in the partitions. I also included a formula that calculates the change between the first and last total sales values and a little message that shows you if sales increased or decreased or if no change occurred at all.

This might make a pretty good report for our analysts if we include all years. A tool like Reporting Services can use this query and include a drop-down list so analysts can select one, two or more years.

**Performance Considerations**

Let’s generate an estimated query plan in the usual manner and see if the memory optimized tables plus the index created to support it yields positive results.

Please refer to figure 3.11 below:

Graphical user interface, application

Description automatically generated

***Figure 3.11 – Estimated Query Plan using a Memory Optimized able***

Looks good. Reading the steps right to left we see an index seek at a cost of 35%, a sort at a cost of 63% and then a stream aggregate step at 1%. Ignoring the 0% cost steps we have a window spool task at 1%. The rest of the steps not shown are all zeroes so we can ignore them for the purpose of our discussion (In a real production environment, look at all tasks!).

Side comment, I added an index based on the following command:

ALTER TABLE [SalesReports].[MemorySalesTotals]

ADD INDEX ieSalesYearQuarterMonth(SalesYear,SalesQuarter,SalesMonth)

GO

I also added an ORDER BY clause that sorted rows by the year, quarter, and month columns. I generated another estimated query plan and the first step, the index seek tasks, went down to 25% from 35% and the sort step went down to 46% from 63%.

So sometimes you need to create other indexes not suggested by the query plan estimator.

Now we check our IO and TIME statistics that were generated when the query was run.

Please refer to table 3.4 below:

***Table 3.4 – TIME and IO Statistics Analysis Report***

| **SQL Server Parse & Compile Time** | **First Run** | **Second Run** |
| --- | --- | --- |
| CPU Time (ms) | 15 | **5** |
| elapsed time (ms) | 47 | **5** |
|  |  |  |
| **Statistic (work table)** | **Existing Index** | **New Index** |
| Scan Count | 26 | 26 |
| Logical Read | 145 | 145 |
| Physical Reads | 0 | 0 |
| read-ahead reads | 0 | 0 |
|  |  |  |
|  |  |  |
| **SQL Server Execution Times** | **Existing Index** | **New Index** |
| CPU Time (ms) | 0 | 0 |
| elapsed time (ms) | **1** | **0** |

I ran the query twice. The first result had CPU and elapsed time values of 15 and 47 in the parse step. The second run had values of 5 for these two values. Most likely the query plan was cached so some reduction in time for parsing occurred. This illustrates the importance of running DBCC to clear the buffers each time you run the query with statistics turned on (like moving old plans out of cache).

Please refer to figure 3.12 below:

Graphical user interface, table

Description automatically generated

***Figure 3.12 – Client Statistics Performance Report.***

Keep in mind that running these performance tools will yield different results each time, not by much but they will most likely be different than the ones displayed in this book due to the difference in the computing platforms the readers are using. Let’s move on to our next set of functions.

**LAG() & LEAD()**

These are very important functions to include in your programming toolkit to analyze data. One of the activities your business analysts perform is to look back in time or forward in time within a set of data that is historical in nature. Of course, the data could be a snapshot that includes current and past data but no future data. These functions allow you to create powerful queries and reports to do exactly that.

Here’s what the LAG() function does:

This function is used to retrieve the previous value of a column relative to the current row column value. For example, given this month’s sales amounts for a product, pull last month’s sales amounts for the same product, or specify some offset, like 3 months ago or even a year or more depending on how much historical data you have collected and what your business users want to see.

Here’s what the LEAD() function does:

This function is used to retrieve the next value relative to the current row column. For example, given this month’s sales amounts for a product, pull next month’s sales amounts for the same product, or specify some offset, like 3 months in the future or even a year in the future.

These functions come particularly handy in a sales scenario where you need to analyze historical sales performance or current performance against past performance.

Now that we have a memory optimized table, we will use it instead of the CTE scheme we used for most of the examples in this chapter. Performance should be very fast!

Please refer to Listing 3.14 below:

***Listing 3.14 – The LAG() & LEAD() Function in Action***

SELECT

SalesYear,

SalesQuarter,

SalesMonth,

StoreNo,

ProductNo,

CustomerNo,

CalendarDate AS SalesDate,

SalesTotal,

LAG(SalesTotal) OVER (

PARTITION BY SalesYear,CustomerNo

ORDER BY SalesMonth,CustomerNo,CalendarDate

) AS LastMonthlySales,

LEAD(SalesTotal) OVER (

PARTITION BY SalesYear,CustomerNo

ORDER BY SalesMonth,CustomerNo,CalendarDate

) AS NextMonthylSales

FROM [SalesReports].[MemorySalesTotals]

WHERE StoreNO = 'S00005'

AND SalesYear = 2002

GO

This time we let the query rip with filters for the store number and year. The query generated 902 rows (still not a lot of rows as far as SQL Server is concerned).

The time to execute was less than one second. Please refer to the partial results in figure 3.13 below:

Graphical user interface, application, table, Excel

Description automatically generated

***Figure 3.13 – Query Results for the Test Script***

I scrolled down a bit so you can see the data patterns when the partition moves to the next partition. Notice the red arrows. We can see that the LastMonthlySales column points to the prior month’s value and that the NextMonthlySales column points to next month’s value in the SalesTotal column.

We can see the window frame action in rows 41 and 42. The next month’s sales amount is set to NULL when the partition changes and the prior month sales value in row 42 is also NULL as we are starting a new partition (Keep in mind the difference between a partition and a window frame within partition).

Download the script and remove the filter predicate for the year so you get all years for this store. If you want not only the years but all the stores you might need to fiddle a bit with the sort orders, so everything lines up correctly.

That’s why I always recommend starting with a small data set when creating a complex script or query. When you are sure it works correctly you can expand the scope by removing any filters.

**Performance Considerations**

Next, we see what kind of performance issues are related to these functions and some possible solutions to improve performance. As usual our first step is to create an estimated query plan in the usual manner. Make sure you run the DBCC command to clear buffers first.

Please refer to figure 3.14 below for the estimated execution plan:

Graphical user interface, application, Word

Description automatically generated

***Figure 3.14 – Estimated Execution Plan***

This estimated query plan shows that the query needs attention.

Reading from right to left we see a table scan at a cost of 78% followed by a filter at a cost of 15% and a sort step at a small cost of 5%. An index was suggested so let’s pull it out and give it a name.

Please refer to Listing 3.15 below for the suggested index:

***Listing 3.15 – Estimate Query Plan Tool Suggested Index***

ALTER TABLE [SalesReports].[MemorySalesTotals]

ADD INDEX ieSalesYearStoreNo

NONCLUSTERED ([SalesYear],[StoreNo])

GO

Notice we must use the ALTER command as memory enhanced tables do not support the CREATE INDEX command!

This index is based on the sales year and the store number column. Let’s create it, run the DBCC command again and generate a new estimated query plan.

Please refer to figure 3.15 below for the estimated plan after the index was created:

Graphical user interface, application, table

Description automatically generated

***Figure 3.15 – Estimated Query Plan after Index was Created***

An index scan at a cost of 36% replaces the table scan which had a cost of 78% so we are off to a good start. The filter task is gone but the sort step increased from 5% to 51%. Once again it seems when we use an index seek the sort step cost goes up. Let’s look at the second part of the plan by scrolling to the left.

Please refer to figure 3.16 below:

Graphical user interface, application, Word

Description automatically generated

***Figure 3.16 – The Left-Hand Side of the Estimated Query Plan***

Working from right to left again, notice the first window spool task. This has a cost of 5% and is used to reference data in the partition (which is in memory) multiple times. There are some other low-cost tasks, and we see a second window spool task. This also costs 5%. One thing to keep in mind, we should aim for 0% for window spool tasks. This means the action is in memory. If the value is greater than zero then the spool activity is on disk. (Will more memory help?)

Here are the final IO and TIME statistics once the index is created. The query was run two times. Each time the DBCC command was executed to clear the memory buffers (a little reminder …).

Please refer to table 3.5 below:

***Table 3.5 – IO and TIME Statistics for LAG() and LEAD() Functions***

| **SQL Server Parse & Compile Time** | **First Run** | **Second Run** |
| --- | --- | --- |
| CPU Time (ms) | **16** | **0** |
| elapsed time (ms) | **76** | **39** |
|  |  |  |
| **Statistic (work table)** | **New Index** | **New Index** |
| Scan Count | 0 | 0 |
| Logical Read | 0 | 0 |
| Physical Reads | 0 | 0 |
| read-ahead reads | 0 | 0 |
|  |  |  |
| **Memory Table** | **New Index** | **New Index** |
| Scan Count | N/A | N/A |
| Logical Read | N/A | N/A |
| Physical Reads | N/A | N/A |
| read-ahead reads | N/A | N/A |
|  | N/A | N/A |
| **SQL Server Execution Times** | **New Index** | **New Index** |
| CPU Time (ms) | 31 | 31 |
| elapsed time (ms) | **174** | **121** |

Seems like this query is very fast. Double digit times in milliseconds. Parse time went down in the second run, all zeros for the work table, no statistics for the memory table (makes sense as the table is in memory). Finally for the execution times the CPU time remained the same at 31ms and the elapsed time statistic went down from 174ms to 121ms.

**PERCENTILE\_CONT() and PERCENTILE\_DISC()**

What do these two functions do?

The PERCENTILE\_CONT() function works on a continuous set of data based on a required percentile so as to return a value within the data set that satisfies the percentile (you need to supply a percentile as a parameter, like .25, .5, .75 etc.).

With this function the value is interpolated. It usually does not exist, so it is introduced. Or by coincidence it could use one of the values in the data set if the numbers line up correctly.

What is continuous data anyway? Simply put, it is data that changes over specific periods of time. Like sales amounts of time periods over time. For example, total sales in a one-month period for a specific product.

The PERCENTILE\_DISC() function works on a discrete set of data based on a required percentile so as to return a value within the data set that satisfies the percentile. With this function the value exists in the data set, it is not interpolated like in the PERCENTILE\_CONT() function..

What is discrete continuous data anyway?

Simply put, discrete data values are numbers that have values that you can count over time. Like sales amounts for a product over days, months, and years. You can add these up to get a grand total sales amount.

Continuous data is data measured over a period of time. Like boiler temperatures in equipment over a 24-hour period. Summing up each boiler temperature makes no sense. Summing up the boiler temperatures and then dividing them by the time period to get an average does.

Let’s start off with a simple example before we dive into some analysis of our sales data.

Please refer to listing 3.16 below:

***Listing 3.16 The PERCENTILE\_CONT()& PERCENTILE\_DISC Function in Action***

DECLARE @ExampleValues TABLE (

TestKey VARCHAR(8) NOT NULL,

TheValue SMALLINT NOT NULL

);

INSERT INTO @ExampleValues VALUES

('ONE',1),('TWO',2),('THREE',3),('FOUR',4),('SIX',6),('SEVEN',7),('EIGHT',8),('NINE',9),('TEN',10),('TWELVE',12);

SELECT

TestKey,TheValue,

PERCENTILE\_CONT(.5)

WITHIN GROUP (ORDER BY TheValue)

OVER() AS PctCont, -- continuous

PERCENTILE\_DISC(.5)

WITHIN GROUP (ORDER BY TheValue)

OVER() AS PctDisc -- discrete

FROM @ExampleValues

GO

A very simple example. The data set consists of ten rows with two columns, a text value, and a numerical value.

The PERCENTILE\_CONT() function is passed .5 as a parameter to interpolate a value that will fit into the 50th percentile position in the data set. It treats the data set as a continuous distribution of values.

The PERCENTILE\_DISC() function is also passed .5 as a parameter to identify an existing value that falls into the 50th percentile position in the data set. It treats the data set as a discrete distribution of values. (You can pass any value less than or equal to 1, like .75 or .25).

Let’s see what the results are. Please refer to figure 3.17 below:

Graphical user interface, application

Description automatically generated

***Figure 3.17 – Percentile Continuous versus Discrete Results***

The percentile continuous value is interpolated while the percentile discrete is not. Notice that the 6.5 value falls between the values 6 and 7 in the TheValue column. The percentile discrete value is 6 and it exists in the TheValue column. Walk in the park!

With this powerful knowledge let’s try these functions out on our sales data. Please refer to Listing 3.6 below and we will go back to our friend, the CTE:

***Listing 3.17 – Percentile Continuous and Discrete Analysis***

WITH StoreSalesAnalysis (

SalesYear,SalesMonth,StoreNo,StoreName ,StoreTerritory,TotalSales

)

AS (

SELECT YEAR(CalendarDate) AS SalesYear

,MONTH(CalendarDate) AS SalesMonth

,StoreNo

,StoreName

,StoreTerritory

,SUM(TotalSalesAmount) AS TotalSales

FROM APSales.SalesReports.YearlySalesReport

GROUP BY YEAR(CalendarDate)

,MONTH(CalendarDate)

,StoreNo

,StoreName

,StoreTerritory

SELECT SalesYear

,SalesMonth

,StoreNo

,StoreName

,StoreTerritory

,FORMAT(TotalSales,'C') AS TotalSales

,FORMAT(PERCENTILE\_CONT(.5)

WITHIN GROUP (ORDER BY TotalSales)

OVER (

PARTITION BY SalesYear

),'C') AS PctCont

,FORMAT(PERCENTILE\_DISC(.5)

WITHIN GROUP (ORDER BY TotalSales)

OVER (

PARTITION BY SalesYear

),'C') AS PctDisc

FROM StoreSalesAnalysis

WHERE SalesYear IN(2010,2011)

AND StoreNo = 'S00004'

GO

I decided to format the percentiles to currency, so they look nice in the reports. Notice that the OVER() clause can only contain the PARTITION BY clause, no ORDER BY is allowed, you will see why. (In this case we partition by SalesYear.)

Also notice right above the OVER() clause we need to include a: WITHIN GROUP (ORDER BY TotalSales) directive. So, as you can see the syntax for these functions is slightly different.

Let’s check out the results. Please refer to the partial results in figure 3.18 below:

Graphical user interface, application, table, Excel

Description automatically generated

***Figure 3.18 – Query Results for the test script***

Notice how the results for percentile discreet appear in the Total Sales column. In other words, the results are not interpolated. This is not the case for percentile continuous, the results may appear in the total sales column but usually they are interpolated. The value $2795.38 does not appear as an actual value in the total sales column.

**Note:** SQL Server 2022 includes two new functions, APPROX\_PERCENTILE\_DIS() and APPROC\_PERCENTILE\_CONT() which we will cover in another chapter.

Let’s see what kind of estimated query plan these functions generate.

**Performance Considerations**

We generate an estimated query plan in the usual manner for the drop-down query selection in the menu bar. Right away we notice that an index is suggested. Let’s examine the plan.

Please refer to figure 3.19 below for the estimated query plan:

Diagram

Description automatically generated

***Figure 3.19 – Estimated Query Plan for Percentile Functions***

Reading from right to left we see an index seek tasks at a cost of 37% but apparently this index does not fulfill all performance requirements. Despite having an index laying around that was used, a new index was suggested so we will have to address this requirement.

The next expensive tasks are a sort task at cost of 19% and a nested loop join task at a cost of 10%. All other tasks have zero or low costs so we will not discuss them at this time. You should always examine each task, even the low ones so you understand how the query will be processed.

Below are the statistics when this query is executed with the existing index.

Please refer to table 3.6 below:

***Table 3.6 – Statistics with Existing Index***

| **SQL Server Parse & Compile Time** | **Existing Index** | **New Index** |
| --- | --- | --- |
| CPU Time (ms) | 0 | TBD |
| elapsed time (ms) | 98 | TBD |
|  |  |  |
| **Statistic (work table)** | **Existing Index** | **New Index** |
| Scan Count | 9 | TBD |
| Logical Read | 171 | TBD |
| Physical Reads | 0 | TBD |
| read-ahead reads | 0 | TBD |
|  |  |  |
| **Statistic (work file)** | **Existing Index** | **New Index** |
| Scan Count | 0 | TBD |
| Logical Read | **0** | TBD |
| Physical Reads | 0 | TBD |
| read-ahead reads | 0 | TBD |
|  |  |  |
| **Statistic (YearlySalesReport)** | **Existing Index** | **New Index** |
| Scan Count | 1 | TBD |
| Logical Read | 4033 | TBD |
| Physical Reads | 1 | TBD |
| read-ahead reads | 4041 | TBD |
|  |  |  |
| **SQL Server Execution Times** | **Existing Index** | **New Index** |
| CPU Time (ms) | 47 | TBD |
| elapsed time (ms) | 1025 | TBD |

Starting at the top, the elapsed time for parse and compile is 98ms. A work file and a worktable appear in the statistics and the YearlySalesReport table is also included. For the worktable we see a scan count of 9 and 171 logical reads. These can be improved.

Looking at the work file all values are 0 so we do not need to worry about this set of statistics.

Now the statistics for the YearlySalesReport need to be paid attention to. We see a scan count of 1, logical reads come in at 4033, physical reads at 1 and finally read ahead at 4041.

Lastly CPU time is 47ms and elapsed time is 1025ms. A bit high. Let’s create the suggested index and see if it helps lower these statistics.

Please refer to Listing 3.7 below that shows SQL Server generated index and comment:

***Listing 3.18 – Suggested Index***

/\*

Missing Index Details from chapter 03 - TSQL code - new - 10-06-2022.sql - DESKTOP-CEBK38L\GRUMPY2019I1.APSales (DESKTOP-CEBK38L\Angelo (52))

The Query Processor estimates that implementing the following index could improve the query cost by 87.5919%.

\*/

/\*

USE [APSales]

GO

CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]

ON [SalesReports].[YearlySalesReport] ([StoreNo])

INCLUDE ([StoreName],[StoreTerritory],[CalendarDate],[TotalSalesAmount])

GO

\*/

SQL Server tells us that if we create this index,after we give it a name it will improve performance by 87.5%.

OK, sounds good but we will see.

Let’s call this index ieStoreTerritoryDateTotalSales. A little copy and paste action to include the command with the name results in the following DDL query:

CREATE NONCLUSTERED INDEX ieStoreTerritoryDateTotalSales

ON [SalesReports].[YearlySalesReport] ([StoreNo])

INCLUDE ([StoreName],[StoreTerritory],[CalendarDate],[TotalSalesAmount])

GO

We execute it and then generate a new estimated query plan.

Please refer to figure 3.20 below:

Diagram, schematic

Description automatically generated

***Figure 3.20 – Estimated Query Plan after index is created***

Right off the bat we see an improvement. Looks like the new index is being used. The index seek tasks chimes in at 51% cost. We do have a hash match aggregate at a cost of 31%. The sort is now at 4% cost and the nested loop join is at 2%.

Below are the statistics once the query is executed with the new index.

Please refer to table 3.7 below:

***Table 3.7 – Statistics after Suggested Index is Created***

| **SQL Server Parse & Compile Time** | **Existing Index** | **New Index** |
| --- | --- | --- |
| CPU Time (ms) | 0 | 31 |
| elapsed time (ms) | 98 | 113 |
|  |  |  |
| **Statistic (work table)** | **Existing Index** | **New Index** |
| Scan Count | 9 | 9 |
| Logical Read | 171 | 171 |
| Physical Reads | 0 | 0 |
| read-ahead reads | 0 | 0 |
|  |  |  |
| **Statistic (work file)** | **Existing Index** | **New Index** |
| Scan Count | 0 | 0 |
| Logical Read | 0 | 0 |
| Physical Reads | 0 | 0 |
| read-ahead reads | 0 | 0 |
|  |  |  |
| **Statistic (YearlySalesReport)** | **Existing Index** | **New Index** |
| Scan Count | 1 | 1 |
| Logical Read | **4033** | **343** |
| Physical Reads | 1 | 0 |
| read-ahead reads | **4041** | **332** |
|  |  |  |
| **SQL Server Execution Times** | **Existing Index** | **New Index** |
| CPU Time (ms) | **47** | **16** |
| elapsed time (ms) | **1025** | **159** |

For the parse step the CPU time went up to 31ms and the elapsed time went up to 113ms. We are going in the wrong direction!

No change for the work table and work file but if we look at statistics for processing the YearlySalesReport table we see that:

* The scan count remained the same – no news is good news.
* Logical reads went from 4033 to 343 – a major improvement.
* Physical reads went from 1 to 0 – we will take it.
* Read-ahead reads went from 4041 to 332 – another major improvement.

Finally, for the SQL execution times the CPU time went from 47 to 16 and the elapsed time went from 1025 to 159. Creating the suggested index helped a lot.

Will a denormalized report table that replaces the CTE help even more? Or what about a memory optimized table?

**Using a Report Table**

Sometimes creating one or more indexes to improve performance for queries that utilize window functions or complex calculations, or both are just not enough. The next step to consider is creating what is called a denormalized table or a report table that is loaded at off hours, assuming the users can tolerate data that is 24 hours old.

Key to this step is understanding business analyst requirements. Some users will want or need current, live data while other users can accept aged data, like when they need to run month end totals or some other type of report that summarizes past performance.

As a SQL developer or architect, you need to work with your business counterpart and collect the data and performance reports the business users expect. Hopefully, after your requirements gathering sessions, you will arrive at the scenario where 80% of users do not need live data, instead day old data or older is sufficient. The other 20% of the users that need live data require complex solutions that might involve hardware, complex replications schemes or querying transaction tables directly, which is not recommended.

Let’s take a look at a possible solution for the query we just examined that uses a report table that is loaded off hours. The CTE portion of the query is replaced with the report table. We will pre-load it and then check out performance on the query that contains our windows functions.

Please refer to listing 3.19 below for the script to load the report table and the revised query:

***Listing 3.19 – YearlySummaryReport Table for Pre-loading***

DROP TABLE IF EXISTS [SalesReports].[YearlySummaryReport]

GO

CREATE TABLE [SalesReports].[YearlySummaryReport](

[SalesYear] [int] NULL,

[SalesMonth] [int] NULL,

[StoreNo] [nvarchar](32) NOT NULL,

[StoreName] [nvarchar](64) NOT NULL,

[StoreTerritory] [nvarchar](64) NOT NULL,

[TotalSales] [decimal](10, 2) NULL

) ON [AP\_SALES\_FG]

GO

INSERT INTO APSales.SalesReports.YearlySummaryReport

SELECT YEAR(CalendarDate) AS SalesYear

,MONTH(CalendarDate) AS SalesMonth

,StoreNo

,StoreName

,StoreTerritory

,SUM(TotalSalesAmount) AS TotalSales

FROM APSales.SalesReports.YearlySalesReport

GROUP BY YEAR(CalendarDate)

,MONTH(CalendarDate)

,StoreNo

,StoreName

,StoreTerritory

GO

CREATE NONCLUSTERED INDEX [ieYearlySalesStoreTerritorySummary]

ON [SalesReports].[YearlySummaryReport] ([StoreNo],[SalesYear])

INCLUDE ([SalesMonth],[StoreName],[StoreTerritory],[TotalSales])

GO

This script uses the output columns defined for the CTE and uses them to create a physical table called YearlySummeryReport. Once the table is created the query that was originally used for the CTE is used in an INSERT statement to load the table. This step was easy. Now for the query that contains the windows functions and the analysis we need to perform to see if we have an improvement on the estimated query plan and the IO and TIME statistics.

Please refer to listing 3.20 below:

***Listing 3.20 –Querying the Report Table***

DBCC dropcleanbuffers

CHECKPOINT;

GO

SET STATISTICS TIME ON

GO

SET STATISTICS IO ON

GO

SELECT SalesYear

,SalesMonth

,StoreNo

,StoreName

,StoreTerritory

,FORMAT(TotalSales,'C') AS TotalSales

,FORMAT(PERCENTILE\_CONT(.5)

WITHIN GROUP (ORDER BY TotalSales)

OVER (

PARTITION BY SalesYear

),'C') AS PctCont

,FORMAT(PERCENTILE\_DISC(.5)

WITHIN GROUP (ORDER BY TotalSales)

OVER (

PARTITION BY SalesYear

),'C') AS PctDisc

FROM APSales.SalesReports.YearlySummaryReport

WHERE SalesYear IN(2010,2011)

AND StoreNo = 'S00004

GO

SET STATISTICS TIME OFF

GO

SET STATISTICS IO OFF

GO

We have covered this query earlier so let’s get right to it and generate the estimated query plan in the usual manner. I did create an index on the new report table. This index was suggested by the estimated query plan tool. Please refer to the listing 3.21 below:

***Listing 3.21 – Index to Support Report Table.***

CREATE NONCLUSTERED INDEX [ieYearlySalesStoreTerritorySummary]

ON [SalesReports].[YearlySummaryReport] ([StoreNo],[SalesYear])

INCLUDE ([SalesMonth],[StoreName],[StoreTerritory],[TotalSales])

GO

We need to compare the new plan to the prior plan to see what improvements, if any, have occurred.

Please refer to figure 3.21 for the estimated query plans for both approaches:

Graphical user interface, application

Description automatically generated

***Figure 3.21 – Comparing Estimated Query Plans***

The top estimated query plan represents the original query with the CTE. The bottom query plan is for the revised script that queries the reporting table that was preloaded.

What we see is that the index seek for the reporting table query is at a cost of 20% compared to the index seek of 51% for the original query. The new query also has a sort step significantly higher than the sort step in the original query that uses the CTE. Seems like when index seek cost goes down the sort cost goes up. Something to keep an eye out for.

Is this an improvement? Not sure so let’s look at the IO and TIME statistics to see what they tell us.

Please refer to table 3.8 below:

***Table 3.7 – Comparing IO and TIME Statistics***

| **SQL Server Parse & Compile Time** | **Existing Index** | **New Index** |
| --- | --- | --- |
| CPU Time (ms) | 15 | 0 |
| elapsed time (ms) | **20** | **58** |
|  |  |  |
| **Statistic (work table)** | **Existing Index** | **New Index** |
| Scan Count | 9 | 9 |
| Logical Read | 171 | 171 |
| Physical Reads | 0 | 0 |
| read-ahead reads | 0 | 0 |
|  |  |  |
| **Statistic (work file)** | **Existing Index** | **New Index** |
| Scan Count | 0 | NA |
| Logical Read | 0 | NA |
| Physical Reads | 0 | NA |
| read-ahead reads | 0 | NA |
|  |  |  |
| **Statistic (YearlySalesReport) vs YearlySummaryReport** | **Existing Index** | **New Index** |
| Scan Count | **1** | **2** |
| Logical Read | **343** | **6** |
| Physical Reads | **2** | **1** |
| read-ahead reads | **345** | **8** |
|  |  |  |
| **SQL Server Execution Times** | **Existing Index** | **New Index** |
| CPU Time (ms) | **16** | **0** |
| elapsed time (ms) | **132** | **67** |

Looking at the statistics that were generated we see significant improvements when we use the reporting table called YearlySummaryReport. The scan count went up from 1 to 2 but the logical reads went from 343 to 6. Physical reads decreased by 1 in the new reporting table and best of all, read-ahead reads went down from 345 to 8. So yes, there seems to be a significant improvement when using preloaded and precalculated reporting tables in queries that use window functions. Come to think of it, eliminating the CTE or load step from the analysis definitely will improve performance of the query.

Last but not least, CPU time for Execution times went down from 16 to 0 and elapsed time went from 132 to 67 milliseconds.

One last comment when performing this type of analysis, always run several tests, clearing the buffers with DBCC for each trial. Record the results in a spreadsheet for later analysis and evaluation as to which performance improvement modification works best.

By the way, isolating the loading of a report table does not mean we completely ignore performance analysis on the load steps. If we are loading millions of rows nightly, then we need to come up with a load strategy that will not take so long it creeps into work hours!

**Summary**

We just covered our second set of analytical functions. I think some interesting discoveries were made in not only analytical techniques to use to make our business analysts happy but also in performance tuning.

We did the usual analysis with the estimated query plans, IO and TIME statistics and we also looked at client statistics.

Next, we introduced denormalized staging or reporting tables in order to simulate a production environment that loads data over night so as to perform all calculations and joins. We performed the same performance analysis as we did for the CTE based queries.

Finally, we introduced memory optimized tables which are loaded into memory. We discovered that these could be very fast.

In conclusion, you now know how to use the analytical functions within partitions that use the OVER() clause. In the next chapter we will take a look at the ranking functions and play around with window frame definitions to make things interesting.

One more comment, we are only touching on performance analysis and tuning techniques. The scope of this book is on the window functions but a bare minimum of learning how to use some of the performance tuning tools and techniques is warranted as it will help you write fast and effective scripts and queries.

For a great read on performance tuning please refer to:

SQL Server 2022 Query Performance Tuning

Troubleshoot and Optimize Query Performance

By Grant Fritchey

Sixth Edition, APRESS

A great read and in my opinion a must in your library if you are serious about learning performance analysis, associated tools and performance improvement.