DAX Query Plans

USING AND UNDERSTANDING DAX QUERY PLANS

produced by

DAX Query Plans

Introduction to performance analysis and DAX optimizations using query plans

Author: Alberto Ferrari

Published: Version 1.0 Revision 2 – July 17, 2012

Contact: alberto.ferrari@sqlbi.com – www.sqlbi.com

Summary: This paper is an introduction to query optimization of DAX code through the usage of the DAX query plans. It uses the Contoso database, which you can download from here: http://sdrv.ms/131eTUK and the Tabular version of AdventureWorks, available on CodePlex.

Acknowledgments: I would like to thank the peer reviewers that helped me improving this document: Marco Russo, Chris Webb, Greg Galloway, Ashvini Sharma, Owen Graupman and all our ssas-insiders friends.

I would also like to give a special thanks to T.K. Anand, Marius Dumitru, Cristian Petculescu, Jeffrey Wang, Ashvini Sharma, and Akshai Mirchandani who constantly answer to all of our fancy questions about SSAS.

Table of Contents

INTRODUCTION 4

LOOKING AT YOUR FIRST QUERY 5

- Clearing the Cache 8
- Iterators are not all that bad 10

FORMULA ENGINE AND STORAGE ENGINE 11

- Cache Usage 12
- FE, SE and CallBackDataID 13

EVENTS IN PROGRESS 16

- Optimizing Events in Progress, Step 1 18
- Optimizing Events in Progress, the Jedi way 21
- Optimizing Events in Progress, the Yoda Solution 26

Introduction

BI professionals always face the need to produce fast queries and measures. In order to obtain the best performance, a correct data model is needed but, once the model is in place, to further proceed with improvements, DAX optimization is the next step.

Optimizing DAX requires some knowledge of the xVelocity engine internals and the ability to correctly read and interpret a DAX query plan. In this paper we focus on very basic optimizations and we will guide you through the following topics:

- How to find the DAX query plan
- The difference between the logical and physical query plan
- A brief description of the difference between formula engine and storage engine
- Some first insights into the query plan operators

The goal of the paper is not that of showing complex optimization techniques. Rather, we focus on how to read different formulations of the same query understanding why they behave differently, by means of reading their query plans.

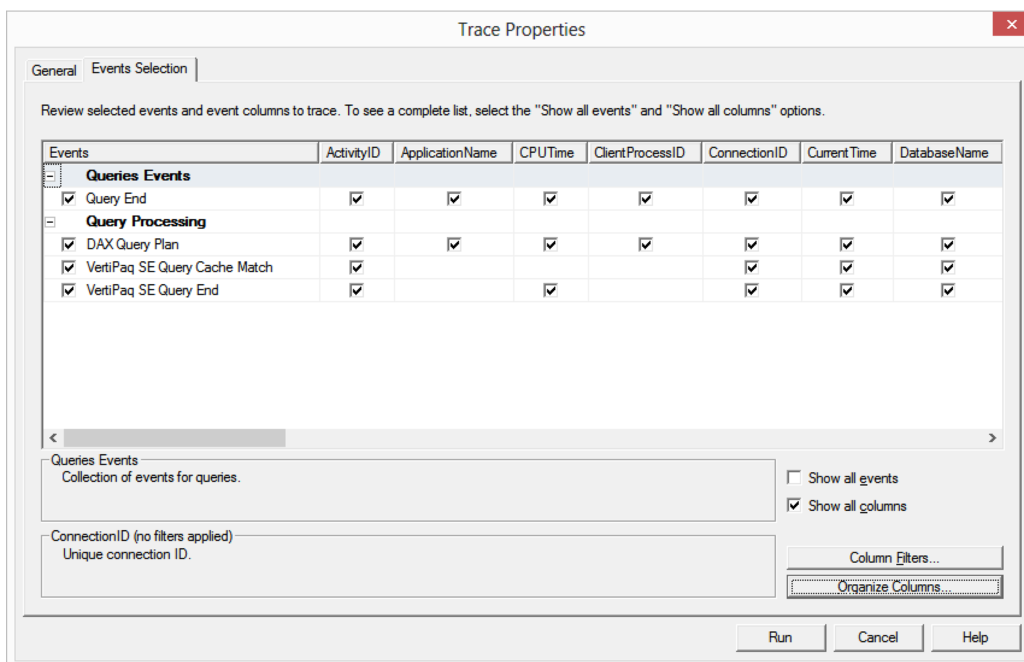
Looking at your first query

Understanding DAX query plans is a long process. We start with very simple queries and only when these basic concepts are clear enough, we will dive into the complexity of DAX expressions. Our first query is amazingly simple and it runs on the Contoso database:

```
EVALUATE
    ROW (
        "Sales",
         SUM ( OnlineSales[SalesAmount] )
    )
```

This query returns the sum of sales for the entire table OnlineSales and, to check it, you can simply run it inside an MDX query window in SSMS on the Contoso demo database. Let's use it to start learning how to read a query plan.

In order to catch the query plan, you need to use the SQL Server Profiler, run a new trace and configure it to grab the interesting events for a DAX query, like in the following picture:



You need to capture four events:

- **Query End**: this event is fired at the end of a query. You can take the Query Begin event too but I prefer to use the Query End, which includes the execution time.

- **DAX Query Plan**: this event is fired when the query engine has finished computing the query plan and contains a textual representation of the query plan. As you will learn, there are two different query plans, so you will always see two instances of this event for any DAX query. MDX queries, on

the other hand, might generate many plans for a single query and, in this case, you will see many DAX query plan for a single MDX query.

- **VertiPaq SE Query Cache Match**: this event occurs when a VertiPaq query is resolved by looking at the VertiPaq cache and it is very useful to see how much of your query performs a real computation and how much just does cache lookups.

- **VertiPaq SE Query End**: as with the Query End event, we prefer to grab the end event of the queries executed by the VertiPaq Storage Engine.

You will learn more about these events in the process of reading the profiler log of the query. Now, it is time to run the trace, execute the query and look at the result:

EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	0
VertiPaq SE Query End	0 - VertiPaq Scan	0	0
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	16	16

Even for such a simple query, SSAS logged five different events:

- One DAX VertiPaq Logical Plan event, which is the logical query plan. It represents the execution tree of the query and is later converted into a physical query plan that shows the actual query execution algorithm.

- Two VertiPaq scan events, i.e. queries executed by the VertiPaq engine to retrieve the result of your query.

- One DAX VertiPaq Physical Plan event. It represents the real execution plan carried on by the engine to compute the result. It is very different from the logical query plan and it makes use of different operators. From the optimization point of view, it is the most important part of the trace to read and understand and, as you will see, it is probably the most complex of all events.

- A final Query End event, which returns the CPU time and query duration of the complete query.

All of the events show both CPU time and duration, expressed in milliseconds. CPU time is the amount of CPU time used to answer the query, whereas duration is the time the user waited for getting the result. Using many cores, duration is usually lower than CPU time, because xVelocity used CPU time from many cores to reduce the duration.

Let us look at the various events in more detail. Looking at the event text, you will notice that they are nearly unreadable because all of the table names are shown with a numeric identifier appended to them. This is because the query plan uses the table ID and not the table name. For example, the first event looks like this:

```
AddColumns: RelLogOp DependOnCols()() 0-0 RequiredCols(0)(''[Sales])
  Sum_Vertipaq: ScaLogOp DependOnCols()() Currency DominantValue=BLANK
Table='OnlineSales_936cc562-4bb8-46e0-8d5b-7cc9c9e8ce49' -BlankRow Aggregations(Sum)
         Scan_Vertipaq: RelLogOp DependOnCols()() 0-142
RequiredCols(134)('OnlineSales'[SalesAmount]) Table='OnlineSales_936cc562-4bb8-46e0-8d5b-
7cc9c9e8ce49' –BlankRow
         'OnlineSales'[SalesAmount]: ScaLogOp
DependOnCols(134)('OnlineSales'[SalesAmount]) Currency DominantValue=NONE
```

For the sake of clarity, we will use a shortened version of the plans (which we edited manually):

```
AddColumns: RelLogOp
    Sum_Vertipaq: ScaLogOp
        Scan_Vertipaq: RelLogOp
        'OnlineSales'[SalesAmount]: ScaLogOp
```

Query plans are represented as simple lines of text. Each line is an operator and the following lines, indented, represent the parameters of the operator. In the previous example, you can see that the outermost operator is AddColumns, which creates the one-row table with the Sales column. The Sales column is the sum of all sales and, in fact, its operator is a Sum_VertiPaq one. Sum_VertiPaq scans the OnlineSales table and sums the OnlineSales[SalesAmount] column.

The logical query plan shows what SSAS plans to do in order to compute the measure. Not surprisingly, it will scan the OnlineSales table summarizing the SalesAmount column using SUM. Clearly, more complex query plans will be harder to decode.

After the logical query plan, there are two VertiPaq queries that contain many numbers after each table name. We removed them, for clarity. This is the original query:

```
SET DC_KIND="DENSE";
SELECT
SUM([OnlineSales_936cc562-4bb8-46e0-8d5b-7cc9c9e8ce49].[SalesAmount]), COUNT()
FROM [OnlineSales_936cc562-4bb8-46e0-8d5b-7cc9c9e8ce49];
```

While this is the cleaned version:

```
SET DC_KIND="DENSE";
SELECT
    SUM ( [OnlineSales].[SalesAmount] ),
    COUNT()
FROM
    [OnlineSales];
```

And this is the second VertiPaq query:

```
SET DC_KIND="AUTO";
SELECT
    SUM ( [OnlineSales].[SalesAmount])
FROM
    [OnlineSales];
```

The two queries are almost identical and they differ for the Event subclass. Event subclass 0, i.e. VertiPaq Scan, is the query as the SSAS engine originally requested it; event subclass 10, i.e. VertiPaq Scan Internal, is the same query, rewritten by the VertiPaq engine for optimization. The two query are – in reality – a single VertiPaq operation for which two different events are logged. The two queries are always identical, apart from a few (very rare) cases where the VertiPaq engine rewrites the query in a slightly different way.

VertiPaq queries are shown using a pseudo-SQL code that makes them easy to understand. In fact, by reading them it is clear that they compute the sum of the SalesAmount column from the OnlineSales table.

After these two queries, there is another query plan:

```
AddColumns: IterPhyOp
    SingletonTable: IterPhyOp
    Spool: LookupPhyOp
        AggregationSpool<Cache>: SpoolPhyOp
            VertipaqResult: IterPhyOp
```

The physical query plan has a similar format as the logical one: each line is an operator and its parameters are in subsequent lines, properly indented with one tab. Apart from this aesthetic similarity, the two query plans use completely different operators.

The first operator, AddColumns, builds the result table. Its first parameter is a **SingletonTable**, i.e. an operator that returns a single row table, generated by the ROW function. The second parameter **Spool** searches for a value in the data cached by previous queries. This is the most intricate part of DAX query plans. In fact, the physical query plan shows that it uses some data previously spooled by other queries, but it misses to show from which one.

As human beings, we can easily understand that the spooled value is the sum of SalesAmount previously computed by a VertiPaq query. Therefore, we are able to mentally generate the complete plan: first a query is executed to gather the sum of sales amount, its result is put in a temporary area from where it is grabbed by the physical query plan and assembled in a one-row table, which is the final result of the query.

Unluckily, in plans that are more complex this association tend to be much harder and it will result in a complex process, which you need to complete to get a sense out of the plan.

Both the logical and physical query plan are useful to grab the algorithm beneath a DAX expression. For simple expressions, the physical plan is more informative. On the other hand, when the expression becomes complex, looking at the logical query plan gives a quick idea of the algorithm and will guide you through a better understanding of the physical plan.

The final event visible in the profiler is the Query End event, which is basically useful to look at the total number of milliseconds needed to run the query (i.e. the duration). In our example, the sum was computed in 16 milliseconds.

Clearing the Cache

If, at this point, you run the query once again, your profiler is still catching data and it will look like this:

EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	0
VertiPaq SE Query End	0 - VertiPaq Scan	0	0
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	16	16
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query Cache Match	0 - VertiPaq Cache exact match		
VertiPaq SE Query End	0 - VertiPaq Scan	0	0
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	0	0

In the figure you can see both queries, one after the other. The second one took 0 milliseconds to execute and this is because the first VertiPaq query has been found in the cache. In fact, instead of a VertiPaq Scan internal, you see a VertiPaq Cache exact match, meaning that the query has not been executed: its result was in the VertiPaq cache and no computation has been necessary.

Whenever you optimize DAX, you always need to clear the database cache before executing a query. Otherwise all the timings will take the cache into account and your optimization will follow incorrect measurements.

In order to clear the cache you can use this XMLA command, either in an XMLA query window in SSMS or in an MDX query window, as we shown below:

```
<Batch xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">
    <ClearCache>
        <Object>
            <DatabaseID>Contoso</DatabaseID>
        </Object>
    </ClearCache>
</Batch>
```

You can conveniently put the clear cache command right before your query, separating them with a GO statement, like in the following example:

```
<Batch xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">
    <ClearCache>
        <Object>
            <DatabaseID>Contoso</DatabaseID>
        </Object>
    </ClearCache>
</Batch>

GO

EVALUATE
    ROW (
        "Sales",
         SUM ( OnlineSales[SalesAmount] )
    )
```

When you will be a guru of optimizations, it will probably be useful to run the queries with and without the cache, in order to understand what usage your DAX code is doing of the cache. As of now, it is better to focus on cold cache optimizations, which require less attention.

Iterators are not all that bad

Before we move on with more complex query plans, it is useful to look at the same query expressed with an iterator. Even if you probably learned that iterators do what their name suggest, i.e. they iterate the result of a table, in reality the optimizer makes a great work in trying to remove the iteration from the query and take advantage of a more optimized plan.

Let us profile, as an example, this query:

```
EVALUATE
    ROW (
        "Sales",
         SUMX ( OnlineSales, OnlineSales[SalesAmount] )
    )
```

Looking at the query plan, you will discover that it is identical to the one expressed by SUM. The optimizer detected that the VertiPaq engine can execute directly the operation inside the iteration, so it run the query using the same plan as of standard SUM.

This kind of optimization not only happens when you use SUMX to aggregate a column, as in this case, but also in many cases when the expression can be safely computed using a pseudo-SQL query. For example, simple multiplications and most math expressions are resolved in VertiPaq queries. Look at this query plan:

```
EVALUATE
    ROW (
        "Sales",
         SUMX ( OnlineSales, OnlineSales[UnitPrice] * OnlineSales[SalesQuantity])
    )
```

SSAS solves it with this VertiPaq query:

```
SET DC_KIND="AUTO";
SELECT
SUM ( (PFCAST( [OnlineSales].[UnitPrice] AS INT ) * PFCAST ( [OnlineSales].[SalesQuantity] AS INT)))
FROM [OnlineSales];
```

The query runs at the same speed as a regular SUM, because the engine executes the calculation during the table scanning of the OnlineSales table.

Formula Engine and Storage Engine

In the first section, we introduced how to grab and read a DAX query plan. Before diving into more complex topics, it is now time to introduce the two engines that work inside DAX: the formula engine (FE) and the storage engine (SE). Whenever a query needs to be resolved, the two engines work together in order to compute the result.

- **Formula Engine** is able to compute complex expressions (virtually any DAX function) but, because of its power, has one strong limitation: it is single threaded.

- **Storage Engine** is much simpler: it is able to perform simple mathematical operations on numbers, follow relationships for joins and retrieve data from memory while applying filters. Because of its simplicity, it is a highly efficient multi-threaded engine that is able to scale over many cores.

When tuning performance of a DAX expression, one of the main goals, if not the primary one, is to write the code to maximize the usage of SE and consequently reduce the amount of work taken by FE.

Roughly speaking, VertiPaq queries are executed by the SE, whereas the DAX query plan is the part of the query that is executed by the FE. If you look again at the queries of previous sections, most of the computation effort was undertaken by SE. In fact, the sum of sales amount was entirely computed by a Vertipaq SE Query (by the way, now you know why it is called VertiPaq **SE** Query), whereas all what FE had to do was gathering the final result of the query and assemble it in a single row table. In fact, the query plan of those queries was a perfect one.

In order to better understand the interaction between formula engine and storage engine, now we use a more complex query, where the formula engine needs to carry on more work.

```
EVALUATE
    ADDCOLUMNS (
        VALUES ( Product[ColorName] ),
        "Sales",
        CALCULATE ( SUM ( OnlineSales[SalesAmount] ) )
    )
```

This query is resolved by following two VertiPaq SE queries and query plan.

The first VertiPaq query retrieves the product color and the sum of sales from the OnlineSales table:

```
SELECT
    Product.ColorName,
    SUM ( OnlineSales.SalesAmount ),
    COUNT ()
FROM
    OnlineSales
    LEFT OUTER JOIN Product ON OnlineSales.ProductKey=Product.ProductKey
```

The second VertiPaq query returns the list of colors from the Product table. It is useful to note that the two lists of colors can be different, in case a color exists for some products that were never sold online.

```
SELECT
    Product.ColorName,
    COUNT ()
FROM
    Product
```

Finally, you see the query plan that, as we already know at this point, relies on temporary results returned by the previous queries:

```
AddColumns: IterPhyOp IterCols(0, 1)('Product'[ColorName], ''[Sales])
    Spool_Iterator<Spool>: IterPhyOp IterCols(0)('Product'[ColorName]) #Records=16
        AggregationSpool<Cache>: SpoolPhyOp #Records=16
            VertipaqResult: IterPhyOp
    Spool: LookupPhyOp LookupCols(0)('Product'[ColorName]) Currency #Records=16
        AggregationSpool<Cache>: SpoolPhyOp #Records=16
            VertipaqResult: IterPhyOp
```

This query plan scans a table containing 16 color names (first Spool_Iterator). Then it adds a column coming from the lookup of the color name in another table, which contains 16 rows composed by a color name and a currency value. Your task is to understand which of the VertiPaq queries returned those tables, so to give to the query plan its complete shape.

The first table is the result of the second VertiPaq query, i.e. the list of colors retrieved from the Product table, whereas the second table used for the lookup is the result of the first VertiPaq query, which returned color names and total of sales for each color. It is the FE and not the SE that performs the final join between the two queries.

This is a very good plan, because the FE is only working on small tables (16 rows each) and, of course, it is going to be very fast, even if single-threaded. The vast majority of the work (scanning the fact table and grouping by product color, following the relationship) is in charge of SE.

Cache Usage

When evaluating what SE and FE execute, remember that the two engines work in a different way regarding cache usage. SSAS caches only the results of VertiPaq queries, not the result of DAX calculations. Any task executed by SE goes in cache and produce faster results in following identical queries, whereas any job executed by FE will repeat the computation again.

If your query has a relevant portion executed by FE, this part is executed repeatedly, every time you query the measure and, if the time spent on FE is predominant, you do not benefit too much from the VertiPaq cache.

It is worth to note that MDX queries still have a calculation cache available to them. The result of an MDX calculation is stored in cache, whereas the result of a DAX FE calculation is not. Thus, regarding cache usage, MDX queries behave slightly better than DAX ones. Nevertheless, generally speaking, using DAX you have a better control over the algorithm used to resolve the query.

FE, SE and CallBackDataID

The storage engine executes simple calculations directly and the formula engine executes the more complex ones, like complex joins and iterations. SE scans tables and either returns a result or spools the resulting table for further execution, whereas FE executes iteration over the data returned by SE.

There is also a mixed scenario, which DAX often uses when the SE has to execute some non-trivial calculations during a table scan, but SE cannot handle them because of their complexity. In such a case, SE has the option to call back the FE in order to compute complex expressions during the table scan.

A special SE operator called CallBackDataID performs this interaction between FE and SE. Consider the following query:

```
EVALUATE
    ADDCOLUMNS (
        VALUES ( Product[ColorName] ),
        "Sales",
        CALCULATE (
            SUMX (
                OnlineSales,
                IF (
                    OnlineSales[SalesAmount] > 10,
                    OnlineSales[SalesAmount]
                )
            )
        )
    )
```

This query is very similar to the one you used at the beginning of this section. The main difference is that, instead of summing all of the sales from OnlineSales, you only sum the ones that are greater than 10 USD. In order to do that, you use SUMX iterating over all the rows and getting rid of the unwanted ones.

The IF inside SUMX is an issue, because SE is not able to evaluate IF conditions. In such a scenario, DAX has two options:

- It scans the OnlineSales[SalesAmount] column using a VertiPaq query and then processes the IF inside FE. This requires the spooling of the VertiPaq query result and, as such, requires memory.

- It scans the OnlineSales[SalesAmount] column inside SE and, during the iteration, SE asks FE to evaluate the IF on a row-by-row basis. SE invokes FE for each row, but the query memory requirements is much lower.

If you look at the query plan, you will see this VertiPaq query:

```
SELECT
    Product.ColorName,
    SUM (
        [CallbackDataID (
            IF (
                OnlineSales[SalesAmount]] > 10,
                OnlineSales[SalesAmount]]
            )
        ) ]
        ( PFDATAID ( OnlineSales.[SalesAmount] ) )
    )
FROM
    OnlineSales
    LEFT OUTER JOIN Product ON OnlineSales.ProductKey = Product.ProductKey
```

The query shows a CallBackDataID call. This means that, during the table scan, prior to summing values the Storage Engine invokes Formula Engine for each row, passing to it the expression to evaluate (which is our IF statement) and the value of the SalesAmount column for the current row.

One of the good things about CallBackDataID is that FE is involved in the calculation, but only as part of a more complex SE process. Because SE is multithreaded, one instance of FE is called for each thread of SE, processing the query in a multithreaded environment.

Thus, with CallBackDataID you get the best of the two worlds: the richness of FE and the speed of SE. CallBackDataID is not as fast as a pure VertiPaq query, but it is much faster than a pure FE query.

The only big drawback of CallBackDataID is that the cache does not store its result, even if computed by SE. Thus, if your query makes a heavy usage of the mixed environment, it will not benefit much from the cache. This might improve in future releases of SSAS but, as of now, cache usage is a limitation you need to keep in mind.

It is useful to note, at this point, that you can be express the previous query in a much more efficient way using this syntax:

```
EVALUATE
    ADDCOLUMNS (
        VALUES ( Product[ColorName] ),
        "Sales",
        CALCULATE (
            SUM ( OnlineSales[SalesAmount] ),
            OnlineSales[SalesAmount] > 10
        )
    )
```

Looking at the query plan of this DAX code, you will note that there is no CallBackDataID and the query is resolved by first looking at the values of SalesAmount that are greater than 10, and then summing the values from the fact table only for the relevant ones. This final query, in fact, is completely resolved inside SE, so it is faster and takes full advantage from the DAX cache system, resulting in optimal performance.

The first VertiPaq query computes the values of SalesAmount that are greater than 10:

```
SELECT
    OnlineSales.SalesAmount,
    COUNT ()
FROM
    OnlineSales
WHERE
  COALESCE( PFCAST( OnlineSales.SalesAmount AS INT)) > COALESCE( [CallbackDataID(10)]())
```

Once VertiPaq knows which values of SalesAmount are greater than 10, it can use the information in the following query, which computes the SUM of SalesAmount already grouped by color:

```
SELECT
    Product.ColorName,
    SUM ( OnlineSales.SalesAmount),
    COUNT ()
FROM OnlineSales
  LEFT OUTER JOIN Product ON
        OnlineSales.ProductKey = Product.ProductKey
WHERE
  OnlineSales.SalesAmount INB (103600, 399680, 239920, 2079920, 3192000, 1664000,
    3040000, 256000, 2400000, 333840...[2359 total values, not all displayed])
```

The final step is gathering the different colors to perform the final JOIN in FE:

```
SELECT
    Product.ColorName,
    COUNT()
FROM
    Product
```

You can observe the final JOIN in FE looking at the physical query plan:

```
AddColumns: IterPhyOp ('Product'[ColorName], ''[Sales])
    Spool_Iterator<Spool>: IterPhyOp ('Product'[ColorName]) #Records=16
        AggregationSpool<Cache>: SpoolPhyOp #Records=16
            VertipaqResult: IterPhyOp
    Spool: LookupPhyOp ('Product'[ColorName]) Currency #Records=15
        AggregationSpool<Cache>: SpoolPhyOp #Records=15
            VertipaqResult: IterPhyOp
```

Events in Progress

In order to make some practice with the analysis of DAX query plans, we use the "event in progress" scenario, looking at different DAX queries to solve it. The "event in progress" scenario applies to any business that handles "events", where an event is something that happens at a certain point in time and has a duration. It is aimed to count the number of events that are active during a specific period of time. For example, if the event is an order, it happens as soon as the order is placed and lasts until it is shipped. The question is: how many orders are active in a specific point in time? For an order to be active, today, it need to have been placed before yesterday and not have been shipped before today. We use the AdventureWorks Tabular sample database for this demo.

A first query that solves this problem, simple both to write and understand, is the following:

```
EVALUATE
    ADDCOLUMNS (
        VALUES ( 'Date'[Date] ),
        "OpenOrders",
        COUNTROWS (
             FILTER (
                 'Internet Sales',
                 'Internet Sales'[Order Date] < 'Date'[Date] &&
                 'Internet Sales'[Ship Date] > 'Date'[Date]
             )
        )
    )
```

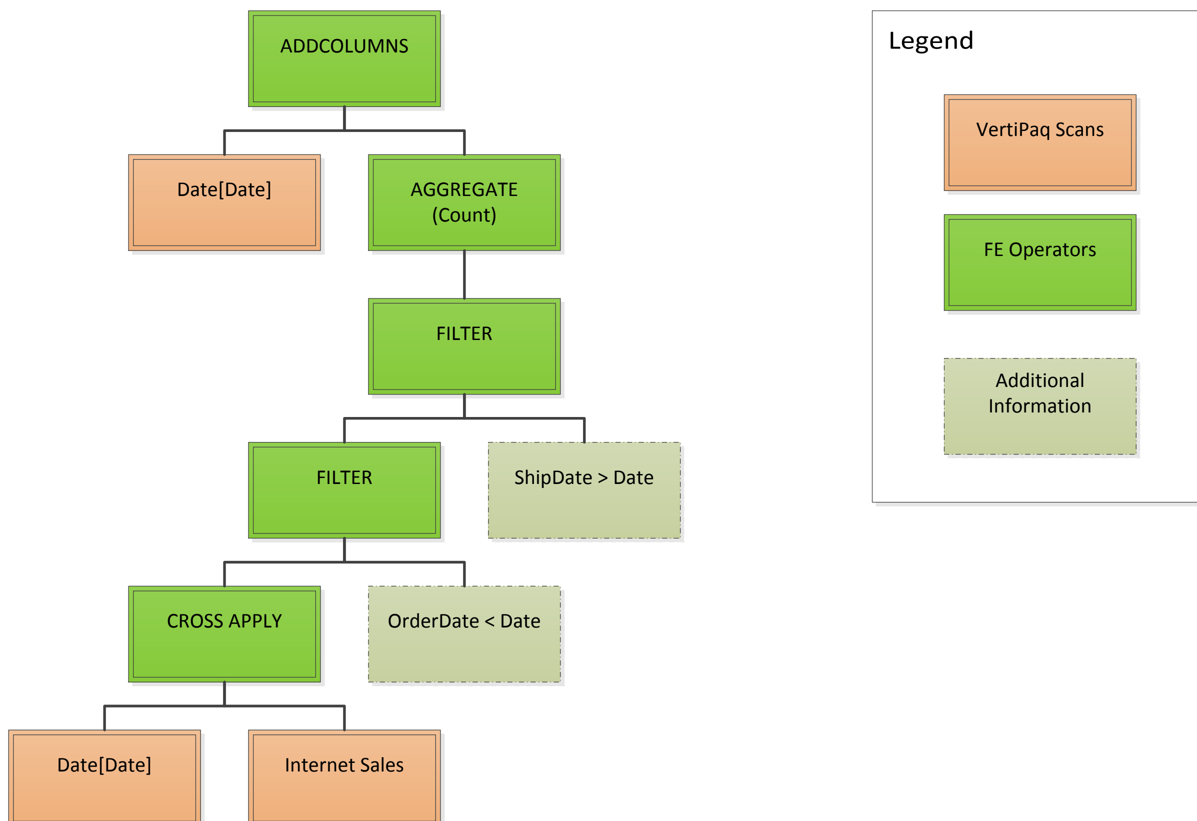
The query runs for 22 seconds before returning. This execution time, for AdventureWorks' size, looks definitely too long. In order to understand what is happening, let us look at the VertiPaq queries and the DAX query plan.

The query is resolved with two VertiPaq queries and a complex physical query plan. The VertiPaq queries are very simple: one gathers RowNumber, Order Date and ShipDate from the Internet Sales table, the other one returns the dates from the Date table. The issue is understanding the query plan, which is non-trivial:

```
1  AddColumns: IterPhyOp ('Date'[Date], ''[OpenOrders])
2     Spool_Iterator<Spool>: IterPhyOp ('Date'[Date]) #Records=2191
3         AggregationSpool<Cache>: SpoolPhyOp #Records=2191
4             VertipaqResult: IterPhyOp
5     Spool: LookupPhyOp LookupCols(0)('Date'[Date]) BigInt #Records=1132
6         AggregationSpool<Count>: SpoolPhyOp #Records=1132
7             Filter: IterPhyOp ('Date'[Date], [RowNumber], [Ship Date])
8                 Extend_Lookup: IterPhyOp ('Date'[Date], [Ship Date])
9                     Spool_Iterator<Spool>: IterPhyOp
10                     ('Date'[Date], [RowNumber], [Ship Date]) #Records=67373061
11                         AggregationSpool<GroupBy>: SpoolPhyOp #Records=67373061
12                            Filter: IterPhyOp ('Date'[Date], [RowNumber], [Order Date], [Ship Date])
13                                 Extend_Lookup: IterPhyOp ('Date'[Date], [Order Date])
14                                     CrossApply: IterPhyOp ('Date'[Date], [Order Date])
15                                         Spool_Iterator<Spool>: IterPhyOp ('Date'[Date]) #Records=2191
16                                             AggregationSpool<Cache>: SpoolPhyOp #Records=2191
17                                                 VertipaqResult: IterPhyOp
18                                         Spool_Iterator<Spool>: IterPhyOp
19                                         ([RowNumber], [Order Date], [Ship Date]) #Records=60398
20                                             AggregationSpool<Cache>: SpoolPhyOp #Records=60398
21                                                 VertipaqResult: IterPhyOp
22                                     LessThan: LookupPhyOp ('Date'[Date], [Order Date])
23                                         [Order Date]: LookupPhyOp ([Order Date])
24                                         'Date'[Date]: LookupPhyOp ('Date'[Date])
25                     GreaterThan: LookupPhyOp ('Date'[Date], [Ship Date])
26                         [Ship Date]: LookupPhyOp ([Ship Date])
27                          'Date'[Date]: LookupPhyOp ('Date'[Date])
```

If you look at rows 17 and 21, you will notice two VertiPaqResults used, i.e. data coming from VertiPaq SE queries subsequently used by the FE to calculate the expression. Row 17 reads 2191 rows, which are the dates of the calendar table, whereas row 21 reads 60938 rows with the columns RowNumber, OrderDate and ShipDate from the Internet Sales table. The row 4 uses the same set of dates used at row 17, in order to produce the result set.

Looking at the query plan in textual form is hard. We produced a nicer chart showing the query plan in a graphical way:



At the bottom of the diagram, you can see the VertiPaq queries (in red). The first operation is a CROSS APPLY of the dates and the internet sales. This cross apply produces a high number of rows (namely, the Cartesian product between dates and sales), which are filtered by the first condition (lower FILTER) and by the second condition (upper FILTER). Finally, a COUNT operation aggregates the result of the filtered cross-join. The result set contains the date and the count of rows that survived the filter. The final ADDCOLUMNS produces the result shown to the user.

The way DAX answers to the query is somewhat different from the original expression. There is no iteration and two different operators handle the two conditions in the FILTER expressions, even if they work on the same flow of data. By looking at the query plan, it seems that the huge number of rows generated for the cross join is responsible for the poor performance of the query.

Now, which parts of the query plan do the formula engine and the storage engine execute, respectively? In this case, the storage engine is not really making a lot of work. Its only task is materializing part of the fact table (namely the pair of OrderDate and ShipDate) and the values of the date column in the date table. The formula engine carries on the remaining part the work, starting from the cross join, the filtering and all the other calculations.

Optimizing Events in Progress, Step 1

We can try a different formulation of the same query. This time, we use CALCULATE and we express the filtering as two different FILTER, one for each column, relying on the fact that the CALCULATE semantics handles automatically the AND between them.

```
EVALUATE
    ADDCOLUMNS (
        VALUES ( 'Date'[Date] ),
        "OpenOrders",
        CALCULATE (
            COUNTROWS ('Internet Sales' ),
            FILTER ( 'Internet Sales', 'Internet Sales'[Order Date] < 'Date'[Date] ),
            FILTER ( 'Internet Sales', 'Internet Sales'[Ship Date] > 'Date'[Date] )
        )
    )
```

This query produces a completely different query plan and performs much better, because it returns the same data in less than one second. The query is resolved with four VertiPaq queries and a slightly complex DAX query plan.

The first VertiPaq query returns the values of the OrderDate column from the fact table:

```
SELECT
    [Internet Sales].OrderDate,
    COUNT()
FROM
    [Internet Sales]
```

The second VertiPaq query returns the values of the Date key:

```
SELECT
    Date.FullDateAlternateKey,
    COUNT ()
FROM
    Date
```

Finally, the third VertiPaq query returns the values of the ShipDate column:

```
SELECT
    [Internet Sales].ShipDate,
    COUNT()
FROM
    [Internet Sales]
```

The plan optimization phase executes all of the queries above. There is a fourth query executed during the run of the DAX plan:

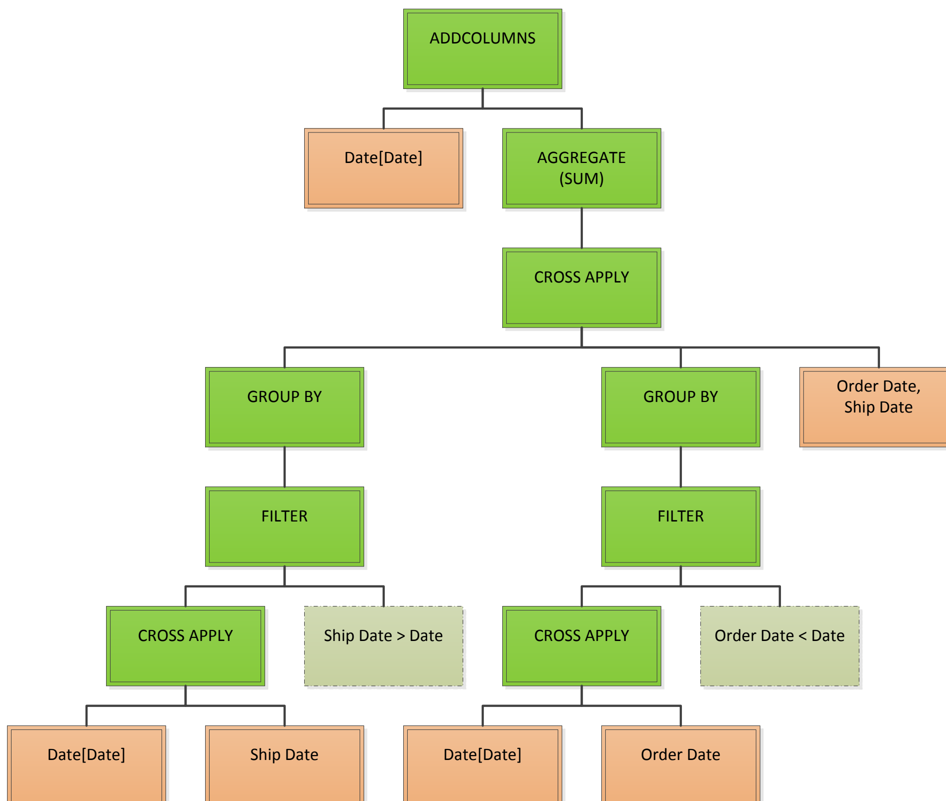
```
SELECT
    [Internet Sales].[OrderDate],
    [Internet Sales].[ShipDate],
    COUNT()
FROM
    [Internet Sales]
WHERE
  [Internet Sales].[OrderDate] IN
        (38913.000000, 39634.000000, 39178.000000, 39597.000000, 39141.000000,
         39560.000000, 38685.000000, 39104.000000, 38648.000000, 39067.000000...
         [1124 total values, not all displayed])
    VAND
  [Internet Sales].[ShipDate] IN
        (38920.000000, 39641.000000, 39185.000000, 39604.000000, 39148.000000,
         39567.000000, 38692.000000, 39111.000000, 38655.000000, 39074.000000...
         [1124 total values, not all displayed])
```

This is the first time we see a VertiPaq query used during the query execution and not during query optimization. This query returns the pairs of OrderDate and ShipDate with a condition that filters only the existing pairs.

The DAX query plan is non-trivial, even in its simplified version shown here:

```
AddColumns: IterPhyOp ([Date], ''[OpenOrders])
    Spool_Iterator<Spool>: IterPhyOp ([Date]) #Records=2191
        AggregationSpool<Cache>: SpoolPhyOp #Records=2191
            VertipaqResult: IterPhyOp
    Spool: LookupPhyOp ([Date]) #Records=1132
        AggregationSpool<Sum>: SpoolPhyOp #Records=1132
            CrossApply: IterPhyOp ([Date])
                Spool_UniqueHashLookup: IterPhyOp ([Date], [Ship Date]) #Records=845580
                    AggregationSpool<GroupBy>: SpoolPhyOp #Records=845580
                        Filter: IterPhyOp ([Date], [Ship Date])
                            Extend_Lookup: IterPhyOp ([Date], [Ship Date])
                                CrossApply: IterPhyOp ([Date], [Ship Date])
                                    Spool_Iterator<Spool>: IterPhyOp ([Date]) #Records=2191
                                        AggregationSpool<Cache>: SpoolPhyOp #Records=2191
                                            VertipaqResult: IterPhyOp
                                    Spool_Iterator<Spool>: IterPhyOp ([Ship Date]) #Records=1124
                                        AggregationSpool<Cache>: SpoolPhyOp #Records=1124
                                            VertipaqResult: IterPhyOp
                                GreaterThan: LookupPhyOp Boolean
                                    [Ship Date]: LookupPhyOp
                                    [Date]: LookupPhyOp
                Spool_MultiValuedHashLookup: IterPhyOp ([Order Date]) ([Date]) #Records=1623848
                    AggregationSpool<GroupBy>: SpoolPhyOp #Records=1623848
                        Filter: IterPhyOp ([Date], [Order Date])
                            Extend_Lookup: IterPhyOp ([Date], [Order Date])
                                CrossApply: IterPhyOp ([Date], [Order Date])
                                    Spool_Iterator<Spool>: IterPhyOp ([Date]) #Records=2191
                                        AggregationSpool<Cache>: SpoolPhyOp #Records=2191
                                            VertipaqResult: IterPhyOp
                                    Spool_Iterator<Spool>: IterPhyOp ([Order Date]) #Records=1124
                                        AggregationSpool<Cache>: SpoolPhyOp #Records=1124
                                            VertipaqResult: IterPhyOp
                                LessThan: LookupPhyOp Boolean
                                    [Order Date]: LookupPhyOp
                                    [Date]: LookupPhyOp
                Spool_Iterator<Spool>: IterPhyOp ([Order Date], [Ship Date]) #Records=1124
                    AggregationSpool<Cache>: SpoolPhyOp #Records=1124
                        VertipaqResult: IterPhyOp
```

In order to let this query plan be more readable, we removed all the table names (column names are enough to understand the plan, because they are unique) and many not-so-useful information. Nevertheless, in order to perform a good analysis of the plan, the graphical visualizations is more helpful.



It is not easy to digest this query plan, so take your time. Compare the graphical representation with the textual query plan and the following explanation. It helps to jump from one to the other during the analysis, in order to grab the exact meaning of the plan. Here is a simple explanation of what the plan does:

1. DAX builds the CROSSJOIN of Date and ShipDate, filtering out the rows where ShipDate is not greater than date. In order to do this operation, it uses the values of the Date and ShipDate columns gathered by the VertiPaq queries executed during the plan optimization phase.

2. It performs a similar operation with Date and OrderDate, generating the set of Date and OrderDate where OrderDate is less than Date.

3. Using another VertiPaq query, it gathers the set of existing OrderDate and ShipDate.

4. These sets are joined together and the result is the set of all the combination between Date, OrderDate, ShipDate where "OrderDate < Date AND ShipDate > Date", with the additional condition that OrderDate and ShipDate must appear together in a row of the fact table. It is worth to note that the same tuple can appear multiple times, once for each occurrence.

5. Finally, this set is grouped by Date and used by ADDCOLUMNS to join the values of the date with the count of (OrderDate, ShipDate) occurring in the previously computed set.

It is very interesting to note that there is no need to do a CROSSJOIN with the fact table. The CROSSJOIN happens only between single columns. Thus, the number of iterated rows is much lower than before. This makes it necessary to further filter the generated subsets with the set of existing (Order Date, Ship Date), in order to remove non-existent values. Nevertheless, the query plan is much better than the previous one. The execution time reflects this optimization and is 20 times faster now.

The good part of this query plan is the fact that, by decomposing the filtering condition in two simpler ones using the two filters of CALCULATE, we let the optimizer use a better algorithm. As it usually happens, using simpler formulas helps the DAX optimizer producing faster query plans.

Optimizing Events in Progress, the Jedi way

The previous query plan is already a very good one and we bet than any human being can already be very happy of learning and using it. That said, Chris Webb found a better solution for this scenario using a formulation that is not very intuitive to write, but it greatly outperforms the previous query. Thus, it is useful to understand how DAX solves it.

This is the DAX code:

```
EVALUATE
    ADDCOLUMNS (
        VALUES ( 'Date'[Date] ),
        "OpenOrders",
        COUNTROWS (
            FILTER(
                'Internet Sales',
                CONTAINS (
                    DATESBETWEEN (
                        'Date'[Date],
                        'Internet Sales'[Order Date],
                        'Internet Sales'[Ship Date] - 1
                    ),
                    'Date'[Date],
                    'Date'[Date]
                )
            )
        )
    )
```

The idea is slightly different from the previous ones. It iterates over the dates and, for each date, it counts the number of rows in the fact table where the currently iterated date is present in the range of dates starting with the order date and ending with the ship date.

The query structure is very similar to the original query, indeed. The difference is in the way it performs the test. Instead of computing two inequalities, the condition tests if the currently iterated date is present in a set of dates ranging from OrderDate to ShipDate.

At first sight, this query does not look promising, but the optimizer is going to shine on this code. In fact, this query outperforms the previous one, running in only 107 milliseconds, i.e. ten times faster than our already optimized version (200 times faster than the original one, by the way). In order to understand how it is possible, we need to dive into the query plan, as usual.

The DAX query plan is a complex one and requires some time to grab its way of working. Thus, it is better to look at the logical query plan before diving into the physical one. This is the logical query plan of the query:

```
AddColumns: RelLogOp RequiredCols([Date], ''[OpenOrders])
    Scan_Vertipaq: RelLogOp RequiredCols([Date]) Table='Date' +BlankRow
    CountRows: ScaLogOp BigInt DominantValue=BLANK
        Filter: RelLogOp RequiredCols([Date], [RowNumber], [Order Date], [Ship Date])
            Scan_Vertipaq: RelLogOp RequiredCols ([RowNumber], [Order Date], [Ship Date])
                Table='Internet Sales' -BlankRow
            Not: ScaLogOp Boolean DominantValue=FALSE
                ISBLANK: ScaLogOp Boolean DominantValue=TRUE
                    MinX: ScaLogOp BigInt DominantValue=BLANK
                        Filter: RelLogOp RequiredCols([Date], [Order Date], [Ship Date], [Date])
                            DatesBetween: RelLogOp RequiredCols([Order Date], [Ship Date], [Date])
                                [Order Date]: ScaLogOp DateTime DominantValue=NONE
                                Subtract: ScaLogOp DateTime DominantValue=NONE
                                    [Ship Date]: ScaLogOp DateTime DominantValue=NONE
                                    Constant: ScaLogOp BigInt DominantValue=1
                            Is: ScaLogOp Boolean DominantValue=FALSE
                                [Date]: ScaLogOp DateTime DominantValue=NONE
                                [Date]: ScaLogOp DateTime DominantValue=NONE
                        Constant: ScaLogOp BigInt DominantValue=1
```

The logical query plan is not very hard to read: for each row of the date table (first Scan_VertiPaq) it counts the rows of a filtered scan of the fact table that tests the condition.

The surprise is in the filter condition. If you look at the DAX code, the condition is CONTAINS, which checks if there is a match for a row in a table. There is no trace of CONTAINS in the logical query plan, because the optimizer transformed it into a more complex condition that looks like:

```
NOT ( ISBLANK ( MINX ( FILTER ( Table, Condition ), 1 ) ) )
```

The table is the result of DatesBetween and the condition is the check of the two dates, exactly what we asked with CONTAINS in the DAX code. This information is important because later, in the physical query plan, you will see the same pattern (i.e. MINX of a table with a constant value of 1), which makes the plan harder to read, if you do not remember that MINX (…, 1) is the translation of CONTAINS.

Sometimes looking at the physical query plan is intimidating, because of its complexity. Whenever you face a complex physical plan, it is always useful spending a few minutes in learning the first insights from the logical query plan and only later diving into the physical one. Remember that both plans are useful: the logical one is easier and the physical one is more complete. Do not waste your time studying a complex physical plan before having looked at the logical one, this simple tip will save you hours of work!

Another consideration is worth doing at this point. In the logical query plan, we left the information about the RequiredCols. RequiredCols is the set of columns that are required for an operator to work. It is important to note this, because the DatesBetween function depends on OrderDate and ShipDate only. If you count the number of distinct (OrderDate, ShipDate) in the fact table, you will discover that there are 1,124 combinations of these two dates out of 60,398 rows in the fact table. A smart engine, like the DAX optimizer, can benefit from this information and avoid computing the DatesBetween table for all the rows of the fact table, when it can perform this slow calculation for only 1,124 combinations and then use the result for the different rows, during iteration.

Now that we have a first insight at the plan, it is time to go deeper and study the physical plan.

There are three VertiPaq queries. The first gathers the date keys:

```
SELECT
    Date.FullDateAlternateKey,
    COUNT ()
FROM
    Date
```

The VertiPaq cache returns the result of the second query, which is identical to the first one. The third query returns the order date and ship date values from the fact table:

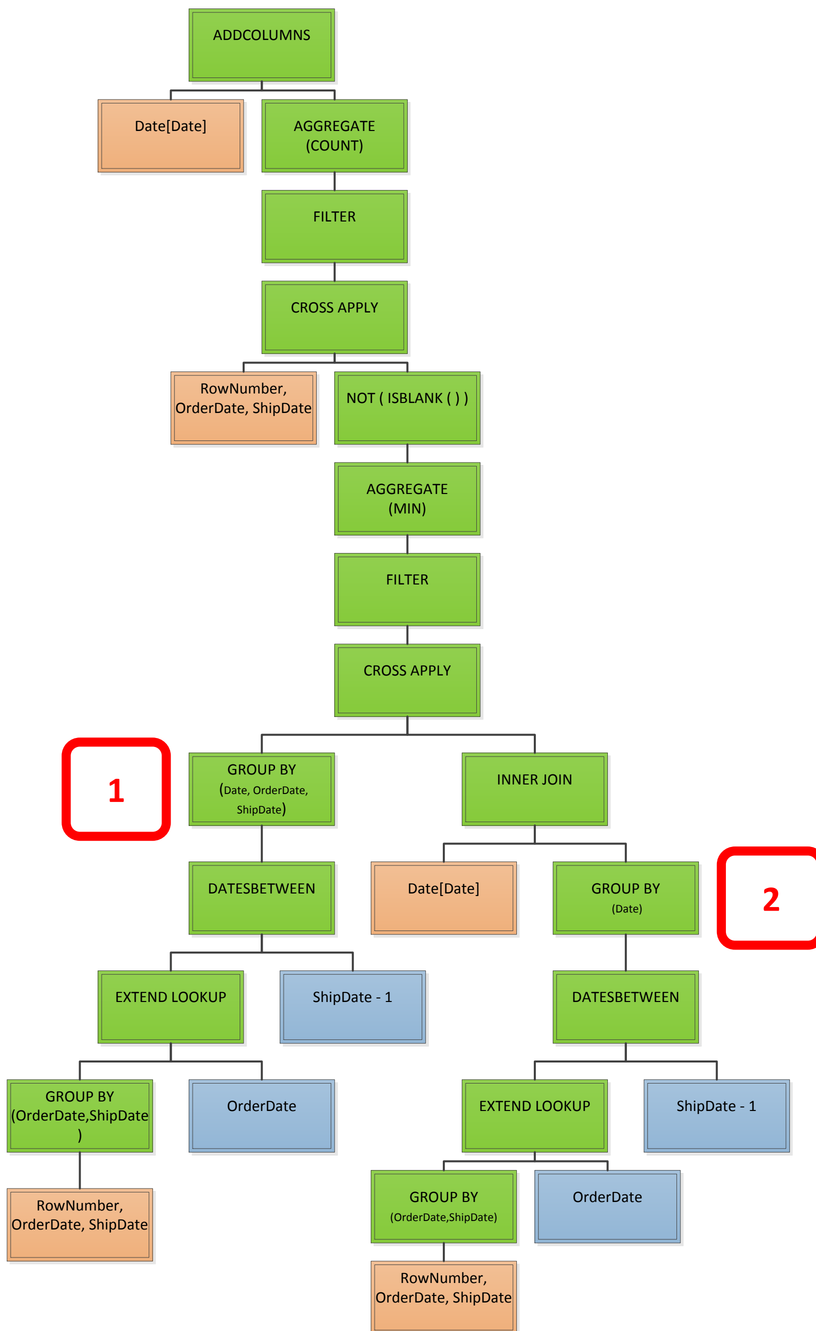
```
SELECT
    [Internet Sales].RowNumber,
    [Internet Sales].OrderDate,
    [Internet Sales].ShipDate,
    COUNT()
FROM
    [Internet Sales]
```

The VertiPaq engine does not perform further computations. It is clear, at this point, that the job performed by the Formula Engine is not trivial. Moreover, it is also evident that the Formula Engine evaluates the CONTAINS and the DATESBETWEEN functions. Thus, the speed comes from a very smart query plan or from some a powerful optimization inside these functions.

The query plan, as you can see here, is somewhat complex:

```
AddColumns: IterPhyOp ([Date], ''[OpenOrders])
    Spool_Iterator<Spool>: IterPhyOp ([Date]) #Records=2191
        AggregationSpool<Cache>: SpoolPhyOp #Records=2191
            VertipaqResult: IterPhyOp
    Spool: LookupPhyOp ([Date]) BigInt #Records=1133
        AggregationSpool<Count>: SpoolPhyOp #Records=1133
            Filter: IterPhyOp ([Date], [RowNumber], [Order Date], [Ship Date])
                CrossApply: IterPhyOp ([Date], [Order Date], [Ship Date])
                    Spool_MultiValuedHashLookup: IterPhyOp ([Order Date], [Ship Date]) ([RowNumber]) #Records=60398
                        AggregationSpool<Cache>: SpoolPhyOp #Records=60398
                            VertipaqResult: IterPhyOp
                    Not: IterPhyOp ([Date], [Order Date], [Ship Date])
                        ISBLANK: IterPhyOp ([Date], [Order Date], [Ship Date])
                            Spool_Iterator<Spool>: IterPhyOp ([Date], [Order Date], [Ship Date]) #Records=7868
                                AggregationSpool<Min>: SpoolPhyOp #Records=7868
                                    Extend_Lookup: IterPhyOp
                                        Filter: IterPhyOp ([Date], [Order Date], [Ship Date], [Date])
                                            CrossApply: IterPhyOp ([Date], [Date])
                                                Spool_MultiValuedHashLookup: IterPhyOp ([Date])
                                                ([Order Date], [Ship Date]) #Records=7868
                                                    AggregationSpool<GroupBy>: SpoolPhyOp #Records=7868
                                                        DatesBetween: IterPhyOp ([Order Date], [Ship Date], [Date])
                                                            Extend_Lookup: IterPhyOp ([Order Date])
                                                                Spool_Iterator<Spool>: IterPhyOp ([Order Date], [Ship Date])
                                                                #Records=1124
                                                                    AggregationSpool<GroupBy>: SpoolPhyOp #Records=1124
                                                                        Spool_Iterator<Spool>: IterPhyOp
                                                                        ([RowNumber], [Order Date], [Ship Date])
                                                                        #Records=60398
                                                                            AggregationSpool<Cache>: SpoolPhyOp #Records=60398
                                                                                VertipaqResult: IterPhyOp
                                                                [Order Date]: LookupPhyOp ([Order Date]) DateTime
                                                            Subtract: LookupPhyOp ([Ship Date]) DateTime
                                                                [Ship Date]: LookupPhyOp ([Ship Date]) DateTime
                                                                Constant: LookupPhyOp BigInt 1
                                                InnerHashJoin: IterPhyOp ([Date], [Date])
                                                    Extend_Lookup: IterPhyOp ([Date])
                                                        Spool_Iterator<Spool>: IterPhyOp ([Date]) #Records=2191
                                                            AggregationSpool<Cache>: SpoolPhyOp #Records=2191
                                                                VertipaqResult: IterPhyOp
                                                        [Date]: LookupPhyOp ([Date]) DateTime
                                                    HashLookup: IterPhyOp ([Date]) #Recs=1133
                                                        HashByValue: SpoolPhyOp #Records=1133
                                                            Extend_Lookup: IterPhyOp ([Date])
                                                                Spool_Iterator<Spool>: IterPhyOp ([Date]) #Records=1133
                                                                    AggregationSpool<GroupBy>: SpoolPhyOp #Records=1133
                                                                        DatesBetween: IterPhyOp
                                                                        ([Order Date], [Ship Date], [Date])
                                                                            Extend_Lookup: IterPhyOp ([Order Date])
                                                                                Spool_Iterator<Spool>: IterPhyOp
                                                                                ([Order Date], [Ship Date]) #Records=1124
                                                                                    AggregationSpool<GroupBy>: SpoolPhyOp
                                                                                    #Records=1124
                                                                                        Spool_Iterator<Spool>: IterPhyOp
                                                                                        ([RowNumber], [Order Date],
                                                                                        [Ship Date]) #Records=60398
                                                                                            AggregationSpool<Cache>:
                                                                                            SpoolPhyOp #Records=60398
                                                                                                VertipaqResult: IterPhyOp
                                                                                [Order Date]: LookupPhyOp ([Order Date])
                                                                            Subtract: LookupPhyOp ([Ship Date]) DateTime
                                                                                [Ship Date]: LookupPhyOp ([Ship Date])
                                                                                Constant: LookupPhyOp BigInt 1
                                                                [Date]: LookupPhyOp ([Date]) DateTime
                                        Constant: LookupPhyOp BigInt 1
```

As we did before, let us look at the query plan in a graphical way. Please note that we marked two sections of the plan with the numbers 1 and 2 both in the physical plan and in the graphical version, it order to simplify recognizing two points that need your attention.



This time, even the graphical representation of the plan is non-trivial and you need some time to study it with care. In order to grab its behavior, you need to start by not thinking at the original query. In fact, the DAX query contained an iteration and a CONTAIN operator that no longer appear in the physical plan.

Let us start with the two highlighted points:

1. This block starts by taking the Internet Sales table (three columns: RowNumber, OrderDate, ShipDate, as returned by one of the VertiPaq queries), then it makes a GROUP BY OrderDate and ShipDate. Once it has computed this set (i.e. the distinct values of OrderDate and ShipDate from the fact table), it feeds the data to DatesBetween, which generates the list of dates between OrderDate and ShipDate – 1. The resulting dataset contains (OrderDate, ShipDate, Date), for all the dates between OrderDate and ShipDate. The final “group by” operation does not change this set.

2. This block is nearly identical to (1). The only difference is the final group by date, instead of grouping by all the three columns. The final set produced by (2) is the set of dates that happen to be included, for some orders, between OrderDate and ShipDate. An INNERJOIN uses this set to filter the dates.

It might help to look at this DAX code, which shows the set computed by these two blocks, which are aggregated later by using different columns.

```
EVALUATE
    SUMMARIZE (
        GENERATE (
            SUMMARIZE (
                'Internet Sales',
                'Internet Sales'[Order Date],
                'Internet Sales'[Ship Date]
            ),
            DATESBETWEEN (
                'Date'[Date],
                'Internet Sales'[Order Date],
                'Internet Sales'[Ship Date] - 1
            )
        ),
        'Date'[Date]
    )
```

1 and 2 are used by a CROSS APPLY operator. This CROSS APPLY generates a set containing (Date, OrderDate and ShipDate). From a functional point of view, this set contains, for each date, all the couples of (OrderDate and ShipDate) in the fact table that embrace the date.

This set goes into the CONTAINS pattern, which computes the MIN (SUMX (…, 1)), as we have previously seen looking at the logical query plan. Due to the data distribution in the database, this pattern does not filter anything. You can see it by checking the row count before and after the FILTER: both show 7868 rows.

VertiPaq is going to use this set in a further CROSS APPLY with the (RowNumber, OrderDate, ShipDate) returned by a VertiPaq query. Doing this join, it is able to generate a new set containing, for each date, the RowNumber from the fact table of the orders whose OrderDate and ShipDate contain the date.

The final step is counting these rows and merging this number with the date table in order to produce the desired result.

Optimizing Events in Progress, the Yoda Solution

Now that you have spent so much time looking at the query plan of the Jedi solution to Event in Progress, you can re-write the same query using a pattern that makes the interpretation of the query plan much easier. Here is the code:

```
EVALUATE
    ADDCOLUMNS (
        VALUES ( 'Date'[Date] ),
        "Open Orders",
        SUMX (
            FILTER (
                GENERATE (
                    SUMMARIZE (
                        'Internet Sales',
                        'Internet Sales'[Order Date],
                        'Internet Sales'[Ship Date],
                        "Rows",
                        COUNTROWS ( 'Internet Sales' )
                    ),
                    DATESBETWEEN (
                        'Date'[Date],
                        'Internet Sales'[Order Date],
                        'Internet Sales'[Ship Date] - 1
                    )
                ),
                'Date'[Date] = EARLIER ('Date'[Date] )
            ),
            [Rows]
        )
    )
```

This final formulation of the query runs faster than all the previous ones (this runs in 40 milliseconds). We cannot say that this is the first solution that comes to mind when you start thinking at the problem, but it is a clear demonstration of how much the hard work of studying and understanding query plans can pay you in terms of performance gain.

Look at the logical query plan:

```
AddColumns: RelLogOp ([Date], ''[Open Orders])
    Scan_Vertipaq: RelLogOp ([Date]) Table= +BlankRow
    SumX: ScaLogOp ([Date]) BigInt DominantValue=BLANK
        Filter: RelLogOp RequiredCols ([Date], [Order Date], [Ship Date], ''[Rows], [Date])
            CrossApply: RelLogOp ([Order Date], [Ship Date], ''[Rows], [Date])
                AddColumns: RelLogOp ([Order Date], [Ship Date], ''[Rows])
                    Scan_Vertipaq: RelLogOp ('Product Category'[Product Category Name]) Table= -BlankRow
                    GroupBy_Vertipaq: RelLogOp ([Order Date], [Ship Date])
                    Table= -BlankRow
                        Scan_Vertipaq: RelLogOp ([Order Date], [Ship Date])
                        Table= -BlankRow
                    Count_Vertipaq: ScaLogOp ([Order Date], [Ship Date]) BigInt
                    Table= -BlankRow JoinCols([Order Date], [Ship Date])
                    Aggregations(Count!)
                        Scan_Vertipaq: RelLogOp RequiredCols([Order Date], [Ship Date])
                        Table= -BlankRow
                        JoinCols([Order Date], [Ship Date])
                DatesBetween: RelLogOp ([Order Date], [Ship Date], [Date])
                    [Order Date]: ScaLogOp
                    Subtract: ScaLogOp ([Ship Date]) DateTime
                        [Ship Date]: ScaLogOp ([Ship Date]) DateTime
                        Constant: ScaLogOp BigInt DominantValue=1
            EqualTo: ScaLogOp ([Date], [Date]) Boolean
                [Date]: ScaLogOp ([Date]) DateTime
                [Date]: ScaLogOp ([Date]) DateTime
        ''[Rows]: ScaLogOp (''[Rows]) BigInt
```

Here is the physical one:

```
AddColumns: IterPhyOp ([Date], ''[Open Orders])
    Spool_Iterator<Spool>: IterPhyOp ([Date]) #Records=2191
        AggregationSpool<Cache>: SpoolPhyOp #Records=2191
            VertipaqResult: IterPhyOp
    Spool: LookupPhyOp ([Date]) BigInt #Records=1133
        AggregationSpool<Sum>: SpoolPhyOp #Records=1133
            Extend_Lookup: IterPhyOp (''[Rows])
                Filter: IterPhyOp ([Date], [Order Date], [Ship Date], ''[Rows], [Date])
                    CrossApply: IterPhyOp ([Date], [Date])
                        Spool_MultiValuedHashLookup: IterPhyOp ([Date]) ([Order Date], [Ship Date], ''[Rows]) #Records=7868
                            AggregationSpool<GroupBy>: SpoolPhyOp #Records=7868
                                CrossApply: IterPhyOp ([Order Date], [Ship Date], ''[Rows], [Date])
                                    AddColumns: IterPhyOp ([Order Date], [Ship Date]) (''[Rows])
                                        Spool_UniqueHashLookup: IterPhyOp ([Order Date], [Ship Date]) #Records=1124
                                            AggregationSpool<Cache>: SpoolPhyOp #Records=1124
                                                VertipaqResult: IterPhyOp #FieldCols=2 #ValueCols=0
                                        Spool: LookupPhyOp ([Order Date], [Ship Date]) BigInt #Records=1124
                                            AggregationSpool<Cache>: SpoolPhyOp #Records=1124
                                                VertipaqResult: IterPhyOp1
                                    DatesBetween: IterPhyOp ([Order Date], [Ship Date], [Date])
                                        Extend_Lookup: IterPhyOp ([Order Date])
                                            Spool_Iterator<Spool>: IterPhyOp ([Order Date], [Ship Date]) #Records=1124
                                                AggregationSpool<GroupBy>: SpoolPhyOp #Records=1124
                                                    AddColumns: IterPhyOp ([Order Date], [Ship Date], ''[Rows])
                                                        Spool_Iterator<Spool>: IterPhyOp
                                                        ([Order Date], [Ship Date]) #Records=1124
                                                            AggregationSpool<Cache>: SpoolPhyOp #Records=1124
                                                                VertipaqResult: IterPhyOp
                                                        Spool: LookupPhyOp ([Order Date], [Ship Date]) BigInt #Records=1124
                                                            AggregationSpool<Cache>: SpoolPhyOp #Records=1124
                                                                VertipaqResult: IterPhyOp #FieldCols=2 #ValueCols=1
                                            [Order Date]: LookupPhyOp ([Order Date]) DateTime
                                        Subtract: LookupPhyOp ([Ship Date]) DateTime
                                            [Ship Date]: LookupPhyOp ([Ship Date]) DateTime
                                            Constant: LookupPhyOp BigInt 1
                        InnerHashJoin: IterPhyOp ([Date], [Date])
                            Extend_Lookup: IterPhyOp ([Date])
                                Spool_Iterator<Spool>: IterPhyOp ([Date]) #Records=2191
                                    AggregationSpool<Cache>: SpoolPhyOp #Records=2191
                                        VertipaqResult: IterPhyOp
                                [Date]: LookupPhyOp ([Date]) DateTime
                            HashLookup: IterPhyOp ([Date]) #Recs=1133
                                HashByValue: SpoolPhyOp #Records=1133
                                    Extend_Lookup: IterPhyOp ([Date])
                                        Spool_Iterator<Spool>: IterPhyOp ([Date]) #Records=1133
                                            AggregationSpool<GroupBy>: SpoolPhyOp #Records=1133
                                                ApplyRemap: IterPhyOp ([Order Date], [Ship Date], ''[Rows], [Date])
                                                    Spool_Iterator<Spool>: IterPhyOp ([Order Date], [Ship Date],
                                                    ''[Rows], [Date]) #Records=7868
                                                        AggregationSpool<GroupBy>: SpoolPhyOp #Records=7868
                                                            CrossApply: IterPhyOp ([Order Date], [Ship Date],
                                                            ''[Rows], [Date])
                                                                AddColumns: IterPhyOp ([Order Date], [Ship Date]) (''[Rows])
                                                                    Spool_UniqueHashLookup: IterPhyOp ([Order Date],
                                                                    [Ship Date]) #Records=1124
                                                                        AggregationSpool<Cache>: SpoolPhyOp #Records=1124
                                                                            VertipaqResult: IterPhyOp
                                                                    Spool: LookupPhyOp ([Order Date], [Ship Date])
                                                                    BigInt #Records=1124
                                                                        AggregationSpool<Cache>: SpoolPhyOp #Records=1124
                                                                            VertipaqResult: IterPhyOp
                                                                DatesBetween: IterPhyOp ([Order Date], [Ship Date], [Date])
                                                                    Extend_Lookup: IterPhyOp ([Order Date])
                                                                        Spool_Iterator<Spool>: IterPhyOp ([Order Date],
                                                                        [Ship Date]) #Records=1124
                                                                            AggregationSpool<GroupBy>: SpoolPhyOp
                                                                            #Records=1124
                                                                                AddColumns: IterPhyOp ([Order Date],
                                                                                [Ship Date], ''[Rows])
                                                                                    Spool_Iterator<Spool>: IterPhyOp
                                                                                    ([Order Date], [Ship Date]) #Records=1124
                                                                                        AggregationSpool<Cache>: SpoolPhyOp
                                                                                        #Records=1124
                                                                                            VertipaqResult: IterPhyOp
                                                                                    Spool: LookupPhyOp ([Order Date],
                                                                                    [Ship Date]) BigInt #Records=1124
                                                                                        AggregationSpool<Cache>: SpoolPhyOp
                                                                                        #Records=1124
                                                                                            VertipaqResult: IterPhyOp
                                                                        [Order Date]: LookupPhyOp ([Order Date]) DateTime
                                                                    Subtract: LookupPhyOp([Ship Date]) DateTime
                                                                        [Ship Date]: LookupPhyOp([Ship Date]) DateTime
                                                                        Constant: LookupPhyOp BigInt 1
                                        [Date]: LookupPhyOp ([Date]) DateTime
                ''[Rows]: LookupPhyOp (''[Rows]) BigInt
```

Understanding and commenting the query plans, this time, is left as a useful exercise to the reader!

We moved from a 22 seconds query to one that runs in 40 milliseconds (i.e. 550 times faster). In order to do that, we had to study several query plans and finally re-write the DAX code leveraging on the knowledge of

the faster operations performed by the engine. In this way, we can improve the performance of our so-far-best-performer.

It is clear that you cannot afford this kind of job for all of the measures in a project but, for time-critical measures, it clearly shows a path to follow when optimizing DAX.

Another useful aspect of this kind of analysis is the fact that, by analyzing the code in this final query, you can predict the performance of the query based on the size of the dataset. In fact, the complexity of the query increases with:

- The size of the fact table (which is evident and expected).
- The number of distinct values of (OrderDate and ShipDate), which is less evident.
- The average difference between OrderDate and ShipDate. In fact, the higher this delta, the higher the number of rows produced by DATESBETWEEN and further elaborated by FILTER, increasing the execution time.

Gathering this information from the previous formulation of the query was nearly impossible, whereas both the query plan and this new format of the query greatly help in understanding how DAX compute the values and what to expect with a growth in the database size.