**Chapter 4 – Sales Use Case – Ranking/Window Functions**

This is our last chapter dealing with the sales data warehouse. We will look at the third category of functions called window or ranking functions. We will take the same approach as prior chapters, that is provide a brief explanation of the function, present the code and the query results, and do some performance analysis and tuning by adding indexes and report tables. Lastly, we will present a data analysis problem called gaps and islands where we use some of the window system functions we studied in chapter 3 to identify gaps and sequences in dates for sales data.

**Ranking/Window Functions**

There are only 4 functions in this category but we will revisit the PERCENT\_RANK() function from the last chapter as it has a lot of similarities with some of these functions:

* RANK()
* PERCENT\_RANK()
* DENSE\_RANK()
* NTILE()
* ROW\_NUMBER()

The RANK() function shows the rank of the value in each row column of a data set relative to the other row values in the data set. In case of ties, the rank is the same for the ties but the next value after the duplicate value skips the next value and a number equal to the number of ties plus the current rank value is assigned.

So, if you have a 3-way tie for a value like 4, and it is assigned a rank of 4, the next rank for a value greater than 4 will be 4 plus the number of ties (3) = 7 (yikes!).

The DENSE\_RANK() function works almost the same as the RANK() function. In case of ties, the rank is the same for the ties but the next value after the duplicate values simply assigns the next rank number. So, if the rank for the ties is 4, the next rank assigned to the next higher value is 5 (makes sense to me …).

The PERCENTAGE\_RANK() function assigns a ranking as a percentage. Ties get the same rank percentage values. 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 data type) by 100.00 or use the FORMAT()function to convert the results to a percentage.

The ROW\_NUMBER() function simply assigns the next highest number to the row regardless of ties.

So, if the current row is the fourth row in the data set, the row number 4 will be assigned. For row number 5 then, wait for it, the number 5 is assigned and so on. This function does not care if there are ties in the column that contains values. It only cares about the position of the row within the data set.

The NTILE() function allows you to divide a set of rows in a data set into tiles or buckets. If you have a data set of 12 rows and you want to assign 4 tiles, each tile will have 3 rows. This function comes in handy when you want to evaluate salesperson performance and grant them bonuses based on sales performance. (We will see how to do this. We will generate several tiles to implement performance buckets and assign a salesperson to them.)

OK, I am sure that all this was a bit confusing, and you are scratching your head. Let’s use the simple example from the prior chapter, modify it a bit and then look at some simple data so we can explain what each function does (except for NTILE()) and how they are alike (and differ from) each other.

Please refer to listing 4.1 below:

***Listing 4.1 – Ranking Functions 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),

('FOUR',4),('SIX',6),('SEVEN',7),

('EIGHT',8),('NINE',9),('TEN',10);

SELECT

TestKey,

TheValue,

ROW\_NUMBER() OVER(ORDER BY TheValue) AS RowNo,

RANK() OVER(ORDER BY TheValue) AS ValueRank,

DENSE\_RANK() OVER(ORDER BY TheValue) AS DenseRank,

PERCENT\_RANK() OVER(ORDER BY TheValue) AS ValueRank,

FORMAT(PERCENT\_RANK() OVER(ORDER BY TheValue),'P') AS ValueRankAsPct

FROM @ExampleValues;

GO

We declare our simple table variable and insert ten rows into it. Notice the duplicate values of (‘FOUR’,4) in the INSERT statement. This was done on purpose, so we will see how duplicate values affect the behavior of each function.

Each function is used with an OVER()clause. We include only an ORDER BY clause that sorts the partition data set by the values in the “TheValue” column. Notice the use of the FORMAT() function to display the results of the PERCENT\_RANK() function as a percentage. Let’s see the results.

**Note:** in this example the partition is the entire data set, as no PARTITION clause was included. After all, we only have 10 rows.

Please refer to figure 4.1 below:

Graphical user interface, table

Description automatically generated

***Figure 4.1 – Ranking Functions in Action Results***

Let’s start with the ROW\_NUMBER() function results. There are 10 rows, and the row numbers are 1 thru 10 in sequential order. It does just what the name implies.

Next is the rank function results. Notice what happens with the two ties represented by the value 4. Each is assigned the rank 4 because they are the same. But look at what happens to the next row value. It skips the number 5 and assigns a value of 6. Just like the formula we discussed earlier, it takes the number of ties (2) and adds it to the current rank 4 to assign a rank of 6.

Now look at the DENSE\_RANK() function result. Just like the rank function, ties are assigned the same rank value but the next row value is assigned the next higher rank which is 5.

Last but not least, the PERCENT\_RANK() function assigns a rank as a percentage between 0 and 1. I added a column so as to use the FORMAT() function to format the results as a percentage. This way we can compare the formats and decide which one makes sense to use.

**NTILE() Example**

Let’s create some tiles. No, not kitchen floor tiles but tiles that are buckets of data (I know, bad pun). We will start off with another simple example so we can clearly see what this function does. We will use a table variable loaded with 10 rows of year-to-date sales amounts for our sales team.

What we want to do is create three tiles or buckets of data so we can use them as criteria to award bonuses to the sales team members according to the tile they fall into.

Please refer to listing 4.2 below:

***Listing 4.2 – Assigning Performance Buckets for Bonuses***

DECLARE @SalesPersonBonusStructure TABLE (

SalesPersonNo VARCHAR(4) NOT NULL,

SalesYtd MONEY NOT NULL

);

INSERT INTO @SalesPersonBonusStructure VALUES

('S001',2500.00),

('S002',2250.00),

('S003',2000.00),

('S004',1950.00),

('S005',1800.00),

('S006',1750.00),

('S007',1700.00),

('S008',1500.00),

('S009',1250.00),

('S010',1000.00);

-- Care must be taken how you sort (ASC or DESC)

SELECT SalesPersonNo

,SalesYtd

,NTILE(3) OVER(ORDER BY SalesYtd DESC) AS BonusBucket

,CASE

WHEN (NTILE(3) OVER(ORDER BY SalesYtd DESC)) = 1

THEN 'Award $500.00 Bonus'

WHEN (NTILE(3) OVER(ORDER BY SalesYtd DESC)) = 2

THEN 'Award $250.00 Bonus'

WHEN (NTILE(3) OVER(ORDER BY SalesYtd DESC)) = 3

THEN 'Award $150.00 Bonus'

END AS BonusAward

FROM @SalesPersonBonusStructure

GO

The solution is simple. The NTILE() function is used with an OVER() clause to create the 3 tiles. There is no PARTITION BY clause due to the small data set but we include an ORDER BY clause so we can sort the year-to-date sales amounts in descending order.

Next, a series of 3 CASE blocks determine the tile assignment by using the NTILE() function again so as to print out a message of the amount the salesperson is awarded based on the tile he or she is in.

Maybe a little brute force as we could have used a CTE to determine the bucket and then write the query so it just tests the tile value instead of using the NTILE() function again. See if you can download the script for this chapter and code a query using the CTE approach. I did provide the solution just in case you get stuck in the script for this section.

Let’s see what kind of awards each salesperson received. Please refer to figure 4.2 below:

Graphical user interface, table, Excel

Description automatically generated

***Figure 4.2 – Bonus Performance Buckets***

There are 10 rows in the result set, and we specified only 3 buckets, so the bucket assignment is uneven. The first two buckets get three rows each and the last bucket get’s 4 rows. If we had 12 rows, then each bucket would get 4 rows. Either case these are rather cheesy bonuses. Our sales staff needs to sell more if they want larger bonuses!

Armed with our understanding of how these functions work, let’s use the functions against our sales data warehouse. We will start with RANK() and PERCENT\_RANK() and revisit NTILE() at the end of the chapter.

**RANK() versus PERCENT\_RANK()**

Let’s see how these two functions work and compare the results to see any similarities. The PERCENT\_RANK() falls under the analytical functions category which we covered in the last chapter but I include it in this chapter so we can compare it to the RANK() function.

We will use our usual structure to put these functions through their paces. Please refer to Listing 4.3 below:

***Listing 4.3 – Rank versus Percent Rank***

WITH CustomerRanking (

CalendarYear,CalendarMonth,CustomerFullName,TotalSales

)

AS

(

SELECT CalendarYear

,CalendarMonth

,CustomerFullName

,SUM(TotalSalesAmount) AS TotalSales

FROM SalesReports.YearlySalesReport YSR

JOIN DimTable.Calendar C

ON YSR.CalendarDate = C.CalendarDate

GROUP BY C.CalendarYear

,C.CalendarMonth

,CustomerFullName

)

SELECT

CalendarYear

,CalendarMonth

,CustomerFullName

,FORMAT(TotalSales,'C') AS TotalSales

,RANK()

OVER (

-- PARTITION BY CalendarYear

ORDER BY TotalSales

) AS Rank

,PERCENT\_RANK()

OVER (

-- PARTITION BY CalendarYear

ORDER BY TotalSales

) AS PctRank

FROM CustomerRanking

WHERE CalendarYear = 2011

AND CalendarMonth = 1

ORDER BY

RANK() OVER (

PARTITION BY CalendarYear

ORDER BY TotalSales

) DESC

GO

Our CTE simply assembles some columns we will need to report on and also includes the SUM() function to calculate total sales by year, month and customer name. To limit the number of rows returned a WHERE clause is used to filter the results by year and for one month only (2011, January).

Reminder – start with small data sets. Develop the solution and test it. Once you are satisfied it works correctly, apply the solution to larger data sets.

Both the RANK() and PERCENT\_RANK() function use an OVER() clause that includes an ORDER BY clause so the partition rows are sorted by the TotalSales column. The PARTITION BY clause is commented out as we are only retrieving 1 years’ worth of data. Once you are comfortable with the logic feel free to retrieve multiple rows and uncomment the PARTITION BY clause so we can process multiple years.

Lastly, here is something interesting. You can include a window function in the query ORDER BY clause (as opposed to the OVER() clause). We do so with the same code used for the RANK() function. Leaving it out will give you the same results but in reverse order. But at least you know it can be done. Just understand the differences. When the ORDER BY clause is in the OVER() clause it sorts the rows in the partition. When the ORDER BY clause is at the end of the query it sorts the final results of the query.

Please refer to the partial results in Figure 4.3 below:

Graphical user interface, application, table, Excel

Description automatically generated

***Figure 4.3 – Rank versus Percent Rank***

Results are in descending order. Notice the behavior of the duplicates in rows 6 and 7 and rows 21 and 22. As mentioned earlier, multiply percent rank by 100.00 so you see values as percentages or use the FORMAT() function instead. This way you can graph the results in Microsoft Excel as shown below.

I copied and pasted the results and multiplied the percent rank results by 100.00 so we see a nice graph instead of small values less than or equal to 1.0.

Please refer to figure 4.4 below:

Graphical user interface, application, table, Excel

Description automatically generated

***Figure 4.4 – Rank versus Percent Rank Analysis***

Not too bad, we do see the graph results are almost the same for both functions with a nice upwards trend. I should have included a comment stating that the percentage results were multiplied by 100.00 to make readers of the graph aware of the manipulation.

The data is displayed to the left for easy reference. It is always a good idea to generate some nice graphs and charts with Microsoft Excel for your users so they can interpret the results. A picture is worth a thousand words! Microsoft Power BI is also a great tool for visualizing data.

Let’s see the performance characteristics for this query.

**Tip:** as a SQL Developer or Architect you want to also master some other skills, like having a good working knowledge of Microsoft Excel or Power BI so you can generate powerful visuals for your users and management. Some ETL skills with SSIS (SQL Server Integration Services) would also be valuable.

**Performance Considerations**

Let’s start off by generating an estimated query plan in the usual manner (Menu Bar -> Query -> Display Estimated Execution Plan). You will see some rather interesting and puzzling results.

Please refer to Figure 4.5 below:

Graphical user interface

Description automatically generated

***Figure 4.5 – Estimated Query Plan – Rank versus Percent Rank***

We had some indexes laying around, so the estimator did not suggest any but wait a minute, the tasks add up to more than 100%:

* Table Scan – 2%
* Index Scan – 50%
* Index Seek – 90%
* Hash Match – 26%

These values add up to 168%

This is not possible. Costs cannot be over 100%. Why is this?

You will not like the answer.

This is a bug with the client-side software and occasionally you will get surprising results like this. I recommend you research this on the Microsoft web site to see if newer versions of SSMS will clear this up. Fortunately, it looks like it does not happen often.

Back to the query plan, as you can see some indexes were laying around from the prior chapter, so they were used. This is what you want, you want your indexes to be used by multiple queries and not have to create a new index every time a new request comes in.

Let’s examine the IO and TIME statistics generated by this query.

Please refer to table 4.1 below:

***Table 4.1 – Query IO & Time Statistics***

| **SQL Server Parse & Compile Time** | **Existing Index** |
| --- | --- |
| CPU Time (ms) | 0 |
| elapsed time (ms) | 127 |
|  |  |
| **Statistic (work table 1)** | **Existing Index** |
| Scan Count | 0 |
| Logical Read | 0 |
| Physical Reads | 0 |
| read-ahead reads | 0 |
|  |  |
| **Statistic (work table 2)** | **Existing Index** |
| Scan Count | 0 |
| Logical Read | 0 |
| Physical Reads | 0 |
| read-ahead reads | 0 |
|  |  |
| **Statistic (YearlySalesReport)** | **Existing Index** |
| Scan Count | 1 |
| Logical Read | **1384** |
| Physical Reads | 1 |
| read-ahead reads | **1401** |
|  |  |
| **Statistic (Calendar)** | **Existing Index** |
| Scan Count | 1 |
| Logical Read | 48 |
| Physical Reads | 0 |
| read-ahead reads | 0 |
|  |  |
| **SQL Server Execution Times** | **Existing Index** |
| CPU Time (ms) | 125 |
| elapsed time (ms) | 334 |

Seems like we have issues with logical reads and read-ahead reads. These exist on the CTE side with the YearlySalesReport table. Logical reads might not be an issue though as they are performed in memory (If you have enough that is!). Stay tuned, we need to look at IO and TIME statistics to see if issues exist related to logical reads.

The query in the CTE might be a candidate for a denormalized report or staging table or better yet, a memory enhanced table. The indexes helped but there is just so much you can do by creating indexes. The more indexes you add the more performance will degenerate when you are loading, modifying, or deleting rows in this table.

Based on the code examples from the last chapter, see if you can modify this table by replacing the CTE with a script that loads a report or memory enhanced table and then try this analysis on your own.

**RANK() versus DENSE\_RANK()**

Let’s look at the RANK()function again but now we will compare it to the DENSE\_RANK() function.

As a reminder, so you do not have to flip back to the start of the chapter, the DENSE\_RANK() function works almost the same as the RANK() function. In case of ties, the rank is the same for the ties but the next value after the duplicate values simply assigns the next rank number.

Please refer to Listing 4.4 below:

***Listing 4.4 – Rank versus Dense Rank***

WITH CustomerRanking (

CalendarYear,CalendarMonth,CustomerFullName,TotalSales

)

AS

(

SELECT YEAR(CalendarDate)

,MONTH(CalendarDate)

,CustomerFullName

-- add one duplicate value on the fly

,CASE

WHEN CustomerFullName = 'Jim OConnel' THEN 17018.75

ELSE SUM(TotalSalesAmount)

END AS TotalSales

FROM SalesReports.YearlySalesReport

GROUP BY YEAR(CalendarDate)

,MONTH(CalendarDate)

,CustomerFullName

)

SELECT

CalendarYear

,CalendarMonth

,CustomerFullName

,FORMAT(TotalSales,'C') AS TotalSales

,RANK()

OVER (

ORDER BY TotalSales

) AS Rank

,DENSE\_RANK()

OVER (

ORDER BY TotalSales

) AS DenseRank

FROM CustomerRanking

WHERE CalendarYear = 2011

AND CalendarMonth = 1

ORDER BY

DENSE\_RANK() OVER (

PARTITION BY CalendarYear

ORDER BY TotalSales

) DESC

GO

I include a CASE statement in the CTE to hard code a duplicate value, so we see the results. This is just a trick to use when prototyping. You can always introduce the duplicate in your data if it is test data. Never mess with production data. If you do, I hope your resume is up to date!

Basically, this is a carbon copy of the prior query but I included the DENSE\_RANK() function so we can compare results. We did say that these functions are similar. The numerical results together with the charts in Microsoft Excel will verify this statement.

Once again, to keep the result set small, the query uses a WHERE clause filter to pull data only for one year, 2011 and for the month of January. Let’s see the query results.

Please refer to Figure 4.6 below:

Graphical user interface, application, table, Excel

Description automatically generated

***Figure 4.6 – Rank versus Dense Rank Analysis***

What we want to look at here is how the results differ between these functions if duplicate values are present. Starting from the bottom with rows 63 and 64, both sales totals are $99.00 so this is where the fun begins. Both get a rank and dense rank of 2, but look at what happens when we move to row 62. The DENSE\_RANK() value starts at the next value 3 while the rank skips a number and goes to 4. Value 3 is ignored.

When we look at rows 54 and 53, we have another set of duplicates, and the pattern repeats itself. Which function to use depends on how you want to treat duplicate values.

One last point, notice with DENSE\_RANK() there are no gaps in the rankings.

Here is our chart generated with Microsoft Excel. Please refer to the figure 4.7 below:

Graphical user interface, chart, application, Excel

Description automatically generated

***Figure 4.7 – Graphing Rank versus Dense Rank***

Sure enough, the results are similar once again. A complete set of data and graphs like these for your users will guarantee productive analysis and decision making.

Generating these graphs is easy by the way. Copy and past the results to a spreadsheet. Tighten up the data and then highlight it so you can generate the graph. Click on “Insert” in the menu bar and then select “Recommended Charts”. Pick a chart and you’re done except for some formatting to make it look professional.

**Performance Considerations**

I think there will be no surprises here due to all the indexes we created in this chapter and prior chapters. Let’s create an estimated query plan in the usual manner (click on query in the menu bar and then click on “Display Estimated Execution Plan” in case you forgot).

Please refer to figure 4.8 below:

Graphical user interface, application, Word

Description automatically generated

***Figure 4.8 – Estimated Execution plan for Dense Rank.***

Once again, an existing index results in an index scan task with a cost of 82%. If we add up all the costs, we get 82% + 1% + 15% = 98%. Not shown to the left are two sort tasks at 1% each so the total cost adds up to 100% here. No bug to contend with this time. Let’s look at IO and TIME statistics next.

Please refer to table 4.2 below:

***Table 4.2 – Query IO & Time Statistics***

| **SQL Server Parse & Compile Time** | **Existing Index** |
| --- | --- |
| **CPU Time (ms)** | **4** |
| **elapsed time (ms)** | **4** |
|  |  |
| **Statistic (workfile)** | **Existing Index** |
| **Scan Count** | **0** |
| **Logical Read** | **0** |
| **Physical Reads** | **0** |
| **read-ahead reads** | **0** |
|  |  |
| **Statistic (work table)** | **Existing Index** |
| **Scan Count** | **0** |
| **Logical Read** | **0** |
| **Physical Reads** | **0** |
| **read-ahead reads** | **0** |
|  |  |
| **Statistic (YearlySalesReport)** | **Existing Index** |
| **Scan Count** | **1** |
| **Logical Read** | **1384** |
| **Physical Reads** | **1** |
| **read-ahead reads** | **1401** |
|  |  |
| **SQL Server Execution Times** | **Existing Index** |
| **CPU Time (ms)** | **62** |
| **elapsed time (ms)** | **58** |

Statistics on the work file and the work table are all good. Logical reads seem high but consider that logical reads usually go against memory cache so most likely this value is nothing to worry about. There is 1 physical read which means SQL Server had to go to the physical disk to retrieve the pages it required. The read-ahead reads statistics indicate how many pages anticipated were required and loaded them into memory (recall that if you have a lot of memory this is usually not a problem). Just be on the lookout for table spool tasks which means TEMPDB is used.

Lastly, the scan count of 1 and the physical read of 1 are low values that we shouldn’t worry about. Don’t forget to run DBCC to clear cache and to set statistics on and off each time you perform some analysis, so you do not get incorrect results. Turn statistics on at the end, so they are only collected for the query and not for when you run an estimated execution plan. In other words, right before you execute the query.

**NTILE() Function Revisited**

We covered this function in the beginning of this chapter but let’s take another look. Here is a little credit rating and payment delinquency example that places customers into buckets so they can be assigned to collection agents based on how many days they are delinquent in their payments. Code is provided in the script folder for this chapter to load the credit related tables.

Please refer to Listing 4.5 below:

***Listing 4.5 – Assigning Credit Analysts to Delinquent Accounts***

DECLARE @NumTiles INT;

SELECT @NumTiles = COUNT(DISTINCT [90DaysLatePaymentCount])

FROM Demographics.CustomerPaymentHistory

WHERE [90DaysLatePaymentCount] > 0;

SELECT CreditYear

,CreditQtr

,CustomerNo

,CustomerFullName

,SUM([90DaysLatePaymentCount]) AS Total90DayDelinquent

,NTILE(@NumTiles) OVER (

PARTITION BY CreditYear,CreditQtr

ORDER BY CreditQtr

) AS CreditAnaystBucket

,CASE NTILE(@NumTiles) OVER (

PARTITION BY CreditYear,CreditQtr

ORDER BY CreditQtr

)

WHEN 1 THEN 'Assign to Collection Analyst 1'

WHEN 2 THEN 'Assign to Collection Analyst 2'

WHEN 3 THEN 'Assign to Collection Analyst 3'

WHEN 4 THEN 'Assign to Collection Analyst 4'

WHEN 5 THEN 'Assign to Collection Analyst 5'

END AS CreditAnalystAssignment

FROM Demographics.CustomerPaymentHistory

WHERE [90DaysLatePaymentCount] > 0

GROUP BY CreditYear

,CreditQtr

,CustomerNo

,CustomerFullName

ORDER BY CreditYear

,CreditQtr

,SUM([90DaysLatePaymentCount]) DESC

GO

We make use of a variable called @NumTiles to count the number of instances of the account being delinquent 90 days. The following query initializes it:

SELECT @NumTiles = COUNT(DISTINCT [90DaysLatePaymentCount])

FROM Demographics.CustomerPaymentHistory

WHERE [90DaysLatePaymentCount] > 0;

The main query pulls out the year, quarter, customer number and customer full name together with the total days the customer is delinquent. Next, the CreditAnalystBucket column is assigned the bucket number the customer falls in by using the NTILE() function and an OVER() clause that is partitioned by year and quarter. The partition rows are sorted by quarter and customer. As we have results loaded for only one year we do not need to partition by year.

Next, a case block is set up to print a message that states:

**“Assign to Collection Analyst N”**

N is a value between 1 and 5. If the credit analyst bucket value is 1 the customer account is assigned to credit analyst 1, if the value is 2 the customer account is assigned to credit analyst 2 and so on. Try to load multiple years’ worth of credit data by modifying the load script for this table. You will need to add a PARTITION BY clause.

Let’s see the results. Please refer to Figure 4.9 below:

Graphical user interface, table, Excel

Description automatically generated

***Figure 4.9 – Credit Analyst Assignments for 90 Day Accounts***

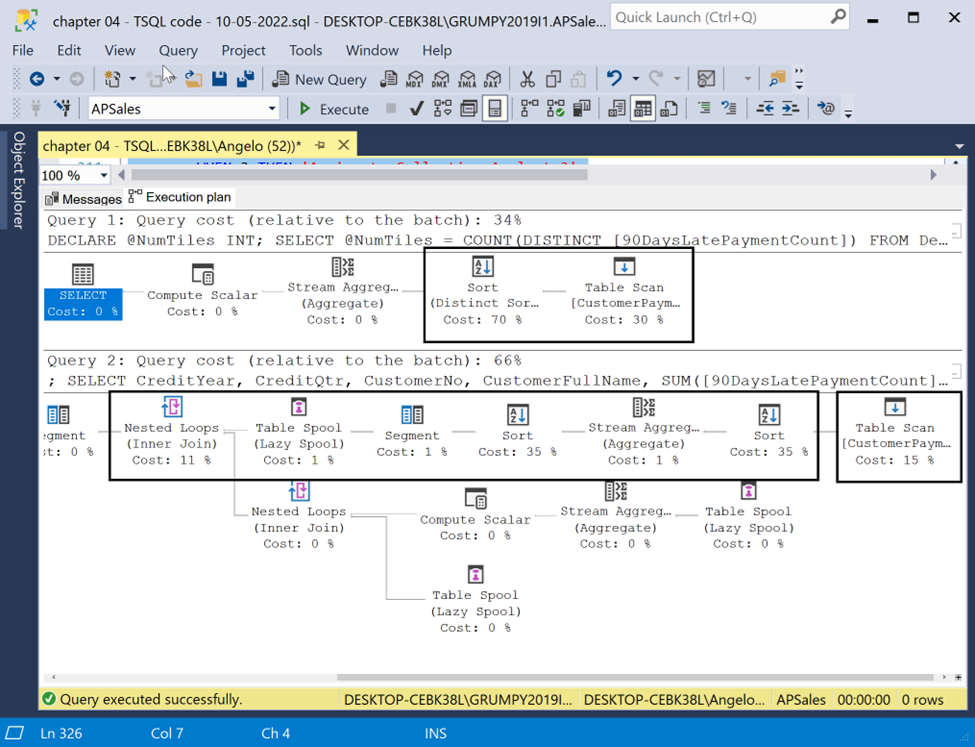
It works well but as can be seen there are some overlaps. Some customers that are 90 days late are assigned to either credit analyst 1 or 2. So this strategy tries to balance out the assignments by volume as opposed to the number of times a customer was 90 days late.

Let’s see how the NTILE() function works as far as performance is concerned.

**Performance Considerations**

Let’s generate an estimated execution plan which was created in the usual manner. (Remember, there are a few ways to create it). This time there are two plans, one for the query that loads the variable and then for the query that uses the NTILE() function.

Please refer to Figure 4.10 below:



***Figure 4.10 – Estimated Execution Plan for NTILE() example.***

Looking at the first execution plan for the query that sets the @NumTiles variable we get the sense that it is expensive. We see (going right to left) the table scan task at a cost of 30% and a very expensive sort step at 70%. The rest of the steps cost 0% so we will not worry about them.

An index was not suggested but I wonder if an index created on the column used in the sort step would help. The answer is no. In this case the table has only 400 rows so indexes would not help or be used. The only alternative is to sort it when it is loaded or pop the table into memory. That is, create it as a memory optimized table.

Let’s look at the second plan. No indexes were suggested, and we see a table scan, two sort tasks and some table spools and two nested loop joins. Nothing to do here.

As I stated earlier, the table has only 400 rows so let’s leave well enough alone. Might as well look at some statistics though for the sake of a complete analysis exercise.

Please refer to table 4.3 below:

***Table 4.3 – Query IO & Time Statistics***

| **SQL Server Parse & Compile Time** | **First Query** | **Second Query** |
| --- | --- | --- |
| **CPU Time (ms)** | **0** | **16** |
| **elapsed time (ms)** | **43** | **41** |
|  |  |  |
| **Statistic (work table)** | **First Query** | **Second Query** |
| **Scan Count** | **0** | **3** |
| **Logical Read** | **0** | **657** |
| **Physical Reads** | **0** | **0** |
| **read-ahead reads** | **0** | **0** |
|  |  |  |
| **Statistic (CustomerPaymentHistory)** | **First Query** | **Second Query** |
| **Scan Count** | **1** | **1** |
| **Logical Read** | **5** | **5** |
| **Physical Reads** | **0** | **0** |
| **read-ahead reads** | **0** | **0** |
|  |  |  |
| **SQL Server Execution Times** | **First Query** | **Second Query** |
| **CPU Time (ms)** | **16** | **0** |
| **elapsed time (ms)** | **41** | **78** |

Also, nothing to worry about here. The second query had 657 logical reads which means it went to memory cache to retrieve data it needed 657 times but that’s not a significant value.

As I stated earlier, when you have a query that works on a small volume of data and the query executes under 1 second, move on to more troublesome queries (choose your battles).

**Note:** at the end of the day, if your query ran under 1 second leave it alone. Ultimately, it is long-running queries that need to be addressed. Also, what are your users willing to tolerate? Times up to 1 minute? Take this into account.

**ROW\_NUMBER() Function**

Let’s look at the ROW\_NUMBER() function again. In this next example we want to keep a running total by month for products that were sold during a two-year time (2011,2012) period for two specific stores. Also, we want to track one product only: Dark Chocolates – Medium size box.

The SUM() aggregate function is used and the ROW\_NUMBER() function is used in a trivial manner to assign an entry number for each month. This function will be used three times so we can see the behavior when we use different PARTITION BY and ORDER BY clauses in the query.

Our last example in this chapter will show you how to use the ROW\_NUMBER() function to solve the island and gaps problem related to data ranges. This will be interesting and practical. Estimated query plans should be interesting also. Let’s see how to calculate rolling totals next.

Please refer to Listing 4.6 below:

***Listing 4.6 – Rolling Sales Total By month***

WITH StoreProductAnalysis

(TransYear,TransMonth,TransQtr,StoreNo,ProductNo,ProductsBought)

AS

(

SELECT

YEAR(CalendarDate) AS TransYear

,MONTH(CalendarDate) AS TransMonth

,DATEPART(qq,CalendarDate) AS TransQtr

,StoreNo

,ProductNo

,SUM(TransactionQuantity) AS ProductsBought

FROM StagingTable.SalesTransaction

GROUP BY YEAR(CalendarDate,

,MONTH(CalendarDate)

,DATEPART(qq,CalendarDate)

,StoreNo

,ProductNo

)

SELECT

spa.TransYear

,spa.TransMonth

,spa.StoreNo

,spa.ProductNo

,p.ProductName

,spa.ProductsBought

,SUM(spa.ProductsBought) OVER(

PARTITION BY spa.StoreNo,spa.TransYear

ORDER BY spa.TransMonth

) AS RunningTotal

,ROW\_NUMBER() OVER(

PARTITION BY spa.StoreNo,spa.TransYear

ORDER BY spa.TransMonth

) AS EntryNoByMonth

,ROW\_NUMBER() OVER(

PARTITION BY spa.StoreNo,spa.TransYear,TransQtr

ORDER BY spa.TransMonth

) AS EntryNoByQtr

,ROW\_NUMBER() OVER(

ORDER BY spa.TransYear,spa.StoreNo

) AS EntryNoByYear

FROM StoreProductAnalysis spa

JOIN DimTable.Product p

ON spa.ProductNo = p.ProductNo

WHERE spa.TransYear IN(2011,2012)

AND spa.StoreNo IN ('S00009','S00010')

AND spa.ProductNo = 'P00000011129'

GO

Starting with the CTE the query generates the sum of transactions by year, month, store, and product. The sum is generated for each month.

The query that uses the CTE needs to generate a rolling total of the sums by month per year. For this purpose the SUM() function is used with an OVER() clause that contains a partition by store number and transaction year and includes an ORDER BY clause by transaction month.

The same OVER() clause is used for the ROW\_NUMBER() function to generate the entry numbers. This is used three times so we can see how the behavior is if we modify the PARTITION BY clause, so it reflects months, quarters, and years.

Let’s check the results. Please refer to Figure 4.11 below:

Graphical user interface, application, table, Excel

Description automatically generated

***Figure 4.11 – Rolling Total Sales by Month***

Report works well enough in that it calculates running totals by Month. Notice where the new year starts or the store changes. The rolling total is reset and so are the entry number levels. We can generate row numbers by month, or quarter or year depending how the PARTITION BY clause is defined:

,ROW\_NUMBER() OVER(

PARTITION BY spa.StoreNo,spa.TransYear

ORDER BY spa.TransMonth

) AS EntryNoByMonth

,ROW\_NUMBER() OVER(

PARTITION BY spa.StoreNo,spa.TransYear,TransQtr

ORDER BY spa.TransMonth

) AS EntryNoByQtr

,ROW\_NUMBER() OVER(

ORDER BY spa.TransYear,spa.StoreNo

) AS EntryNoByYear

This combination of uses of the ROW\_NUMBER()function gives you an indication of how it works. Generally, you can generate row numbers for the entire result set or for the partitions. You cannot include ROW or RANGE clauses though. Makes sense if you think about it. It must work on the entire partition or result set!

Let’s see the performance impact when this function is used.

**Performance Considerations**

This time I will show most of the estimated query plan, it is a long one. I need to split the execution plan into two screen shots. We will start from right to left as usual. The first figure below shows the first (right hand side) half of the plan.

Please refer to Figure 4.12 below:

Graphical user interface, application, table

Description automatically generated with medium confidence

***Figure 4.12 – Estimated Index Plan for Rolling Monthly Sales Analysis***

An index seek appears as the first step with a cost of 16% of the total estimated execution time. Ignoring the low cost plans, a sort step clocks in at 30%. Next, since we are linking (with a JOIN) the SalesTransaction table to the Product table a table scan step appears at a cost of 5%. Not too expensive as this is a small table so no index is needed. A nested loop join task joins both streams of data. Following is a sort step at 22% and finally a Window spool at 1%.

We always want to pay attention to spool tasks as this is an indicator that cached data is spooled so it can be repeatedly accessed. Depending on the type of spool tasks (table, index, lazy, eager), temporary storage is used and then things could get really expensive in terms of execution time (having TEMPDB on a solid-state drive helps …).

Research Microsoft documentation to see which spool tasks are physical (TEMPDB), logical (memory cache) or both. Here is a table with selected tasks that identifies if they are logical or physical or both. Please refer to table 4.4 below:

| **Task** | **Physical** | **logical** |
| --- | --- | --- |
| Sort | X | X |
| Eager Spool |  | X |
| Lazy Spool |  | X |
| Spool |  |  |
| Table Spool |  | X |
| Window Spool | X | X |
| Table Scan | X | X |
| Index Scan | X | X |
| Index Seek | X | X |
| Index Spool |  | X |

***Table 4.4 – Logical and Physical Plan Tasks***

Lastly, physical tasks are tasks that go against physical disk, logical tasks are tasks that go against memory cache. As can be seen in the table above, some tasks can be both depending on the query. Keep this information in mind together with the IO and TIME statistics when you are analyzing performance.

**Note:** I have seen inexpensive solid state USB drives at around $90 for those of you that want to experiment and have some cash laying around. Install SQL Server 2022 and make sure TEMPDB is created on the solid-state drive. Thinking of getting one myself for Christmas!

Let’s look at the left-hand side of the estimated execution plan. Please refer to Figure 4.13 below:

Graphical user interface, application

Description automatically generated

***Figure 4.13 – Left Hand Side of the Plan.***

There’s the Sort step that costs 22% as a point of reference from the prior screenshot. All we see here is the Window spool step we just discussed. This appears because we are using window functions. A 1% cost is a nice low value, so we do not need to worry about it. Or do we?

If logical reads are not zero and the window spool tasks are greater than zero, then the window spool is not using memory.

Here comes our statistics. Please refer to table 4.5 below:

***Table 4.5 – Query IO & Time Statistics***

| SQL Server Parse & Compile Time | Existing Index |
| --- | --- |
| **CPU Time (ms)** | **15** |
| **elapsed time (ms)** | **70** |
|  |  |
| **Statistic (work table)** | **Existing Index** |
| **Scan Count** | **52** |
| **Logical Read** | **289** |
| **Physical Reads** | **0** |
| **read-ahead reads** | **0** |
|  |  |
| **Statistic (SalesTransaction)** | **Existing Index** |
| **Scan Count** | **2** |
| **Logical Read** | **28** |
| **Physical Reads** | **1** |
| **read-ahead reads** | **21** |
|  |  |
| **Statistic (Product)** | **Existing Index** |
| **Scan Count** | **1** |
| **Logical Read** | **1** |
| **Physical Reads** | **1** |
| **read-ahead reads** | **0** |
|  |  |
| **SQL Server Execution Times** | **Existing Index** |
| **CPU Time (ms)** | **0** |
| **elapsed time (ms)** | **44** |

The only high cost is the logical reads on the work table, but remember, logical reads go against cache in memory (if you have enough that is) so this is not a value that should cause concern.

The rest of the values are low, and the estimated query plan analyzer did not suggest a new index, so the query works well. Recall our window spool task. It is not zero so that means it is performing spooling to TEMPDB on physical disk.

All this information is great but what does it all mean? How can we use this information to improve performance?

The rule of thumb is to create indexes based on the columns in the PARTITION BY clause and then the ORDER BY clause. In this prior example we had the following columns in the PARTITION BY clause: StoreNo, TransYear, TransQtr and the ORDER BY clause used the TransMonth column.

OK, but we have a poser for you. The date columns were all generated in the CTE by using the YEAR(), MONTH() and DATEPART() function. These used the CalendarDate column from the SalesTransaction table. These are all derived columns so our index would need to be based on the StoreNo column and the CalendarDate column. Would a JOIN to the Calendar dimension table be more effective so we can pull out these date parts? Would this strategy be more efficient than deriving them with the date functions? Try it out!

Let’s see what our Index Seek task is doing in the estimated query plan. Please refer to figure 4.14 below:

Graphical user interface, text, application

Description automatically generated

***Figure 4.14 – Index Seek Details***

By placing the mouse pointer over any task, a nice yellow popup panel appears with all the details. This is a lot of information and might be hard to read but let me highlight the columns identified in two of the four sections:

**Predicate -** uses column CalendarDatebelonging to the SalesTransaction table**.**

**Seek Predicates -** uses seek keys ProductNo and StoreNo belonging to the SalesTransaction table**.**

So, this indicates that we need to look at not only the columns in the PARTITION BY and ORDER BY clauses of the OVER() clause but also the columns in the WHERE clause, specifically on the WHERE clause PREDICATES also.

The index used in this query is called ieProductStoreSales. It is based on the columns ProductNo, StoreNo and the INCLUDE columns are: CalendarDate and TransactionQuantity.

**Reminder:** INCLUDE columns are columns you include in the CREATE INDEX command by using the INCLUDE keyword. The prefix “ie” in the name stands for inversion entry index. This means that we are presenting the user with a non-unique access path to the data. If you see “pk” it means primary key, “ak” means alternate key which is an alternate to the primary key and “fk” stands for foreign key. I think it is good practice to include these prefixes in the index name.

So now we have taken our performance analysis to a lower level. By looking at the details behind the high-cost tasks we can further see if a strategy is working or not.

So, let’s summarize what we need to base our performance analysis and strategy on:

* Generate estimated and actual query plans and identify high-cost steps.
* Be aware of which tasks are logical or physical operators or both.
* Generate IO and TIME performance statistics.
* Always run DBCC for each test run to make sure the memory cache is clear.
* Base index columns on the columns used in the PARTITION BY clause (in the OVER() clause).
* Base index columns on the columns used in the ORDER BY clause (in the OVER()clause).
* Base index columns on the columns used in the WHERE clause (if one is used in the CTE).
* Base index columns on columns used in derived fields if we use functions like YEAR(), MONTH(), DATEPART(), etc.to pull out the year, month, quarter and day parts of a date.
* Leverage prior indexes by modifying them so they support multiple queries.
* As a last resort consider memory enhanced tables or report tables that are preloaded.
* Solid state drives for TEMPDB will help!

**Denormalization strategies**

The second to last step above said to consider building staging or reporting tables so preprocessing data like data generated by a CTE can be loaded once in a staging table and used by the queries containing the windows functions (assuming users can work with 24-hour data or older).

As part of the above effort, perform an analysis of all your queries to see if you can design a common index set and staging table for multiple queries. Indexes are powerful but too many will slow down the table load and modification processing time to a dangerous level.

Consider building memory optimized tables for fast performance.

If all else fails, we need to consider query or table redesign or even consider hardware upgrades like more memory or more and faster CPUs and storage, like solid state disk.

Finally, test, test, and did I say test your queries? Come up with a simple test strategy and document results for analysis.

**Suggestion:** once you are comfortable reading estimated execution plans and IO and TIME statistics, modify the load scripts that were used to load the databases and try loading a large number of rows, like millions of rows and see how the performance is impacted.

**Island and Gaps Example**

A classic data challenge is the concept of gaps and islands. For example, do we have gaps in the days that sales staff sell products? Or are there clusters of days that a product is selling consistently? This last example shows you how to solve this challenge by using some of the window functions we covered in this chapter.

The solution requires a few steps. The trick to this puzzle is to find out a way to generate some sort of category name we can use in a GROUP BY clause so we can extract the minimum start date and maximum end dates of the islands and gaps using the MIN() and MAX() aggregate function.

We will use the ROW\_NUMBER() function and the LAG() function to identify when a gap or island starts but to also generate a number that will be used in the value that will be used in the GROUP BY clause mentioned above (like ISLAND1, GAP1, etc.).

In our test scenario we are tracking whether a salesperson generated sales over a 31-day period only. Some sales amounts are set to zero for 1 or more days so these gaps in days can be one day, two days or more (these are the gaps).

**Tip:** always start with a small data set. Apply your logic, test it and once you are satisfied it works apply it to your larger production data set.

The same can be said for the groups of days that sales that were generated (these are the islands).

Some steps that need to occur:

Step 1 – Generate some test data. For simplicity include a column that identifies rows as islands or gaps. These will be used to create the groups used in the GROUP BY clause.

Step 2 - Add a numerical column to identify the start days of islands or gaps. This will be used to identify the GROUP BY text value.

Step 3 - Add a column that will contain a unique number so for each set of islands and gaps we

can generate category names for the GROUP BY clause that look something like this:

ISLAND1, ISLAND2, GAP1, GAP2, etc. (we will use the ROW\_NUMBER() function for this).

Once these categories have been correctly assigned to each row the start and stop

days are easily pulled out with the MIN() and MAX() aggregate functions.

Here we go. Let’s start by generating some test data.

Please refer to Listing 4.7a below:

***Listing 4.7a – Loading the SalesPersonLog***

USE TEST

GO

DROP TABLE IF EXISTS SalesPersonLog

GO

CREATE TABLE SalesPersonLog (

SalesPersonId VARCHAR(8),

SalesDate DATE,

SalesAmount DECIMAL(10,2),

IslandGapGroup VARCHAR(8)

);

TRUNCATE TABLE SalesPersonLog

GO

INSERT INTO SalesPersonLog

SELECT 'SP001'

,[CalendarDate]

,UPPER (

CONVERT(INT,CRYPT\_GEN\_RANDOM(1)

)) AS SalesAmount

,'ISLAND'

FROM APSales.[DimTable].[Calendar]

WHERE [CalendarYear] = 2010

AND [CalendarMonth] = 10

GO

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Set up some gaps \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

UPDATE SalesPersonLog

SET SalesAmount = 0,

IslandGapGroup = 'GAP'

WHERE SalesDate BETWEEN '2010-10-5' AND '2010-10-6'

GO

UPDATE SalesPersonLog

SET SalesAmount = 0,

IslandGapGroup = 'GAP'

WHERE SalesDate BETWEEN '2010-10-11' AND '2010-10-16'

GO

UPDATE SalesPersonLog

SET SalesAmount = 0,

IslandGapGroup = 'GAP'

WHERE SalesDate BETWEEN '2010-10-22' AND '2010-10-23'

GO

-- Just in case the random sales value generator

-- set sales to 0 but the update labelled it as an ISLAND

UPDATE SalesPersonLog

SET IslandGapGroup = 'GAP'

WHERE SalesAmount = 0

GO

The gaps and island text strings could have been generated in the query below, but I wanted to simplify things. Notice all the update statements. These set up the test scenario for us.

Please refer to Listing 4.7b below:

***Listing 4.7b – Generating the Gap and Island Report***

SELECT SalesPersonId,GroupName,SUM(SalesAmount) AS TotalSales

,MIN(StartDate) AS StartDate,MAX(StartDate) AS EndDate

,CASE

WHEN SUM(SalesAmount) <> 0 THEN 'Working, finally!'

ELSE 'Goofing off again!'

END AS Reason

FROM (

SELECT SalesPersonId,SalesAmount,

,IslandGapGroup + CONVERT(VARCHAR,(SUM(IslandGapGroupId)

OVER(ORDER BY StartDate) )) AS GroupName

,StartDate

,PreviousSalesDate AS EndDate

FROM

(

SELECT ROW\_NUMBER() OVER(ORDER BY SalesDate) AS RowNumber

,SalesPersonId

,SalesAmount

,IslandGapGroup

,SalesDate AS StartDate

,LAG(SalesDate)

OVER(ORDER BY SalesDate) AS PreviousSalesDate

,CASE

WHEN LAG(SalesDate) OVER(ORDER BY SalesDate) IS NULL

OR

(

LAG(SalesAmount) OVER(ORDER BY SalesDate) <> 0

AND SalesAmount = 0

) THEN ROW\_NUMBER() OVER(ORDER BY SalesDate)

WHEN (LAG(SalesAmount) OVER(ORDER BY SalesDate) = 0

AND SalesAmount <> 0)

THEN ROW\_NUMBER() OVER(ORDER BY SalesDate)

ELSE 0

END AS IslandGapGroupId

FROM SalesPersonLog

) T1

)T2

GROUP BY SalesPersonId,GroupName

ORDER BY StartDate

GO

This query has three levels or three nested queries acting like tables to solve the problem. The in-line query used like a table that is labeled T1 produces the values below:

Please refer to Figure 4.15 below:

Graphical user interface, application, table, Excel

Description automatically generated

***Figure 4.15 – Gaps and Islands Interim Results***

Notice how the values that are used to identify the start of a gap or island are generated by the ROW\_NUMBER() function. This guarantees they are unique. Let’s look at the next level query.

Please refer to Figure 4.16 below:

Table

Description automatically generated

***Figure 4.16 – Generating values for final GROUP BY clause***

At this level we have all the category values generated and we used the LAG() function to set the end date dates. We still have results in liner, sequential dates. All we need to do now is use the MAX() and MIN() functions together with a GROUP BY clause so we pull out the start and end dates for each gap or island.

Here is the code snippet from the original query that does this:

SELECT SalesPersonId,GroupName,SUM(SalesAmount) AS TotalSales,

MIN(StartDate) AS StartDate,MAX(StartDate) AS EndDate,

CASE

WHEN SUM(SalesAmount) <> 0 THEN 'Working, finally!'

ELSE 'Goofing off again!'

END AS Reason

FROM …

Let’s see the results. Please refer to Figure 4.17 below:

Graphical user interface, text

Description automatically generated with medium confidence

***Figure 4.17 – The Gap and Island Report***

Looks good. A column called Reason tells us the cause of gaps in sales days. This salesperson needs a good talking too!

Conclusion, come up with logic to identify the start dates of the islands and gaps. Next, come up with a strategy for generating values that can be used in the GROUP BY clause to categorize the islands and gaps. Finally, use the MAX() and MIN() function to generate the start and stop dates of the gaps and islands.

**Summary**

We covered a lot of ground. What did we learn?

* We learned how to apply the window and ranking functions to our sales data warehouse.
* We took a deeper dive into performance analysis by understanding how data is either cached in memory or physical disks.
* We started to analyze and come up with conclusions on performance based on the nature of estimated plan tasks and IO and TIME statistics.
* We had a brief discussion and summarized some steps to consider when doing performance analysis.
* Lastly, we looked at a real-world data problem called gaps and islands.

Armed with all this knowledge we continue to see how all our functions work against a financial database.