**Chapter 2 – Sales DW Use Case – Aggregate Functions**

Our main goal in this chapter is to learn and apply the category of window functions called aggregate functions. We start off by describing the sales data warehouse called APSales that we will use.

A simple business conceptual model is presented together with some simple data dictionaries to help us understand the business contents of the database.

We include discussions on performance tuning and then proceed to take each window function for a test drive. We conclude the chapter with a brief discussion on the new SQL Server 2022 feature called WINDOW which is used to define the OVER() clause for the window functions.

**Note:** The DDL used to create this small data warehouse is found on the Google web site for this book together with scripts to create all tables, views and to load the tables with test data. The code is easy to use. Each step is labeled and where necessary comments help clarify what each step does.

Let’s spend a little time understanding what our small sales data warehouse is all about.

**Sales Data Warehouse**

The APSales data warehouse stores transaction and historical information for a fictitious company that bakes and sells all sorts of cakes, pies, tarts, and chocolates with a traditional European style.

The company has 100 customers, and we want to track and analyze their purchase patterns over several years’ worth of data. The company sells 60 products that fall under several categories like cakes, pies, tarts, croissants, and chocolates.

Customers purchase their favorite confection at one of 16 stores situated throughout the United States.

Lastly, a staging table is used to simulate purchases by each customer of a few years. Your first assignment is to create the database and tables, load them and write some simple queries to see the actual data you will analyze.

Below is a snapshot of SSMS Object Explorer that shows the database and the tables. Once you create the database and all table objects your own Object Explorer panel should look something like this.

Please refer to figure 2.1 below:

 Graphical user interface, text, application

Description automatically generated

***Figure 2.1 – Data Warehouse Tables***

Notice the tables in the SalesReports and StagingTable schema. The tables assigned to the SalesReports tables represent denormalized views generated by querying the Sales fact table with a star query that joins all dimensions to the fact table so that we can present business analysts with some data that uses all the names, codes, etc.

The tables assigned to the StagingTable schema are used by TSQL batch scripts that generate some random transaction data so we can then load it into the Sales fact table. Let’s check out a simple business conceptual model to understand how all this ties together.

**Sales Data Warehouse Conceptual Model**

Below is a simple conceptual business data model that identifies the main business objects of the Sales Data Warehouse. The target database is a basic star schema model in that it has 1 fact table and several dimension tables (you can have multiple fact tables of course).

A couple of denormalized report tables are also included to facilitate running complex queries.

These tables are called SalesStarReport and YearlySalesReport and are assigned to the SalesReports schema. These are used to tie in the fact and dimension tables and replace the surrogate keys with the actual dimension table columns. They are preloaded each night so that they are available to users the following day. Preloading the tables provides performance benefits by allowing calculations on data to be performed prior to inserting into the table, simpler queries will run faster.

For those not familiar with data warehouse or data marts, columns assigned the role of surrogate keys are used to establish the associations between dimension and fact tables. These are numerical columns as they are faster to use in joins than alpha numeric columns.

Viewed in a diagram, using your imagination they resemble a star and are therefore called star schemas.

The model below is intended to just show the basic business objects and the relationships. It is not a full physical data warehouse model. By the way, I assume you know what a fact table is and what a dimension is.

**Note:** there are many sources on the web that describe what a data warehouse or data mart is. Here is a link that might be helpful if you are not familiar with this architecture:

<https://www.snowflake.com/data-cloud-glossary/data-warehousing/>

Another good resource is: https://www.kimballgroup.com/

Each dimension table has a surrogate key that plays the role of primary key. All these surrogate keys appear in the fact table as foreign keys. Technically, all the foreign keys in the fact table can be used as the primary key for the fact table. The fact table also contains columns that represent data you can count, in other words numerical data. In special cases a fact table can contain other types of columns, but this is a topic for another book. (Look up fact less fact tables and junk dimensions.)

Please refer to figure 2.2 below:

Diagram

Description automatically generated

***Figure 2.2 – The APSales Conceptual Business Model.***

This is called a conceptual business model as it is intended to clearly identify the main business data objects to non-technical users (our friends, the business analysts, and managers). These types of models are used during the initial phases of the database design.

Data modelers, business analysts and other stakeholders are gathered to identify the main data objects that will be in the database or data warehouses. At this stage of the design process only basic metadata is identified, like entity names and their relationships between each other.

Subsequent sessions evolve the conceptual model in a physical denormalized star or snowflake schema design where all the surrogate keys, columns and other objects are identified.

I present this simple model so that the reader can have a basic understanding of what the database that will implement the data warehouse looks like.

Next, we need to create a series of simple documents called data dictionaries that further describe the meta data.

Below is a simple table that shows the associations between the entities that will become the fact table and the dimension tables. You need to understand these so you can build queries that join the tables together.

Please refer to table 2.1 below:

| **Dimension** | **Business Rule** | **Fact/Dimension Table** | **Cardinality** |
| --- | --- | --- | --- |
| Country | Identifies location for a | Sales | One to zero, one or many |
| Store | Fulfills sales for | Sales | One to zero, one or many |
| Customer | Is the subject of | Sales | One to zero, one or many |
| Product | Is subject of | Sales | One to zero, one or many |
| Product Category | Category classification of a product | Product | One to zero, one to many |
| Product Sub Category | Sub Category classification for | Product Category | One, Many to one |
| Calendar | Identifies sales date of | Sales | One to Zero, one to many |

***Table 2.1 – Fact - Dimension Associations***

Typically, these types of tables appear in data dictionaries which contain information about the various data objects that make up a database. If you look at the associations in the table above, we also want to highlight the business rules that link these together:

A country identifies a location in zero, one or more rows in the sales fact table. A fact table row needs to contain a country code, but it is possible to have rows in the country table that have no related row in the sales table. In the database you will build there are only sales transactions for the United States although the Country dimension contains most of the country information for all the ISO standard country codes.

**Note:** a data warehouse is a specialized type of table. In this chapter I use the terms interchangeably.

A store can be the subject of zero, one or more records in the sales fact table. It is possible to have a new store that has not sold any products yet so there are no records in the sales fact table.

The same rule applies for a customer. A customer can appear in zero, one or more records in the sales table. It is possible to have a new customer that has not bought anything yet so the customer will appear in the Customer dimension but not the sales fact table.

A product can appear in zero one or more sales records in the sales fact table. It is possible to have a product not yet sold appear in the product table.

A product category can refer to one or more product subcategories and each product category can refer to one or more products. It is possible to have product categories and product sub categories that have not been assigned to a product yet for various reasons. Like it has not been manufactured yet. (Example, we plan to sell ice creams but have not manufactured them yet).

Last but not least is a simple data dictionary that describes the tables themselves although the names are fairly self-explanatory. Please refer to table 2.2 below:

| **Table Name** | **Description** |
| --- | --- |
| Calendar | Basic calendar date objects used for time reporting. (Like dates, quarter, month and year objects.) |
| Country | Contains ISO 2- and 3-character codes for most of the world’s countries. Example: US or USA. |
| Customer | Basic customer table used to store names of the customers and identifiers. |
| Product | Basic product table that stores names and descriptions of products. |
| Product Category | High level category codes and description for products: chocolate, cakes, pies, croissants, and tarts. |
| Product Subcategory | Further breaks down a product category. For example, chocolates can be broken down into dark chocolates-small, dark chocolates-medium and dark chocolates-large. |
| Store | Table contains basic information for 16 stores, the store number, name, and territory. Useful for sales performance analysis. |
| Sales | Table that stores all sales transactions by customers over multiple years. Who bought what, when and where! |

***Table 2.2 – Table Definition Data Dictionary***

Keep in mind that this is just a basic set of design specifications for the simple data warehouse we will be using. I took great liberties in just presenting the most basic concepts of data warehouse and star schema. You would also need data dictionaries to describe the primary and foreign keys, surrogate keys and also data dictionaries that describe each of the columns, their data types and descriptions in the fact and dimension tables.

To summarize, our goal is to have a simple data warehouse we can load and use as we study the capabilities of all the window functions that are discussed in this book. Hopefully this section gave you a basic understanding of the data warehouse you will build with the script that can be found in the google web site for this book.

**A word about Performance Tuning**

Key to creating queries and batch scripts that utilize the window functions (and the OVER() clause) is performance. We want to create queries that are not only accurate but fast. No user will be happy with a query that runs more than 1 minute.

You will need to develop some skills in performance tuning which entails using tools provided by SQL Server and how to create indexes based on the feedback these tools give you when you analyze the queries you are designing.

There are several tools available with SQL Server and tools available from different vendors. We will concentrate on the SQL Server tools below:

* Query Plans
* Setting STATISTICS IO ON/OFF
* Setting STATISTICS TIME ON/OFF
* Client Statistics
* Creating Indexes (suggested or your own)
* Lots of curiosity and patience
* There is also table denormalization, partitioned tables, physical table file placement, hardware upgrades but we will not cover these.

So, what are query plans?

Visually, query plans are a graphical image that shows the steps SQL Server will take to execute a query. It shows you the flow and the cost of each task in terms of execution time. This tool can be accessed via the menu bar by clicking **Query** and then scrolling down to **Display Estimated Execution plan.**

Please refer to figure 2.3 below:

Graphical user interface, application, table, Excel

Description automatically generated

***Figure 2.3 – The Query Drop Down Menu***

You can alsoinclude the execution plan, live query statistics and client statistics by selecting the options called out in the square box in the diagram above.

Another way of accessing these tools is by clicking on one or more of the buttons in the menu bar that are called out in the figure below by the square boxes.

Please refer to figure 2.4 below:

Graphical user interface, application, table

Description automatically generated

***Figure 2.4 – Query Performance Tools***

The first button will turn on the **Display Estimated Execution Plan** tool. Referring to the next 3 buttons called out by the square box in the figure above you can also generate query plans by clicking the **Include Actual Query Plan** button or generate **IO** statistics by clicking the **Include Live Statistics** button. Lastly you can include client statistics by clicking the last button called **Include Client Statistics**. Using your mouse, move the cursor over each button to generate a message that shows you what the button does.

Let’s look at an example query plan. Please refer to figure 2.5 below:

Timeline

Description automatically generated

***Figure 2.5 – Estimated Query Plans***

The query plan is read right to left. It contains icons called tasks (or operations or plain old steps) that perform specific functions like sort data or merge data or use indexes. Associated with each task is a cost estimate displayed as a percentage of the total time spent to execute a query.

For example, looking at the first task on the right-hand side of the plan in the figure above, we see a table scan that has a cost of 94%. This tells us immediately that we have a problem. This task is taking up most of the execution time.

These types of tasks are the ones that we need to address with some sort of strategy, like adding an index or re-writing the query.

**Note:** sometimes, due to large volumes all that can be done is to run the query overnight, so it loads a denormalized report table. Why do we want to do this? Because this will give the appearance of fast performance to any user accessing the denormalized table. JOINs, calculations, and other complex logic is eliminated when processing the user report. The user is querying a preloaded and calculated table!

Lastly, make sure to run these queries within a reporting tool, like SSRS so you can add filters to limit what the user sees. (Create indexes based on how the users filter the query.) No one wants to scroll through millions of rows of data.

We also see a step that sorts the data, and it has a cost of 6% of the total execution time. Notice that 94% plus 6% = 100%. Usually when you add up all the estimated costs, they will add up to 100%.

Next, we set IO and TIME statistics on with the following commands:

SET STATISTICS IO ON

GO

SET STATISTICS TIME ON

GO

By the way, at the end of the analysis turn off statistics by replacing the keyword ON with OFF:

SET STATISTICS IO OFF

GO

SET STATISTICS TIME OFF

GO

Using these commands and executing the query we are presented with some important performance statistics:

SQL Server parse and compile time:

CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 0 ms.

SQL Server parse and compile time:

CPU time = 15 ms, elapsed time = 70 ms.

(46 rows affected)

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

Table 'SalesTransaction'. Scan count 1, logical reads 718, physical reads 0, page server reads 0, read-ahead reads 718, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.

(6 rows affected)

Some of the interesting values we want to pay attention to are scan counts, logical and physical reads and whether a worktable is used. Notice the logical read count of 718. This needs to go down. Logical reads are performed in memory if you have enough memory available or else on hard disk which really slows things down. Aim for single digit read counts.

The goal right now is to make you aware of these tools and what the output looks like. We will examine what they mean as we develop our TSQL queries that use the window functions and the OVER() clause. For now, just know where to find them, how to turn them on and how to look at the results when you run the query.

**Tip:** you might want to look up the counters above in the Microsoft documentation to begin understanding what they mean. Yes, there are quite a few but understanding them will pay dividend: <https://learn.microsoft.com/en-us/sql/relational-databases/showplan-logical-and-physical-operators-reference?view=sql-server-ver16>

The URL above is fairly long to type, just use your favorites search engine and search on: “Showplan Logical and Physical Operators Reference”

Finally, the **Display Estimated Execution Plan** tool will also suggest a missing index that is needed to improve query performance. Let’s look at an example. Please refer to listing 2.1 below:

***Listing 2.1 – Suggested Index***

/\*

Missing Index Details from chapter 02 - TSQL code - 09-08-2022.sql - DESKTOP-CEBK38L\GRUMPY2019I1.APSales (DESKTOP-CEBK38L\Angelo (63))

The Query Processor estimates that implementing the following index could improve the query cost by 96.4491%.

\*/

/\*

USE [APSales]

GO

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

ON [StagingTable].[SalesTransaction] ([ProductNo])

INCLUDE ([CustomerNo],[StoreNo],[CalendarDate])

GO

\*/

Lots of nice comments and a template for the index is generated. Notice square brackets by the way. When SSMS generates the code for you these are always included. Probably just in case you used names with spaces or weird characters in your database table and columns!

The statement generated tells us a non-clustered index is needed and it is based on the table called SalesTransaction assigned to the StagingTable schema. The columns to include are the ProductNo column and the CustomerNo, StoreNo and CalendarDate column that appear after the INCLUDE keyword.

If we create this index and rerun the **Display Estimated Execution Plan** tool, we should see some improvements.

Please refer to figure 2.6 below:

Text

Description automatically generated with low confidence

***Figure 2.6 – Revised Estimated Query Plans***

Success! The table scan step has been replaced with an Index seek step at a cost of 29%. The sort step is still there with an increased cost of 69%. Is this better or worse?

Let’s execute the query with the statistics settings turned on and see what we get.

Please refer to figure 2.7 below:

Graphical user interface, text, application, email

Description automatically generated

***Figure 2.7 – Statistics IO Report***

Wow! Prior logical reads was 718 and is now reduced to 8. CPU time before was 15ms and now it is reduced to 4ms. Dramatic improvement. Recall that logical reads indicate how many times the memory cache must be read. If enough memory is not available, then physical disk must be read.

This will be expensive when the tables are large, and your disk is slow! Corporate production environments usually implement TEMPDB on a solid-state drive for increased performance.

I almost forgot Client Statistics. Here is the report that is generated when you click on this selection. Please refer to figure 2.8 below:

Graphical user interface, application, table

Description automatically generated

***Figure 2.8 – Client Statistics***

A very comprehensive set of statistics in this report. Notice all the trials. What this means is that the query was run 6 times and we obtained client execution times for each trial run. There is also average execution time for the statistics. Another important set of information is the total execution time and Wait time on Server replies. Trial 6 information seems to be the best, possibly because of the index.

This together with the output of the tools we just examined will allow you to diagnose and identify performance bottlenecks. May sure you practice a bit with some simple queries and see if you can interpret the statistics that are returned.

By the way, I do not want to imply that performance tuning is a walk in the park. This is a difficult discipline to master and takes years of practice to achieve expert level. I am just touching the surface of this very interesting and important activity.

This section was long and as stated earlier, it was meant to introduce you to the tools and look at the output. Lots of information so do not worry if you don’t understand what all the statistics mean yet. What’s important at this stage is that you know where to find them and to try them out on some queries. As we proceed with the chapter, we will do a deep dive into what the important query plan steps tells us and how to use the information to create indexes to deliver better performance.

We will do this for a few queries, but chapter 14 will be dedicated to performance tuning. I will include examples of query plans for the queries we developed in the other chapters so you can get a feeling of performance issues related to window functions and how to solve them.

We are ready to start writing some code!

**Aggregate Functions**

Below are the window aggregate functions we will discuss in this chapter:

* COUNT() & COUNT\_BIG()
* SUM()
* MAX()
* MIN()
* AVG()
* GROUPING()
* STRING\_AGG
* STDEV()
* STDEVP()
* VAR()
* VARP()

The approach we will take is to present the code for each function, discuss it, display the results when it is executed and review the results. We might even generate a few graphs with Excel to better analyze the results.

For the GROUPING and STDEV() function, we generate IO and TIME statistics and an estimated query execution plan by following the steps discussed earlier. Usually, the query plan will also suggest a recommended index to create. If an index is suggested, we will copy the template and modify it by giving it a name and create it.

Once the index is created, we re-generate statistics and the query plan and compare the new set to the prior set of statistics to see what improvements in performance have been gained by creating the new index.

Sometimes creating indexes does not help and can even hurt performance. Not to mention that the more indexes you create the more performance decreases when you need to delete, update, or insert rows.

**COUNT(),MAX(),MIN(),AVG() & SUM() Function**

Let’s start examining the workhorse functions you will use frequently in your queries and reports. Below is a query that reports the count of transactions, the largest and smallest number of transactions and the total quantities of a particular product purchased.

We want to create a crosstab report by year, month, store and of course product to give us a sales performance profile. Let’s not confuse transaction counts versus quantities of items sold. A single transaction could have 10 products or 1 or maybe 20 or more. Note the difference.

Lastly, we want to limit the results by a single store, product, and year so we can analyze the results. Here’s the query. Please refer to listing 2.2 below:

***Listing 2.2 – Basic Sales Profile Report***

SELECT YEAR(CalendarDate) AS PurchaseYear,

MONTH(CalendarDate) AS PurchaseMonth,

StoreNo,

ProductNo,

ProductName,

COUNT(\*) AS NumTransactions,

MIN(TransactionQuantity) AS MinQuantity,

MAX(TransactionQuantity) AS MaxQuantity,

AVG(TransactionQuantity) AS AvgQuantity,

SUM(TransactionQuantity) AS SumQuantity

FROM SalesReports.YearlySalesReport

WHERE StoreNo = 'S00001'

AND ProductNo = 'P0000001112'

AND YEAR(CalendarDate) = 2010

GROUP BY YEAR(CalendarDate),

MONTH(CalendarDate),

StoreNo,

ProductNo,

ProductName

ORDER BY YEAR(CalendarDate),

MONTH(CalendarDate),

StoreNo,

ProductNo,

ProductName

GO

Nothing fancy, no OVER() clause, no CTE, just a SELECT clause that retrieves rows from the YearlySalesReport table. We include the COUNT(),MIN(),MAX(),AVG() and SUM() functions and pass the TransactionQuantity column as a parameter.

A WHERE clause filters the results, so we only get rows for store S00001, product P0000001112 and calendar year 2010.

The mandatory GROUP BY clause is included when you use aggregate functions, and we utilize an ORDER BY clause to sort the results. Please refer to the partial results in figure 2.9 below:

Graphical user interface, application, table, Excel

Description automatically generated

***Figure 2.9 – Sales Profile Report***

Seems like February was a slow month which is strange as you have the Valentine’s Day holiday. You would think sales would be up.

**With OVER()**

Let’s use the prior query and add some window capabilities. There are two parts to this query. First we need to create a CTE (common table expression) called ProductPurchaseAnalysis that performs a little date manipulation and also includes the COUNT() function to generate the count of transactions by year, month, store number and product number.

The second part to the solution is to create a query that references the CTE and uses all the basic aggregate functions but this time we will include an OVER()clause to create and order rows for processing in the partitions. Let’s examine the code.

Please refer to listing 2.3 below:

***Listing 2.3 – Part 1 the CTE***

WITH ProductPurchaseAnaysis (

PurchaseYear,PurchaseMonth,CalendarDate,StoreNo,CustomerFullName,ProductNo,ItemsPurchased,NumTransactions

)

AS (

SELECT YEAR(CalendarDate) AS PurchaseYear,

MONTH(CalendarDate) AS PurchaseMonth,

CalendarDate,

StoreNo,

CustomerFullName,

ProductNo,

TransactionQuantity AS ItemsPurchased,

COUNT(\*) AS NumTransactions

FROM SalesReports.YearlySalesReport

GROUP BY YEAR(CalendarDate) ,

MONTH(CalendarDate),

CalendarDate,

StoreNo,

CustomerFullName,

ProductNo,

ProductName,

TransactionQuantity

)

Pretty straight forward. The calendar year and month are pulled out from the calendar date which is included in the SELECT clause. The store number, customer’s full name plus the product number is also included. Lastly the transaction quantity, meaning how many items were purchased is included and we count the number of transactions within each group as defined by the GROUP BY clause.

Now for the second part of this batch, the query that references the CTE. This query will give us a profile of the number of transactions and quantities purchased for each product by year, month, data, customer, and store.

Please refer to listing 2.4 below:

***Listing 2.4 – Part 2 – Using Window Functions***

SELECT PurchaseYear,PurchaseMonth,CalendarDate,StoreNo,

CustomerFullName,ProductNo,NumTransactions,

SUM(NumTransactions) OVER (

PARTITION BY PurchaseYear,CustomerFullName

ORDER BY CustomerFullName,PurchaseMonth

) AS SumTransactions,ItemsPurchased,

SUM(ItemsPurchased) OVER (

PARTITION BY PurchaseYear,CustomerFullName

ORDER BY CustomerFullName,PurchaseMonth

) AS TotalItems,

AVG(CONVERT(DECIMAL(10,2),ItemsPurchased)) OVER (

PARTITION BY PurchaseYear,CustomerFullName

ORDER BY CustomerFullName,PurchaseMonth

) AS AvgPurchases,

MIN(ItemsPurchased) OVER (

PARTITION BY PurchaseYear,CustomerFullName

ORDER BY CustomerFullName,PurchaseMonth

) AS MinPurchases,

MAX(ItemsPurchased) OVER (

PARTITION BY PurchaseYear,CustomerFullName

ORDER BY CustomerFullName,PurchaseMonth

) AS MaxPurchases

FROM ProductPurchaseAnalysis

WHERE StoreNo = 'S00001'

AND ProductNo = 'P0000001112'

AND PurchaseYear = 2010

AND PurchaseMonth = 1

AND ItemsPurchased > 0

GROUP BY PurchaseYear,PurchaseMonth,CalendarDate,StoreNo,

CustomerFullName,ProductNo,NumTransactions,ItemsPurchased

ORDER BY CustomerFullName,PurchaseYear,PurchaseMonth,CalendarDate,StoreNo,

ProductNo,ItemsPurchased

GO

Notice I included a filter to only report results for January of 2010. This was done in order to keep the result sets small so we can debug and analyze the query. Once it works correctly, you can remove these filters to get all months (when you download the code). You can also remove the PurchaseYear filter predicate to get more years. If you want this information for all stores and customers, you can modify the query so it inserts the results in a report table and then you can write queries to view any combination of customers, store and products that you wish.

Let’s walk through the rest of the code.

An ORDER BY clause and a PARTITION BY clause is included in each OVER() clause. The data set results are partitioned by the PurchaseYear and CustomerFullName columns. The partition rows are sorted by CustomerFullName and PurchaseMonth.

As mentioned earlier, a WHERE clause is included to filter results by store ‘S00001’, product ‘P0000001112’, Purchase Year 2010 and for the month of January. I also filter out any ItemsPurchased values equal to zero. (They somehow sneaked in during the table load process of the test data. This number is randomly generated so zero values will sneak in!).

Let’s see the results. Please refer to the figure 2.10 below:

Graphical user interface, application, table, Excel

Description automatically generated

***Figure 2.10 – Transaction Quantities & Items Purchased Profile***

Let’s focus on purchases made by Bill Brown. There are 4 dates on which purchases were made. Assuming one purchase per day this comes to a total of 4 transactions which appear in the SumTransactions column.

Next, if we look at the ItemsPurchased column we see the 4 values corresponding to each of the 4 days Bill made purchases. On January 28 and 29 he made one purchase each day. On January 30 he made 2 purchases, and on January 31 he made 3 purchases. If we add them all up, we get a total of 7 purchases which is reflected in the ItemsPurchased column.

Notice the default behavior of the PARTITION BY and ORDER BY clause if no window frame clause is included. The totals appear in each row of the result. This means include any previous and following rows (relative to the current row being processed) as each total value for the current row is calculated.

The same applies to the average calculation. Lastly, notice the minimum and maximum values. They can be clearly seen in the ItemsPurchased column. The minimum purchase Bill made were 1 item and the largest purchase he made was 3 items.

This processing pattern repeats itself for each set of rows defined by the partition.

**GROUPING() Function**

The grouping function allows you to create rollup values like sums for each level of a hierarchy defined by categories of data (think of rollups in Excel pivot tables). A picture is worth a thousand words and clarifies the results of this function. Please refer to the figure 2.11 below:

Timeline

Description automatically generated

***Figure 2.11 – Conceptual Model of Rollups within Groups of Data***

Starting with the 3 categories of rows to the left of the diagram, each row in the data set has a category and a value. The categories are A, B, and C. If we summarize the values for each category at level 0, we get rollups of 15 for category A, 30 for category B and 45 for category C at level 1. If we summarize these 3 rollup totals, we get a final value of 90 at level 2.

This looks easy but imagine the complexity if you must summarize data over 5 or more levels in the group hierarchy. For example, we can summarize by year, month, product category, product subcategory and product. Let's throw in customer too so we can see totals by customer! This could make the output difficult to read and analyze so make sure you provide the minimum required information that satisfies user requirements.

Armed with this knowledge, let’s write a query that will create rollups for our sales data and try to use a clever trick in the ORDER BY clause to make the results easy to navigate.

Please refer to listing 2.5 below:

***Listing 2.5 – Generating a Rollup Report***

WITH StoreProductSalesAnalysis

(TransYear,TransQuarter,TransMonth,TransDate,StoreNo,ProductNo,MonthlySales)

AS

(

SELECT

YEAR(CalendarDate) AS TransYear,

DATEPART(qq,CalendarDate) AS TransQuarter,

MONTH(CalendarDate) AS TransMonth,

CalendarDate AS TransDate,

StoreNo,

ProductNo,

SUM(TotalSalesAmount) AS MonthlySales

FROM FactTable.YearlySalesReport

GROUP BY

CalendarDate,

StoreNo,

ProductNo

)

SELECT TransYear,

TransQuarter,

TransMonth,

StoreNo,

ProductNo,

MonthlySales,

SUM(MonthlySales) AS SumMonthlySales,

GROUPING(MonthlySales) AS RollupFlag

FROM StoreProductSalesAnalysis

WHERE TransYear = 2011

AND ProductNo = 'P0000001103'

AND StoreNo = 'S00001'

GROUP BY TransYear,

TransQuarter,

TransMonth,

StoreNo,

ProductNo,

MonthlySales WITH ROLLUP

ORDER BY TransYear,

TransQuarter,

TransMonth,

StoreNo,

ProductNo,

(

CASE

WHEN MonthlySales IS NULL THEN 0

END

) DESC,

GROUPING(MonthlySales) DESC

GO

We use the same CTE as the prior example, but the query has some complexity to it. We use the SUM() function to generate summaries for monthly sales and the GROUPING() function to create rollups by the columns included in the GROUP BY clause. The GROUPING() function appears right after the last column in the SELECT clause.

Why are we doing all this?

Specifically, we want to rollup sales summaries by year, quarter, month, store number and product number. These represent the levels in our rollup hierarchy.

We need to include some clever code in the ORDER BY clause so that the display looks a little bit like the diagram we just discussed. Each time we roll up a level a lot of NULLS appear in the rows for the values that no longer apply to the level in the hierarchy because we summarized the prior levels values. If you do not sort the results correctly, it will be impossible to understand.

What we need to do is introduce a CASE statement in the ORDER BY clause right after the ProductNo column that will tell the query what to do with the monthly Sales column if it is NULL:

ProductNo,

(

CASE

WHEN MonthlySales IS NULL THEN 0

END

) DESC,

If the value is NULL then use 0 and sort in descending order. Notice the GROUPING() function at the end. For these values sort in descending order also. For all other columns in the ORDER BY clause sort in ascending order. (Reminder: if you do not include the ASC keyword then the default is ascending sort.)

Let’s execute the query and see the results.

Please refer to figure 2.12 below:

Graphical user interface, table

Description automatically generated

***Figure 2.12 – A Rollup Report***

Not bad if I say so myself!

The results look like the ones in the conceptual diagram we discussed earlier (maybe I’m stretching it a bit).

As you navigate up the hierarchy, more NULLS appear. Also notice the SummaryLevel column. If the value is zero, then we are at a leaf row (or lowest level node) in the hierarchy. If the value is 1, we are at a summary row somewhere in the hierarchy.

Checking out the two highlighted sets of rows, if we add $150.00, $168.75, $206.25, and $225.00 we get a total value of $750.00. If we examine the second group and add $187.50, $262.50, $281.25, and $318.75 we get $1050.00. This last value is the summary for the level group defined by year, quarter, month, store, and product.

This is only a partial screenshot, but the rollup pattern repeats for each set of rows in the hierarchy group.

Peaking at the $2062.50 total in row 3. This represents the rollup for the first quarter in the year. row 2 contains the total for the year and row 1 the grand total. It is the same as the year total. If we would have included other years this total would of course be much larger.

Try this query on your own by downloading the code in the book’s dedicated Google website and include 2- or 3-years’ worth of data to see what the results look like.

**Tip:** with queries like this, always start off with small result sets by using the WHERE clause so you can make sure everything tallies up and of course sorts correctly so you can generate legible and understandable output. Once you are satisfied it works correctly, then you can remove the WHERE clause filters, so the original query specification is satisfied. (Test on small sets of data!)

**GROUPING - Performance Tuning Considerations**

Although chapter 14 is dedicated to performance tuning, let me sneak in an example. Below is the query plan for the query we just discussed.

Please refer to figure 2.13 below:

Graphical user interface, application, timeline

Description automatically generated

***Figure 2.13 – Query Plan for the GROUPING() Function Example***

Notice the Index Seek to the right of the plan. Even though we are using an index the query plan estimator is suggesting we add a different index. The index seek has 0% cost but the RID (Row ID) Lookup task cost 99%. The RID lookup is performed against the YearlySalesReport table based on the RID in the index. So, this looks very expensive. The remaining tasks have 0% costs so we will ignore these for now.

Let’s copy the suggested index template by right clicking on it and selecting “Missing Index Details”. It appears in a new query window in SSMS. All we really need to do is give the new index a name.

Please refer to listing 2.6 below

***Listing 2.6 – suggested new index***

/\*

Missing Index Details from chapter 02 - TSQL code - 09-13-2022.sql - DESKTOP-CEBK38L\GRUMPY2019I1.APSales (DESKTOP-CEBK38L\Angelo (55))

The Query Processor estimates that implementing the following index could improve the query cost by 98.1615%.

\*/

/\*

USE [APSales]

GO

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

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

INCLUDE ([CalendarDate],[TotalSalesAmount])

GO

\*/

DROP INDEX IF EXISTS [ieProductNoStoreNoDateTotalSalesAmt]

ON [SalesReports].[YearlySalesReport]

GO

CREATE NONCLUSTERED INDEX [ieProductNoStoreNoDateTotalSalesAmt]

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

INCLUDE ([CalendarDate],[TotalSalesAmount])

GO

Copy the suggested index template and paste it below the comments. All you need to do is give it a name: ieProductNoStoreNoDateTotalSalesAmt.

As we are learning about indexes and performance tuning, I decided to give verbose index names so when you are working with them, you know what columns are involved. This is a non-clustered index. Notice how the date and sales amount columns are included right after the INCLUDE keyword. Let’s create the index and generate a new query plan.

We need to split up the query plan as it is fairly large, please refer to the two following figures.

Please refer to figure 2.14 below for the first part:

Graphical user interface

Description automatically generated with medium confidence

***Figure 2.14 – Revised Query Plan Part A***

The RID Lookup is gone, so is the Nested Loops join. Our index seek now has a 26% cost associated with it. There is also a Sort step of the data stream at a cost of 15%. Let’s look at the second part of the query plan. Please refer to figure 2.15 below:

Graphical user interface

Description automatically generated

***Figure 2.15 – Revised Query Plan Part B***

Some more compute scalar and stream aggregate steps appear but at a cost of 0%. We will ignore these. There is a second Sort step with a cost of 15%. Is this better?

Let’s re-run the query but before we do we need to clear the memory buffers and generate some IO statistics by executing the following commands:

DBCC dropcleanbuffers;

CHECKPOINT;

GO

-- turn set statistics io/time on

SET STATISTICS IO ON

GO

SET STATISTICS TIME ON

GO

The DBCC command will make sure the memory buffers are cleared out, so we start with a clean slate. If you do not perform this step, older query plans will be in memory cache, and you will get erroneous results.

The next two commands will generate our IO and timing statistics. Let’s see what results we get when we execute the query.

Please refer to table 2.3 below:

| **SQL Server Parse & Compile Time** | **No Index** | **Index** |
| --- | --- | --- |
| CPU Time (ms) | 15 | 16 |
| elapsed time (ms) | **4492** | **89** |
|  |  |  |
| **Statistic (YearlySalesReport)** | **No Index** | **Index** |
| Scan Count | 1 | 1 |
| Logical Read | **260** | **10** |
| Physical Reads | 1 | 1 |
| read-ahead reads | **51** | **7** |
|  |  |  |
| **SQL Server Execution Times** | **No Index** | **Index** |
| CPU Time (ms) | **83** | **74** |
| elapsed time (ms) | 0 | 0 |

***Table 2.3 – Comparing Statistics Between Plans***

Checking out the statistics we see CPU time went up by 1ms (millisecond) but elapsed time went down from 4492ms to 89ms. Scan count remained the same but logical reads dropped to 10 from 260. This is good.

Physical reads remained the same at 1 and read-ahead reads dropped from 51 to 7. Finally, SQL Server Execution CPU time dropped from 83ms to 74ms. Overall, it looks like this new index improved performance.

This was a quick example of the steps that you can take to analyze performance. As stated earlier, chapter 14 will entirely be dedicated to performance tuning on selected queries we covered. Let’s look at our next function, the STRING\_AGG() function to tie some strings together (pun intended!).

**STRING\_AGG Function**

What does this do?

This function allows you to aggregate strings. Just as the name implies. This comes in handy when you need to list items of interest, like when you want to show all stops in a flight itinerary, or in our case, items purchased on a specific date. Another interesting example is if you need to list stops in a logistics scenario, like all the stops a truck will make when delivering the chocolate candies and cakes sold by the stores in our examples.

This function is easy to use. You need to supply the name of the column and a delimiter which is usually a comma.

Let’s take a look at a simple example that shows what items customer C00000008 purchased. We want results by year and month.

Please refer to listing 2.7 below:

***Listing 2.7 – Product Report using STRING\_AGG()***

WITH CustomerPurchaseAnalysis(PurchaseYear,PurchaseMonth,CustomerNo,ProductNo,PurchaseCount)

AS

(

SELECT DISTINCT

YEAR(CalendarDate) AS PurchaseYear,

MONTH(CalendarDate) AS PurchaseMonth,

CustomerNo,

ProductNo,

COUNT(\*) AS PurchaseCount

FROM StagingTable.SalesTransaction

GROUP BY YEAR(CalendarDate),

MONTH(CalendarDate),

CustomerNo,

ProductNo

)

SELECT

PurchaseYear,

PurchaseMonth,

CustomerNo,

STRING\_AGG(ProductNo,',') AS ItemsPurchased,

COUNT(PurchaseCount) AS PurchaseCount

FROM CustomerPurchaseAnalysis

WHERE CustomerNo = 'C00000008'

GROUP BY

PurchaseYear,

PurchaseMonth,

CustomerNo

ORDER BY CustomerNo,

PurchaseYear,

PurchaseMonth

GO

Notice how the STRING\_AGG() function is used. The ProductNo column is passed together with a comma as the delimiter. A CTE is used so we can pull out the year and date from the CalendarDate column and count the number of purchases.

In this example, the count value will always be the value 1 in the CTE, so we need to count the occurrences to get the total number value of the purchases. The resulting purchase count and the aggregate product numbers should match up, as far as the number of items purchased.

**Tip:** always test the query in the CTE to make sure it works as expected!

Let’s take a look at the results. Please refer to figure 2.16 below:

Graphical user interface, application, table, Excel

Description automatically generated

***Figure 2.16 – Purchase Product List Report***

This looks correct. The ItemsPurchased column lists 4 items and the PurchaseCount column has a value of 4. It seems this customer always buys the same items each month and the same quantity. When you like something stick with it, what can I say?

You can validate the results by modifying the CTE query, so it filters by the customer:

SELECT DISTINCT

YEAR(CalendarDate) AS PurchaseYear,

MONTH(CalendarDate) AS PurchaseMonth,

CalendarDate,

CustomerNo,

ProductNo,

COUNT(\*) AS PurchaseCount

FROM StagingTable.SalesTransaction

WHERE CustomerNo = 'C00000008'

GROUP BY YEAR(CalendarDate),

MONTH(CalendarDate),

CalendarDate,

CustomerNo,

ProductNo

ORDER BY

YEAR(CalendarDate),

MONTH(CalendarDate),

CalendarDate,

CustomerNo,

ProductNo

GO

Something for you to try. To make things interesting, tweak the load script that loads the SalesTransaction table so that different products are bought each month.

It’s time for some statistics.

**STDEV()/STDEVP() Functions**

The STDEV() function calculates the statistical standard deviation of a set of values when the entire population is not known or available.

The STDEVP() function calculates the statistical standard deviation when the entire population of the data set is known.

But what does this mean?

What this means is how close or how far apart the numbers are in the data set from the arithmetic **MEAN** (which is another word for average). In our context the average or **MEAN** is the sum of all data values in question divided by the number of data values.

There are other types of **MEAN** like geometric, harmonic and weighted **MEAN** which I will briefly cover in appendix 2.

So how is the standard deviation calculated?

You could use the following algorithm to calculate the standard deviation with TSQL:

* Calculate the average (**MEAN**) of all values in question and store it in a variable.
* For each value in the data set
  + subtract the calculated average.
  + square the result.
* Take the sum of all the results and divide the value by the number of data elements minus 1. (This step calculates the variance, to calculate for the entire population, do not subtract 1)
* Now take the square root of the variance and that’s your standard deviation.

Or you could just use the STDEV() function! There is a STDEVP() function as mentioned earlier. As stated earlier, the only difference is that STDEV() works on part of the population of the data. This is when the entire data set is not known. The STDEVP() function works on the data set that represents the entire population.

Here is the syntax:

SELECT STDEV( ALL | DISTINCT ) AS [Column Alias]

FROM [Table Name]

GO

You need to supply a column name that contains the data you want to use. You can optionally supply the ALL or DISTINCT keyword if you wish. If you leave these out, just like the other functions the behavior will default to ALL. Using the DISTINCT keyword will ignore duplicate values in the column.

Let’s look at a TSQL query that uses this window function. We want to calculate the standard deviation plus the average by year, customer, store number, product number and customer.

As usual we start off with a CTE to do some preprocessing and return our data set.

Please refer to listing 2.8 below:

***Listing 2.8 – Standard Deviation Sales Analysis***

WITH CustomerPurchaseAnalysis

(PurchaseYear,PurchaseMonth,StoreNo,ProductNo,CustomerNo,TotalSalesAmount)

AS

(

SELECT

YEAR(CalendarDate) AS PurchaseYear,

MONTH(CalendarDate) AS PurchaseMonth,

StoreNo,

ProductNo,

CustomerNo,

SUM(TransactionQuantity \* UnitRetailPrice) AS TotalSalesAmount

FROM StagingTable.SalesTransaction

GROUP BY YEAR(CalendarDate),MONTH(CalendarDate),ProductNo,CustomerNo,StoreNo

)

SELECT

cpa.PurchaseYear,

cpa.PurchaseMonth,

cpa.StoreNo,

cpa.ProductNo,

c.CustomerNo,

CONVERT(DECIMAL(10,2),cpa.TotalSalesAmount) AS TotalSalesAmount,

AVG(CONVERT(DECIMAL(10,2),cpa.TotalSalesAmount)) OVER(

--PARTITION BY cpa.PurchaseYear,c.CustomerNo

ORDER BY cpa.PurchaseYear,c.CustomerNo

) AS AvgPurchaseCount,

STDEV(CONVERT(DECIMAL(10,2),cpa.TotalSalesAmount)) OVER(

ORDER BY cpa.PurchaseMonth

) AS StdevTotalSales,

STDEVP(CONVERT(DECIMAL(10,2),cpa.TotalSalesAmount)) OVER(

ORDER BY cpa.PurchaseMonth

) AS StdevpTotalSales,

STDEV(CONVERT(DECIMAL(10,2),cpa.TotalSalesAmount)) OVER(

) AS StdevTotalSales,

STDEVP(CONVERT(DECIMAL(10,2),cpa.TotalSalesAmount)) OVER(

) AS StdevpYearTotalSales

FROM CustomerPurchaseAnalysis cpa

JOIN DimTable.Customer c

ON cpa.CustomerNo = c.CustomerNo

WHERE cpa.CustomerNo = 'C00000008'

AND PurchaseYear = 2011

AND ProductNo = 'P00000038114';

GO

The CTE pulls out the year and month parts from the date. We also calculate the total sales amount by multiplying the transaction quantity by the unit retail price.

The average is calculated for the entire year as we wish to use that value in an Excel spreadsheet to calculate and graph the normal distribution bell curve. We will see it shortly.

Next, the standard deviation values are calculated by using the STDEV() and STDEVP() functions. We perform the calculations two different ways. The first pair uses an ORDER BY clause in the OVER() clause so that we get rolling monthly results. The second pair does not use a PARTITION BY or ORDER BY clause, so the default frame behavior kicks in: to process results, including all prior and following rows relative to the current row.

Let’s see the results. Please refer to figure 2.17 below:

Graphical user interface, application, table

Description automatically generated

***Figure 2.17 – Calculating Standard Deviation.***

Short and sweet. Results for one customer, one year by month. Notice the NULL value for the first standard deviation results in column StdevTotalSales1. This happens because it is the first value in the partition, and it cannot calculate the standard deviation for one value so NULL is the result. The STDEVP() function though is happy. It simply returns 0. For the remaining months the rolling standard deviation is calculated month by month.

If you do not like the NULL, just use a CASE block that tests the value returned and substitute it with a zero!

Column StdevTotalSales2 uses STDEVP() and the values as can be seen are a bit lower since the entire population of the data set is used.

Finally, columns StdevTotalSales3 and StdevTotalSales4 calculate the standard deviation for the entire year. We will use the STDEV() year result in our Excel spreadsheet to generate normal distribution values and generate a graph called a bell curve based on each normal distribution value.

Let’s check out the graph now. Please refer to figure 2.18 below:

Graphical user interface, application, table

Description automatically generated

***Figure 2.18 – Standard Deviation Bell curve(s)***

What we have here are a series of small bell curves. For example, notice the curve between months 2 and 4 and 4 and 6. In this case each curve is generated from 3 values, a low one, then a high one and back to a low value. The curve generated for months 2 through 4 has values 0.068514,0.759391, and 0.194656 These are great values for the bell curve.

To generate the normal distribution for each month the Excel NORM.DIST function was used. This function uses these parameters: the value, the average (MEAN) and the standard deviation generated by our query. We then insert a suggested graph for these values.

**STDEV - Performance Tuning Considerations**

Let’s do some performance analysis. By highlighting the query we just discussed and generating a query plan in one of the usual methods, we are presented with a plan that appears to already use an index.

Please refer to figure 2.19 below:

Timeline

Description automatically generated

***Figure 2.19 – First Analysis Query Plan***

Starting from right to left we see an Index Seek with a cost of 4% and a RID (Row ID lookup) task with an estimated cost of 80%. This is where most of the work seems to be happening.

Ignoring the tasks with 0 cost we see a Table Scan on the customer table with a cost of 7% and a Sort step with a cost of 7%. Finally, we see a Nested Loops inner join step with a cost of 1%.

This is a partial screenshot, I ignored the steps with zero cost so we can concentrate on the high-cost steps (In your queries, take all steps into account).

Notice that a suggested index was generated. Hold the mouse pointer over the index (it will be in green font) and right click on the index. When the pop-up menu appears, click on “**Missing Index Details …**” to generate the TSQL code in a new query window. The code below is presented.

Please refer to Listing 2.9 below:

***Listing 2.9 – Estimated Query Plan - Suggested Index***

/\*

Missing Index Details from chapter 02 - TSQL code - 09-13-2022.sql - DESKTOP-CEBK38L\GRUMPY2019I1.APSales (DESKTOP-CEBK38L\Angelo (65))

The Query Processor estimates that implementing the following index could improve the query cost by 80.174%.

\*/

/\*

USE [APSales]

GO

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

ON [StagingTable].[SalesTransaction] ([CustomerNo],[ProductNo])

INCLUDE ([StoreNo],[CalendarDate],[TransactionQuantity],[UnitRetailPrice])

GO

\*/

/\* Copy code from above and paste and supply name \*/

DROP INDEX IF EXISTS [CustNoProdNoStoreNoDateQtyPrice]

ON [StagingTable].[SalesTransaction]

GO

CREATE NONCLUSTERED INDEX [CustNoProdNoStoreNoDateQtyPrice]

ON [StagingTable].[SalesTransaction] ([CustomerNo],[ProductNo])

INCLUDE ([StoreNo],[CalendarDate],[TransactionQuantity],[UnitRetailPrice])

GO

All I needed to do was copy and paste the suggested index template and supply a name for the index. I used all the columns related to the index in the name, so it is clear what the index covers.

In a real production environment, you would most likely just give it a shorter name but as we are in learning mode, I decided to be a bit verbose as far as the names are concerned. I also added a DROP INDEX command so that it is available to you when you start analyzing this query on your own.

Let’s create the index, generate a new query plan, and then execute the query so we generate some statistics with the SET STATISTICS commands.

Please refer to figure 2.20 below for the new plan:

A picture containing graphical user interface

Description automatically generated

***Figure 2.20 – Revised Query Plan***

Some definite improvements. Starting from right to left in the query plan, we now see an Index Seek on the SalesTransaction table at a cost of 12%. A fairly expensive Sort step at 40% appears next. The customer table uses a Table Scan at a cost of 38% which cannot be helped because we did not create an index for this table. This table has only 100 rows so an index would most likely not be used but on your own, you can create one and experiment to see if there are any benefits.

Right in the middle there is a Nested Loops Inner Join at 7% cost. Not too expensive. There is also a final Nested Loops Inner Join at 2%. For our discussion we will ignore the tasks at 0% cost.

So, is this an improvement? Looks like it because of the Index Seek. The IO and TIME statistics will give us a clearer picture though so let’s look at them.

Please refer to Table 2.4 below:

| **SQL Server Parse & Compile Time** | **Existing Index** | **New Index** |
| --- | --- | --- |
| CPU Time (ms) | 0 | 31 |
| elapsed time (ms) | 11 | 79 |
|  |  |  |
| **Statistic (work table)** | **Existing Index** | **New Index** |
| Scan Count | 18 | 18 |
| Logical Read | 133 | 133 |
| Physical Reads | 0 | 0 |
| read-ahead reads | 0 | 0 |
|  |  |  |
| **Statistic (SalesTransaction)** | **Existing Index** | **New Index** |
| Scan Count | 1 | 1 |
| Logical Read | **43** | **5** |
| Physical Reads | 0 | 0 |
| read-ahead reads | 0 | 0 |
|  |  |  |
| **Statistic (Customer)** | **Existing Index** | **New Index** |
| Scan Count | 1 | 1 |
| Logical Read | 216 | 216 |
| Physical Reads | 0 | 0 |
| read-ahead reads | 0 | 0 |
|  |  |  |
| **SQL Server Execution Times** | **Existing Index** | **New Index** |
| CPU Time (ms) | 16 | 0 |
| elapsed time (ms) | 4 | 30 |

***Table 2.4 – Comparing IO Statistics***

This table shows statistics before and after the index was created.

Some interesting results, I think. Starting with SQL Server Parse & compile time and elapsed time statistics actually went up with the new index, not down.

Statistics for the work table remained the same while statistics for the SalesTransaction table went down significantly. Logical reads went from 43 to 5 which is very good. Physical reads and read-ahead reads remained at 0.

Finally, at the end of processing, for SQL Server Execution time CPU time went from 16 to zero so that’s a significant improvement while elapsed time increased from 4 to 30 milliseconds.

Conclusion, adding the index reduced logical reads but elapsed time for parsing and execution increased.

The SalesTransaction table has about 24,886 rows which is not a lot. Real performance improvements or changes would be noticeable if we had a couple of million rows.

The table below represents before and after statistics after I loaded about 1.75 million rows by removing any WHERE clause filters in the load script. (This is available in the load script for this chapter on the publishers Google site.) Please refer to table 2.5 below.

| **SQL Server Parse & Compile Time** | **Existing Index** | **New Index** |
| --- | --- | --- |
| CPU Time (ms) | **32** | **0** |
| elapsed time (ms) | **62** | **0** |
|  |  |  |
| **Statistic (work table)** | **Existing Index** | **New Index** |
| Scan Count | 18 | 18 |
| Logical Read | 133 | 134 |
| Physical Reads | 0 | 0 |
| read-ahead reads | 0 | 0 |
|  |  |  |
| **Statistic (SalesTransaction)** | **Existing Index** | **New Index** |
| Scan Count | **5** | **1** |
| Logical Read | **41840** | **76** |
| Physical Reads | 0 | 3 |
| read-ahead reads | **38135** | **71** |
|  |  |  |
| **Statistic (Customer)** | **Existing Index** | **New Index** |
| Scan Count | 1 | 1 |
| Logical Read | 18 | 18 |
| Physical Reads | 0 | 0 |
| read-ahead reads | 0 | 0 |
|  |  |  |
| **SQL Server Execution Times** | **Existing Index** | **New Index** |
| CPU Time (ms) | 485 | 0 |
| elapsed time (ms) | **2926** | **48** |

***Table 2.5 – Comparing Statistics on 2 Million Rows***

Now we really have some interesting statistics. For the SQL Server Parse and Compile time step CPU and elapsed time statistics went down from 32 and 62 respectively to 0.

Statistics for the work table remained the same.

For the SalesTransaction we really see significant improvement in the logical read category, from 41840 reads before the index was created to 76. Physical reads went up a bit from 0 to 3 while read-ahead reads went down from 38135 to 71.

Statistics for the customer table remained the same, both before the index was created and after the index was created. (Makes sense as the index was created on a different table).

Wrapping it up with the SQL Server Execution times the CPU time went from 485ms to 0ms, and the elapsed execution time went from 2926ms to 48ms.

Looks like we really need a significant amount of data rows to work with to see how indexes improve or in some case decrease performance.

One last comment, the statistics generated will vary from time to time as you keep performing analysis on your queries so make sure you use the DBCC command to clear all memory buffers.

Also, run the DBCC and SET STATISTICS ON steps separately from the query otherwise if you run them together you will get statistics for both these little steps and the query being evaluated. This could be confusing.

**VAR()/VARP() Functions**

The VAR() and VARP() function is used to generate the variance for a set of values in a data sample. The same discussion related to the data sets used in the STDEV()/STDEVP() examples applies to the VAR() and VARP() functions relative to population of the data:

The VAR() function works on a partial set of the data when the entire data set is not known or available, while the STDEVP() function works on the whole data set population when it is known.

So, if your data set consists of 10 rows, these function names ending with P will include all 10 rows when processing the values, the ones not ending in the letter P will look at N – 1 or 10 - 1 = 9 rows.

OK, so what does variance mean anyway?

The VAR() and VARP() functions are used to generate the variance for a set of values in a data sample. The same discussion related to the data sets used in the STDEV()/STDEVP() examples applies to the VAR() and VARP() functions relative to population of the data:

The VAR() function works on a partial set of the data when the entire data set is not known or available while the VARP() function works on the whole data set population when it is known.

Informally, if you take all the differences of each value from the mean (average) and square the results and then add all the results and finally divide them by the number of data samples you get the variance.

For our scenario, let’s say we had some data that forecasted sales amounts for one of our products during any particular year. When sales data was finally generated for the product, we wanted to see how close or far the sales figures were from the forecast. It answers the questions:

* Did we miss our target?
* Did we meet our targets?
* Did we exceed our targets?

This statistical formula is also discussed in appendix 2 by the way. We will develop TSQL code to duplicate the STDEV() and VAR() functions. Let’s see if they get the same results!

Let’s check out our query that uses these functions.

Please refer to listing 2.10 below:

***Listing 2.10 – Calculating Sales Variance***

WITH CustomerPurchaseAnalysis

(PurchaseYear,PurchaseMonth,StoreNo,ProductNo,CustomerNo,TotalSalesAmount)

AS

(

SELECT

YEAR(CalendarDate) AS PurchaseYear,

MONTH(CalendarDate) AS PurchaseMonth,

StoreNo,

ProductNo,

CustomerNo,

SUM(TransactionQuantity \* UnitRetailPrice) AS TotalSalesAmount

FROM StagingTable.SalesTransaction

GROUP BY YEAR(CalendarDate),MONTH(CalendarDate),

ProductNo,CustomerNo,StoreNo

)

SELECT

cpa.PurchaseYear,

cpa.PurchaseMonth,

cpa.StoreNo,

cpa.ProductNo,

c.CustomerNo,

c.CustomerFullName,

CONVERT(DECIMAL(10,2),cpa.TotalSalesAmount) AS TotalSalesAmount,

AVG(CONVERT(DECIMAL(10,2),cpa.TotalSalesAmount)) OVER(

ORDER BY cpa.PurchaseMonth) AS AvgPurchaseCount,

VAR(CONVERT(DECIMAL(10,2),cpa.TotalSalesAmount)) OVER(

ORDER BY cpa.PurchaseMonth

) AS VarTotalSales,

VARP(CONVERT(DECIMAL(10,2),cpa.TotalSalesAmount)) OVER(

ORDER BY cpa.PurchaseMonth

) AS VarpTotalSales,

VAR(CONVERT(DECIMAL(10,2),cpa.TotalSalesAmount)) OVER(

) AS VarTotalSales,

VARP(CONVERT(DECIMAL(10,2),cpa.TotalSalesAmount)) OVER(

) AS VarpYearTotalSales

FROM CustomerPurchaseAnalysis cpa

JOIN DimTable.Customer c

ON cpa.CustomerNo = c.CustomerNo

WHERE cpa.CustomerNo = 'C00000008'

AND PurchaseYear = 2011

AND ProductNo = 'P00000038114';

GO

Basically I did a copy and paste of the standard deviation query and just replaced the STDEV() functions with the VAR() functions. The usual CTE is used. I also commented out the product and customer columns as they are for one customer and one product as can be identified in the WHERE clause. I did this as the results had too many columns and they did not display well.

When you test out these queries, please feel free to uncomment them and get results for more than one customer. Also display the customers and product.

Let’s see the results. Please refer to figure 2.21 below:

Graphical user interface, application, table, Excel

Description automatically generated

***Figure 2.21 – Comparing Variance versus Variance for the Whole Population***

The average and the first two variance columns are rolling monthly values as can be seen by the different values for each month. These only have an ORDER BY clause, so the default window frame behavior is:

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

The last two columns do not specify either a PARTITION BY or an ORDER BY clause, so the default behavior is:

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

If you have not done so already, download the code for this chapter and try these examples out. For this example, uncomment the customer and product related columns. When you are comfortable with the code, also try out the query for multiple years, then multiple customers and products. Lastly, try playing around with the RANGE and ROWS clauses to see how it affects window frame behavior.

One comment on report generation. What you do not want to do is generate a report that has so much data that it cannot be read. You can create a query that SSRS (SQL Server Reporting Services) can use that includes data for all customers, products, and years. All you need to do is add some filters in the SSRS report so the user can pick and choose what combinations of the filters they want to view from a drop down list.

**SQL Server 2022 – Named Window Example**

Let’s check out the new WINDOW feature in SQL Server 2022 by creating a query that uses the AVG() function with the OVER() clause. We will gather the IO and TIME statistics plus generate a query plan in the usual manner to see how this new feature affects performance.

If the query plan suggests an index, which it will as there are none, we will create the index and then run the query again and look at the new plan and statistics.

**Note:** I ran this on an evaluation copy of SQL Server 2022. Feel free to download and install it. Also do some research on the other new features of this version. Very important to keep up with the changes of the latest editions.

Here’s the code. Please refer to listing 2.11 below for the query:

***Listing 2.11 – Average by Year,Month and Customer***

WITH CustomerPurchaseAnalysis

(PurchaseYear,PurchaseMonth,CustomerNo,TotalSalesAmount)

AS

(

SELECT

YEAR(CalendarDate) AS PurchaseYear,

MONTH(CalendarDate) AS PurchaseMonth,

CustomerNo,

SUM(TransactionQuantity \* UnitRetailPrice) AS TotalSalesAmount

FROM StagingTable.SalesTransaction

GROUP BY YEAR(CalendarDate),MONTH(CalendarDate),CustomerNo

)

SELECT

cpa.PurchaseYear,

cpa.PurchaseMonth,

c.CustomerNo,

c.CustomerFullName,

cpa.TotalSalesAmount,

AVG(cpa.TotalSalesAmount) OVER SalesWindow AS AvgTotalSales

FROM CustomerPurchaseAnalysis cpa

JOIN DimTable.Customer c

ON cpa.CustomerNo = c.CustomerNo

WHERE cpa.CustomerNo = 'C00000008'

WINDOW SalesWindow AS (

PARTITION BY cpa.PurchaseYear

ORDER BY cpa.PurchaseYear ASC,cpa.PurchaseMonth ASC

)

GO

As usual we use our CTE code structure to set things up. No changes to the CTE syntax of course. What we want to do is calculate the total sales amount by year, month, and customer. We use the SUM() function in the CTE so we need to include a GROUP BY clause.

The query refers to the CTE and calculates the average of all the sums. This time the OVER() clause simply refers to a name of the block that defines the PARTITION and ORDER BY clauses. This is called SalesWindow and is declared at the end of the query.

The new WINDOW command is used followed by the name SalesWindow and then between parenthesis the PARTITION BY and ORDER BY clauses are declared:

WINDOW SalesWindow AS (

PARTITION BY cpa.PurchaseYear

ORDER BY cpa.PurchaseYear ASC,cpa.PurchaseMonth ASC

)

Executing the query, we are presented with the results below.

Please refer to figure 2.22 below:

Graphical user interface, application, table, Excel

Description automatically generated

***Figure 2.22 – Query Results***

This query generated 36 rows. Large enough to evaluate the results. No surprises here. Works as expected.

The rolling averages seem to work. Let’s generate a query plan and see if there are any suggestions for indexes or more importantly if this new feature generated more steps or less steps in the query plan.

Please refer to figure 2.23 below:

Graphical user interface, application

Description automatically generated

***Figure 2.23 – Pre-index Query Plan***

As expected, since there are no indexes, we see a nice expensive table scan with a cost of 98% of the total execution time. We also see aHash Match task with a small cost of 1%. There is a suggestion for an index. Right click on it and extract the index to a new query window in the usual manner.

Here is what we get. Please refer to listing 2.12 below:

***Listing 2.12 – Suggested Index***

/\*

Missing Index Details from SQLQuery2.sql - DESKTOP-CEBK38L.APSales (DESKTOP-CEBK38L\Angelo (66))

The Query Processor estimates that implementing the following index could improve the query cost by 99.0667%.

\*/

/\*

USE [APSales]

GO

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

ON [StagingTable].[SalesTransaction] ([CustomerNo])

INCLUDE ([CalendarDate],[TransactionQuantity],[UnitRetailPrice])

GO

\*/

DROP INDEX IF EXISTS [CustomerNoieDateQuantityRetailPrice]

ON [StagingTable].[SalesTransaction]

GO

CREATE NONCLUSTERED INDEX [CustomerNoieDateQuantityRetailPrice]

ON [StagingTable].[SalesTransaction] ([CustomerNo])

INCLUDE ([CalendarDate],[TransactionQuantity],[UnitRetailPrice])

GO

A little copy and paste action so we can give the index a name. I left the square brackets alone as they were generated by SQL Server (and I am too lazy to remove them). Let’s create the index and rerun statistics and generate a new query plan.

Please refer to figure 2.24 below:

Graphical user interface, text, application

Description automatically generated

***Figure 2.24 – Post Index Query Plan***

We see a bunch of steps but at least there is an Index Seek now on the SalesTransaction table. This task costs 33% of total execution count. There’s a small compute scalar task at 1% and a hash match at a cost of 52%. Finally, a sort task that costs 10% (of the total execution time).

There is also a Window Spool task with a cost of 0% which is important. This task is associated with the OVER() clause and ROW and RANGE clauses as it loads the required rows in memory so they can be retrieved as often as required. If we do not have enough memory, then the rows need to be stored in temporary storage which could slow things down considerably. This task is one you should always look out for and see what the cost is!

As usual I am ignoring the tasks with 0% cost (except for the Window Spool) to keep things simple but on your own, look them up in the Microsoft documentation so you know what they do.

This is important, remember what window spool does:

***This task is associated with the OVER() clause and ROW and RANGE clauses as it loads the required rows in memory so they can be retrieved as often as required. If we do not have enough memory, then the rows need to be stored in temporary storage which could slow things down considerably. This task is one you should always look out for and see what the cost is!***

Seems like the Index Seek eliminated the table scan but several new steps are introduced.

Let’s compare the IO statistics pre and post index creation to see what improvements there are.

Please refer to table 2.6 below:

| **SQL Server Parse & Compile Time** | **Existing Index** | **New Index** |
| --- | --- | --- |
| CPU Time (ms) | **31** | **0** |
| elapsed time (ms) | **290** | **49** |
|  |  |  |
| **Statistic (work table)** | **Existing Index** | **New Index** |
| Scan Count | **39** | **39** |
| Logical Read | **217** | **217** |
| 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 (SalesTransaction)** | **Existing Index** | **New Index** |
| Scan Count | **5** | **1** |
| Logical Read | **17244** | **42** |
| Physical Reads | 0 | 1 |
| read-ahead reads | 17230 | 10 |
|  |  |  |
| **Statistic (Customer)** | **Existing Index** | **New Index** |
| Scan Count | **1** | **1** |
| Logical Read | **648** | **18** |
| Physical Reads | 1 | 1 |
| read-ahead reads | 0 | 0 |
|  |  |  |
| **SQL Server Execution Times** | **Existing Index** | **New Index** |
| CPU Time (ms) | 156 | 0 |
| elapsed time (ms) | 1210 | 105 |

***Table 2.6 – Statistics IO comparison, before & after index***

Looking through the statistics there seems to be significant improvement. What we are looking for is how significant the improvements are. Does the new WINDOW feature cause execution to be more expensive, faster or the same?

Statistics for the work table and the work file remain the same while statistics for the SalesTransaction table have improved significantly. Our logical reads went from 17244 to 42 and read-ahead reads went down from 17230 to 10. This is great!

Customer table logical reads went from 648 to 18 and our CPU time went from 156 to 0. Lastly elapsed execution time went from 1210 to 105.

In conclusion, the good news is that the new WINDOW feature does not decrease performance. Improvements in the query plans are similar to the prior OVER() clause syntax.

Seems like the benefit is that we can place the OVER() clause logic in one place and call it from multiple columns eliminating redundant typing if the window frame specifications are the same.

There’s more. You can define more than one window and even have a window reference another window definition as illustrated by the partial code in listing 2.13 below:

***Listing 2.13 – Defining Multiple Windows***

SELECT

cpa.PurchaseYear,

cpa.PurchaseMonth,

c.CustomerNo,

c.CustomerFullName,

cpa.TotalSalesAmount,

AVG(cpa.TotalSalesAmount) OVER AvgSalesWindow AS AvgTotalSales,

STDEV(cpa.TotalSalesAmount) OVER StdevSalesWindow AS StdevTotalSales,

SUM(cpa.TotalSalesAmount) OVER SumSalesWindow AS SumTotalSales

FROM CustomerPurchaseAnalysis cpa

JOIN DimTable.Customer c

ON cpa.CustomerNo = c.CustomerNo

WHERE cpa.CustomerNo = 'C00000008'

WINDOW

StdevSalesWindow AS (AvgSalesWindow),

AvgSalesWindow AS (

PARTITION BY cpa.PurchaseYear

ORDER BY cpa.PurchaseYear ASC,cpa.PurchaseMonth ASC

),

SumSalesWindow AS (

);

GO

Interesting feature. Let’s wrap up the chapter. Feel free to go back and review some areas you might be fuzzy on.

**Summary**

Did we meet our objectives?

We introduced the APSales data warehouse and studied the conceptual model for this database.

We also became familiar with the tables, columns, and relationships between the tables by introducing some simple data dictionaries.

We’ve covered the window functions in the aggregate function category by creating examples that used the OVER() clause to define window frames for the data sets generated by the queries.

One important objective was that we needed to understand performance analysis and tuning associated with queries that use the window functions and OVER() clauses. This was accomplished by generating and studying query plans and IO statistics. This information was used to arrive at conclusions on how to improve performance.

We also identified the window spool statistics as an important value to monitor. Make sure you understand what this means.

Last but not least, we took a peek at the new WINDOW feature in SQL Server 2022 that shows a lot of promise in the areas of code readability and implementation. Our quick analysis showed that it did not degrade performance but also it did not improve performance over the old syntax, but further testing needs to be done with larger data sets.

By the way, now is a good time, if you have not done so already to download the example script for this chapter and practice what you just learned.

In our next chapter, chapter 3 we cover the analytical functions which are a very powerful set of tools used in data analysis and reporting. We will also start adding ROWS and RANGE clauses to the queries we discuss that override the default behavior of window frames when we do not specify them.