Understanding Distinct Count in DAX Query Plans

DISTINCT COUNTS ANALYZED USING QUERY PLANS

produced by

sqlbi

Understanding Distinct Count in DAX Query Plans

Analysis of different query plans for different formulations of the distinct count. While speaking about distinct count, I will show some common optimization techniques for the DAX language.

Author: Alberto Ferrari

Published: Version 1.0 Revision 2 – January, 15th 2014.

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

Summary: This paper performs an in-depth analysis of different ways of expressing distinct counts in DAX. For each pattern and each query, we provide a brief description of the query plan used by DAX to compute the value, searching for the best formulation for distinct count.

Acknowledgments: I would like to thank the peer reviewers who helped me to improve 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—smart individuals who patiently answer all of our fancy questions about SSAS.

Table of Contents

Introduction

Distinct count is a very common calculation in any SSAS database—there is a good chance that you already know the DISTINCTCOUNT function, which exists for this purpose. DISTINCTCOUNT provides good performance in many cases, but there are scenarios where using other patterns to compute distinct count will provide better performance. The speed improvement can be impressive, although, as you might expect, you will need to perform some tests before choosing the best technique for your model.

In this article, I will explore different methods to perform distinct count and, by following the DAX query plans, understand the differences in how the VertiPaq engine (yes, I know it is named "xVelocity-InMemory-Engine," but I love to call it VertiPaq) resolves the query. Thus, get prepared for a geeky dive in the DAX query engine internals.

For the tests, I used a data model with a fact table containing 4 billion rows and one dimension (customers) with 152,275 rows. The dimension is a slowly changing dimension of type 2, so that multiple records exist for the same customer. I cannot share the database because it contains customer data, but you should be able to reproduce similar results on your database.

I have executed all of the tests with a cold cache, because most of the queries use only SE—they will be completely cached, and running them with a hot cache would not provide useful information.

As you will learn by reading this white paper, different patterns provide different performance but, in order to obtain the best performance, you need to work at the query level and write the query in such a way that the VertiPaq engine follows the best execution path. Unfortunately, you do not always have full control over the queries and, in this case, you will need to make decisions based on the expected usage of the measures in your database.

IMPORTANT: Please note that the goal of this paper is to understand the different query plans and operations performance by the query engine. **This is not an ultimate guide to distinct count optimization**. The performance of distinct count calculations is affected by many other factors, such as the number of distinct values in the column and in the result set. Your mileage may vary a lot, so test everything in your specific data model. At the end of the paper, we provide a comparison between the same set of queries on two different databases in order to give you an idea of how important testing is.

Base Measures

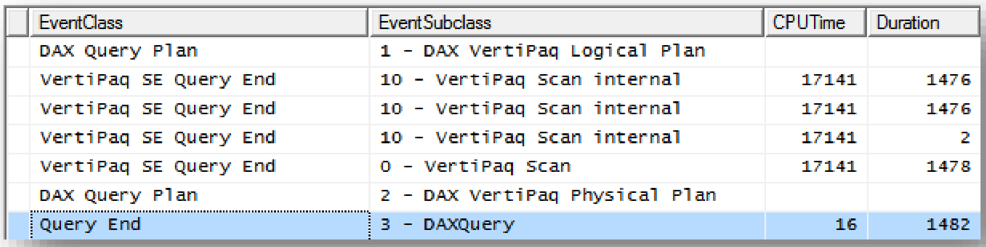
Before trying more complex queries, it is useful to spend some time looking at the query plans of different formulations of distinct count. Later we will work on queries that more authentically reflect real-world scenarios.

Using Basic DISTINCTCOUNT

The first measure I tested is the naïve one:

```
EVALUATE
    ROW ( "Result", DISTINCTCOUNT ( Fact[CustomerKey] ) )
```

This query runs in 1,482 milliseconds on my workstation. Looking at the query plan, you see this result:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	17141	1476
VertiPaq SE Query End	10 - VertiPaq Scan internal	17141	1476
VertiPaq SE Query End	10 - VertiPaq Scan internal	17141	2
VertiPaq SE Query End	0 - VertiPaq Scan	17141	1478
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	16	1482

The interesting parts of this query plan are:

- It is resolved by VertiPaq scans, so it makes very good use of hardware: 17,141 milliseconds of CPU time results in a duration of 1,476 milliseconds due to the high level of parallelism in VertiPaq.
- Formula Engine (FE) time is negligible, and this is very good.
- The original VertiPaq query is:

```
SELECT
    DCOUNT ( Fact.CustomerKey )
FROM Fact;
```

And there are two internal scans:

```
SELECT
    Fact.CustomerKey
FROM Fact


SELECT COUNT () FROM $DCOUNT_DATACACHE
```

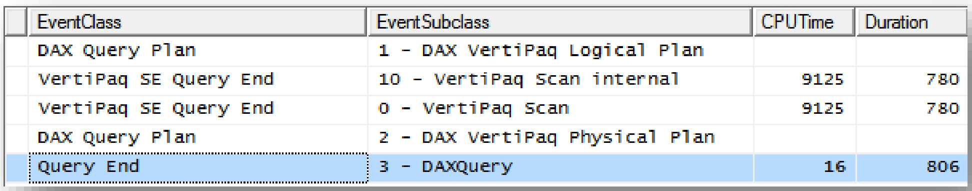
DAX computes the distinct count by first running a query to perform a GROUPBY on the fact table for the CustomerKey and then retrieving the number of rows in that dataset. This is why you see a single VertiPaq scan even though it is resolved by different internal VertiPaq scans (one for the GROUPBY and one for the COUNT of the resulting set).

Using the SUMMARIZE Pattern

You can use the SUMMARIZE function to compute the same measure, performing—with a more complex DAX expression—the same operations carried on by the optimizer when it evaluates the DISTINCTCOUNT function:

```
EVALUATE
    ROW ( "Result", COUNTROWS ( SUMMARIZE ( Fact, Fact[CustomerKey] ) ) )
```

Although this query looks more complex, the result is impressive: it runs in 806 milliseconds (compared with the 1,482 of the previous measure). The query plan is simpler this time:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	9125	780
VertiPaq SE Query End	0 - VertiPaq Scan	9125	780
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	16	806

The VertiPaq scan executes this query:

```
SELECT
    Fact.CustomerKey
FROM Fact
```

But this time, the query runs for 780 milliseconds, using only 9,125 milliseconds of CPU time. The result of the VertiPaq scan is then aggregated in FE using a simple COUNT, and it is very fast.

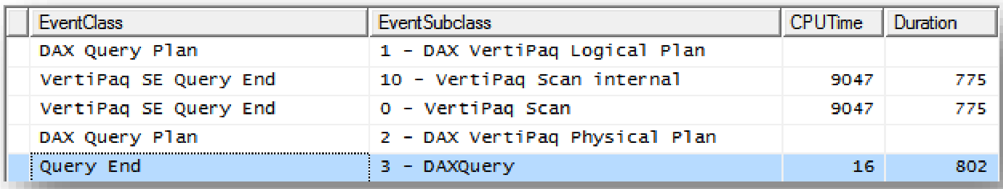
When I test this pattern for different columns in the fact table, the execution time strongly depends on the number of distinct values of the column counted, but the SUMMARIZE pattern always results in better performance. Of course, the pattern of the query plan is always the same.

Testing the SUMX Pattern

There is another pattern for distinct count that some customers report as a better alternative to the use of the DISTINCTCOUNT function. Instead of using a counting function (COUNTROWS or DISTINCTCOUNT), you use an iteration: SUMX with a constant value of 1.

```
EVALUATE
    ROW ( "Result", SUMX ( VALUES ( Fact[CustomerKey] ), 1 ) )
```

The query plan of this query is very simple, resulting in performance that is identical to the SUMMARIZE version. In fact, the algorithm used by DAX for SUMX is very similar to the SUMMARIZE pattern but, this time, FE uses a SUM with a constant instead of a COUNT to aggregate the result of the VertiPaq scan.



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	9047	775
VertiPaq SE Query End	0 - VertiPaq Scan	9047	775
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	16	802

The VertiPaq scan is the same as that for the SUMMARIZE pattern:

```
SELECT
    Fact.CustomerKey
FROM Fact
```

Thus, for the base measure, there seem to be no difference between the usage of SUMMARIZE and SUMX: both result in better performance when compared with DISTINCTCOUNT.

Handling Slowly Changing Dimensions

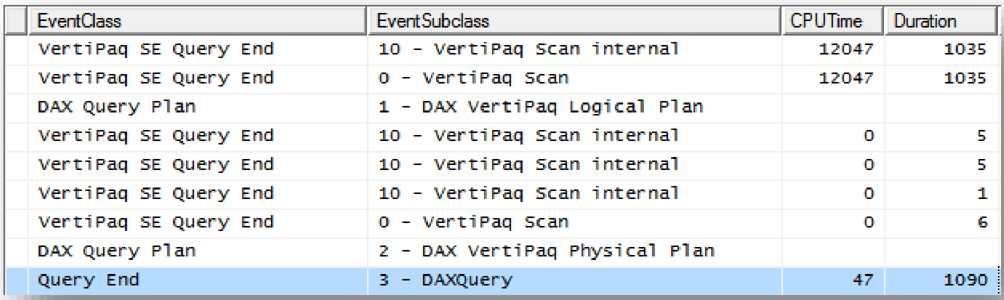
As I said at the beginning, the customer dimension is a slowly changing dimension (SCD). Thus, we cannot compute the number of distinct customers with any of the previous formulas. In fact, what we have counted is the number of surrogate keys (CustomerKey), while we should have counted the number of customers' codes. The number of codes is always smaller than the number of keys for a type 2 SCD.

To compute the distinct count of codes, we can use the many-to-many pattern. In fact, in a dimensional model, you can think of the fact table as a bridge table between any two dimensions, and you can use the many-to-many pattern to find relationships between dimensions. In this case, we only have a single dimension to analyze, but we will compute the number of codes in the dimension after having filtered it with the fact table as if it was a bridge.

Let us start with the simple DISTINCTCOUNT:

```
EVALUATE
    ROW (
        "Result",
        CALCULATE ( DISTINCTCOUNT ( Customers[CustomerCode] ), Fact )
    )
```

The results are surprising. This query runs in 1,090 milliseconds (yes, faster than the distinct count of the customer key—isn't it unbelievable?). Let us take a look at the query plan:



EventClass	EventSubclass	CPUTime	Duration
VertiPaq SE Query End	10 - VertiPaq Scan internal	12047	1035
VertiPaq SE Query End	0 - VertiPaq Scan	12047	1035
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	5
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	5
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	1
VertiPaq SE Query End	0 - VertiPaq Scan	0	6
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	47	1090

The first VertiPaq query computes the customer keys from the customers table reached using a JOIN from the fact table, and it is responsible for the majority of the query duration:

```
SELECT
    Customers.CustomerKey,
    COUNT()
FROM
    Fact
    LEFT OUTER JOIN Customers
        ON Customers.CustomerKey = Fact.CustomerKey
```

It is important to note that, although the column for which we requested a distinct count is the customer code, this query gathers the surrogate keys from the fact table. The remaining three VertiPaq queries follow the DISTINCTCOUNT pattern (retrieval of customer code and then computation of the count from the $DCOUNT_DATACACHE) and their execution time is negligible. It is worth looking at the first of these queries:

```
SELECT
    Customers.CustomerCode
FROM
    Customers
WHERE
  Customers.CustomerKey IN (17547, 28294, 115157, 56591, 132707, ...
                           [144631 total values, not all displayed])
```

The query scans the customers table using a bitmap built in the previous VertiPaq query that shows which CustomerKeys should be part of the query. Do not be impressed by the textual representation of the query: the filter CustomerKey IN (...) seems to be very inefficient (and it would be, in SQL) but it is only intended to be human-readable. In reality, it is a simple bitmap scan of a small table and, in fact, it runs very quickly.

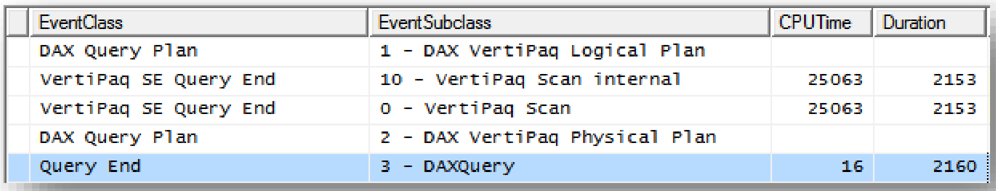
Thus, DAX followed a simple algorithm to compute the distinct count: it built a bitmap of customer keys and then used that bitmap to scan the dimension and compute the distinct count on the smaller table. The resulting execution time is very good.

> At this point, a careful reader should note that, because this algorithm is faster than the one we used to compute distinct count without SCD handling, a good query to test would have been:
>
> ```
> EVALUATE
>     ROW (
>         "Result",
>         CALCULATE ( DISTINCTCOUNT  ( Customers[CustomerKey] ), Fact )
>     )
> ```
>
> In fact, it turns out that this query computes the very same result as the basic DISTINCTCOUNT but it is faster, running in 1,097 milliseconds instead of the original 1,482.

It is worth noting that, in this case, the DISTINCTCOUNT function performed a different algorithm, so it might be the case that expressing the query with the SUMMARIZE pattern will not result in the same boost in performance we have previously seen. We tried this query:

```
EVALUATE
    ROW (
        "Result",
        COUNTROWS ( SUMMARIZE ( Fact, Customers[CustomerCode] ) )
    )
```

In fact, this time the query runs in 2,160 milliseconds. The query plan shows what happened:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	25063	2153
VertiPaq SE Query End	0 - VertiPaq Scan	25063	2153
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	16	2160

All of the time is used by a single VertiPaq query, which contains the exact algorithm expressed by our DAX query:

```
SELECT
    Customers.CustomerCode, COUNT()
FROM
    Fact
LEFT OUTER JOIN Customers
    ON Customers.CustomerKey = Fact.CustomerKey
```

Surprisingly, this time the JOIN is executed inside VertiPaq with a pattern very similar to the one used in the previous query, but it takes twice the time. You can see that the VertiPaq query does not retrieve the CustomerKey but instead directly retrieves the CustomerCode.

It turns out that, this time, the SUMMARIZE version performs worse than the DISTINCTCOUNT one. It seems strange that a JOIN performed in VertiPaq results in slower performance than a bitmap scan following a query. Thus, we must test a different query. This time, instead of computing the distinct count of the customer code (which has the same cardinality of the customer key), we will compute the distinct count of the customer age range, which has a much lower cardinality (there are only 9 age ranges of customers, compared with 150,000 customer codes).

The DISTINCTCOUNT query is the following; others follow a similar pattern:

```
EVALUATE
    ROW (
        "Result",
        CALCULATE ( DISTINCTCOUNT ( Customers[AgeRange] ), Fact )
    )
```

Suppose we replace CustomerCode with AgeRange in all of the previous queries. It turns out that, with a lower cardinality attribute, the SUMMARIZE version is faster than the DISTINCTCOUNT one. In fact, the SUMMARIZE version runs in 639 milliseconds while the DISTINCTCOUNT one uses 1,069 milliseconds. The query plans are the same as before, the only difference being in the column retrieved and in timing.

It is interesting to note that the DISTINCTCOUNT version of the algorithm uses the same time to compute the distinct count on attributes of different cardinality, while the SUMMARIZE pattern performs better with lower cardinality and worse with higher cardinality. Because you normally can predict the cardinality of an attribute, it might be useful to adopt different patterns for different measures.
Finally, what "cardinality" means really depends on your scenario. The number of rows in the fact table and the number of distinct values for dimensions interact in such a complex way that testing is always necessary in order to take an educated guess about which pattern to use.

Using More Complex Queries

Exploring how the base measures work is useful to understand the different ways in which DAX solves the different formulas, and to grasp a first understanding of the patterns. Of course, testing the base measure only evaluates the time required to compute a single cell. We now want to use the different patterns in more complex queries in order to check the overall performance of a query.

It is important to note that when you work with a complex query, the shape of the query plan changes depending on both the base measure and the DAX query. Thus, your specific scenario results likely will be different from the ones I show in this paper.

We will test the different formulations of DISTINCTCOUNT in various scenarios:

- SUMMARIZE on a single column from the same table

```
SUMMARIZE (
    Fact,
    Customers[AgeRange],
    "Result", <ExpressionToCompute>
)
```

- SUMMARIZE on a single column from a different table

```
SUMMARIZE (
    Fact,
    'Date'[MonthName],
    "Result", <ExpressionToCompute>
)
```

- SUMMARIZE on multiple columns from different tables

```
SUMMARIZE (
    Fact,
    'Date'[MonthName],
    Time[Period60Minutes],
    "Result", <ExpressionToCompute>
)
```

SUMMARIZE on Same Table

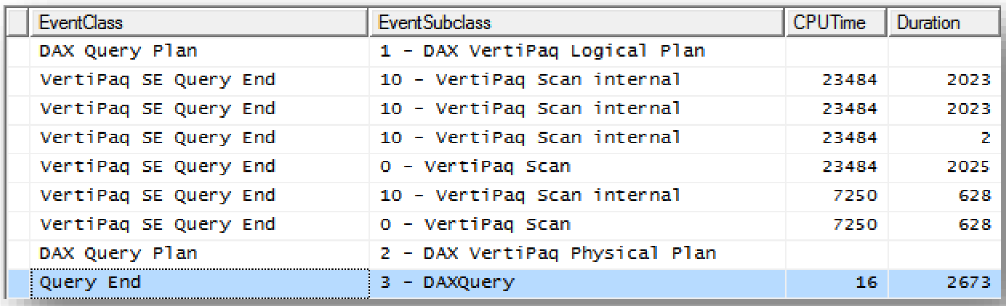
Let us start investigating on performance, computing the customer count and performing a GROUPBY with a column in the Customers table.

DISTINCTCOUNT

Although very simple, this query is interesting to analyze:

```
SUMMARIZE (
    Fact,
    Customers[AgeRange],
    "Result", DISTINCTCOUNT ( Fact[CustomerKey] )
)
```

Please note that we are aggregating on the AgeRange column from the Customers table but the DISTINCTCOUNT is—in reality—working on the CustomerKey column from the fact table. It is worth noting this because in a real-world database you might end up writing base measures for the NumOfCustomers using a DISTINCTCOUNT on the fact table and then simply forget that, when aggregating values, you need to work on different tables. As we will see, this query can be greatly optimized by writing the measure in a different way. Anyway, let us start with this query plan:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	23484	2023
VertiPaq SE Query End	10 - VertiPaq Scan internal	23484	2023
VertiPaq SE Query End	10 - VertiPaq Scan internal	23484	2
VertiPaq SE Query End	0 - VertiPaq Scan	23484	2025
VertiPaq SE Query End	10 - VertiPaq Scan internal	7250	628
VertiPaq SE Query End	0 - VertiPaq Scan	7250	628
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	16	2673

The full execution time is 2,673 milliseconds, all spent in SE on two VertiPaq queries. The first one uses the DISTINCTCOUNT pattern of a first SELECT followed by the usage of $DCOUNT_DATACACHE:

```
SELECT
    Customers.AgeRange,
    Fact.CustomerKey,
    COUNT()
FROM
    Fact
    LEFT OUTER JOIN Customers
        ON Customers.CustomerKey = Fact.CustomerKey
```

This query, by itself, already returns the final result and runs in 2,023 milliseconds. Yet DAX needs to aggregate by Customers[AgeRange] and, thus, it runs a second query to get the values of Customers[AgeRange] from the same fact table. This is the second query, running for 628 milliseconds:

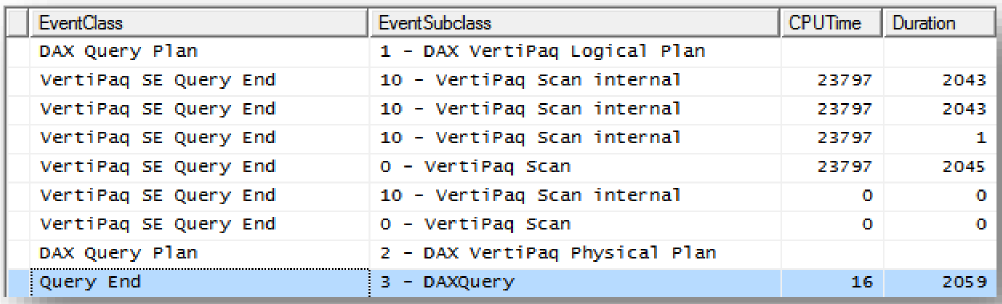
```
SELECT
    Customers.AgeRange,
    COUNT()
FROM
    Fact
    LEFT OUTER JOIN Customers
        ON Customers.CustomerKey = Fact.CustomerKey
```

Finally, FE makes a JOIN between the two results, producing the final dataset.

A standard optimization technique in DAX is to replace the SUMMARIZE with ADDCOLUMNS. If we do that now, the DAX query becomes:

```
ADDCOLUMNS (
    VALUES ( Customers[AgeRange] ),
    "Result", CALCULATE ( DISTINCTCOUNT ( Fact[CustomerKey] ) )
)
```

We needed to add CALCULATE to force a context transition, and the query plan becomes:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	23797	2043
VertiPaq SE Query End	10 - VertiPaq Scan internal	23797	2043
VertiPaq SE Query End	10 - VertiPaq Scan internal	23797	1
VertiPaq SE Query End	0 - VertiPaq Scan	23797	2045
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	2059

You can easily see that the second VertiPaq query changed from 628 milliseconds to zero. The reason is that, this time, DAX did not need to scan the fact table to get the values of Customer[AgeRange]—instead, it used the dimension, which is much smaller. Thus, the execution time is much better than before.

If you are concerned about blank results, which might appear in the result set because the AgeRange values come from the Customer table instead of the fact table, you can change the query to:

```
FILTER (
    ADDCOLUMNS (
        VALUES ( Customers[AgeRange] ),
        "Result", CALCULATE ( DISTINCTCOUNT ( Fact[CustomerKey] ) )
    ),
    NOT ( ISBLANK ( [Result] ) )
)
```

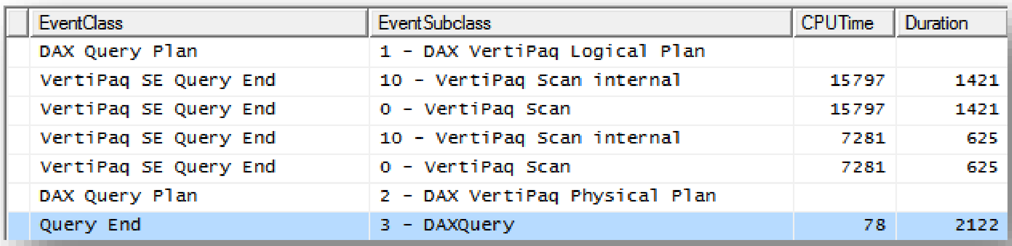
Because the result set contains few columns, the FILTER operation—although executed in FE—is very fast. In fact, this final query returns the same result as the first one, but it is 25% faster.

SUMMARIZE

Let us perform the same tests we did before but, this time, using the SUMMARIZE version of distinct counts:

```
SUMMARIZE (
    Fact,
    Customers[AgeRange],
    "Result", COUNTROWS ( SUMMARIZE ( Fact, Fact[CustomerKey] ) )
)
```

Again, we are summarizing using the AgeRange in Customers and counting values from the fact table. We can expect performance similar to the original query using DISTINCTCOUNT. In fact, the query plan looks similar to our first test:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	15797	1421
VertiPaq SE Query End	0 - VertiPaq Scan	15797	1421
VertiPaq SE Query End	10 - VertiPaq Scan internal	7281	625
VertiPaq SE Query End	0 - VertiPaq Scan	7281	625
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	78	2122

The queries are identical to the ones used with DISTINCTCOUNT but, this time, there is no $DCOUNT_DATACACHE involved in the calculation. In fact, the first query returns the fact table grouped by AgeRange and CustomerKey.

```
SELECT
    Customers.AgeRange,
    Fact.CustomerKey,
    COUNT()
FROM
    Fact
    LEFT OUTER JOIN Customers
        ON Customers.CustomerKey = Fact.CustomerKey
```

The second one returns the AgeRange values from the fact table:

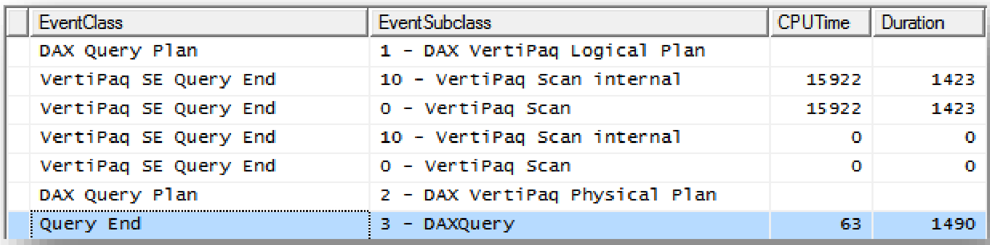
```
SELECT
    Customers.AgeRange,
    COUNT()
FROM
    Fact
    LEFT OUTER JOIN Customers
        ON Customers.CustomerKey = Fact.CustomerKey
```

And the FE joins the two results using the AgeRange column, grouping by COUNT on the CustomerKey. It is worth noting that, although there are 144,631 rows in the result of the first query, the full join takes only 78 milliseconds. As a result, this query is already faster than the DISTINCTCOUNT one.

The same considerations we followed for the DISTINCTCOUNT apply here. We are using SUMMARIZE on the fact table while we could have optimized the code using ADDCOLUMNS, to remove 625 milliseconds from the execution time:

```
ADDCOLUMNS (
    VALUES ( Customers[AgeRange] ),
    "Result", CALCULATE ( COUNTROWS ( SUMMARIZE ( Fact, Fact[CustomerKey] ) ) )
)
```

As expected, the query plan shows an interesting decrease in execution time because the second VertiPaq query no longer takes 625 milliseconds but goes down to a beautiful zero:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	15922	1423
VertiPaq SE Query End	0 - VertiPaq Scan	15922	1423
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	63	1490

Still, we strive for perfection, and we can add one more consideration to this query: can we help VertiPaq reduce the scans? We can, if we apply a different algorithm:

- First, we filter the Customers table to make it contain only rows that are referenced in the fact table, using the many-to-many pattern.

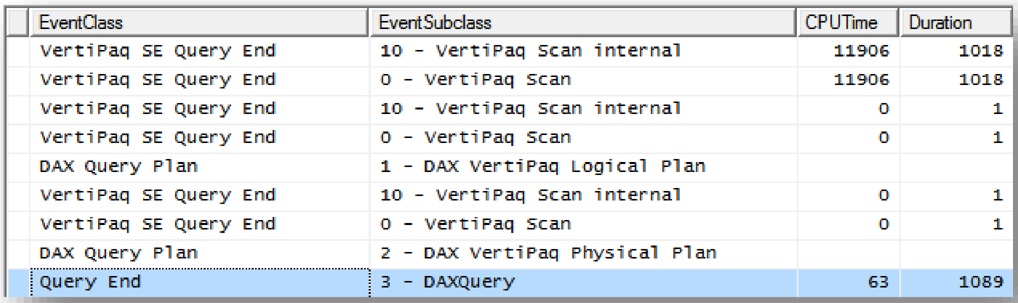
- Second, we do a GROUPBY on the Customers table, counting the customer keys that are present in the table after the first filter, and grouping by AgeRange.

The good part of this plan is that most of the calculation happens on the Customers table. In this way, we reduce the usage of the fact table to the building of a filter that needs to be applied to the Customers table before starting the computation.

This is the DAX query:

```
CALCULATETABLE (
    ADDCOLUMNS (
        VALUES ( Customers[AgeRange] ),
        "Result", CALCULATE ( COUNTROWS ( VALUES ( Customers[CustomerKey] ) ) )
    ),
    Fact
)
```

And this is the query plan:



EventClass	EventSubclass	CPUTime	Duration
VertiPaq SE Query End	10 - VertiPaq Scan internal	11906	1018
VertiPaq SE Query End	0 - VertiPaq Scan	11906	1018
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	1
VertiPaq SE Query End	0 - VertiPaq Scan	0	1
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	1
VertiPaq SE Query End	0 - VertiPaq Scan	0	1
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	63	1089

The execution time is 1,089 milliseconds. By looking at the queries, you will get a clear feeling of what is happening under the cover. There are three VertiPaq queries.

The first one retrieves the keys of customers starting from the fact table in JOIN with the Customers table.

```
SELECT
    Customers.CustomerKey,
    COUNT()
FROM
    Fact
    LEFT OUTER JOIN Customers
        ON Customers.CustomerKey = Fact.CustomerKey
```

The second query computes the different age ranges and, this time, does not need to scan the fact table. It already knows which customers to query, thanks to the previous result, which built a bitmap to apply to the Customers table to show only the customers who are present in the fact table:

```
SELECT
    Customers.AgeRange,
    COUNT()
FROM
    Customers
  WHERE Customers.CustomerKey IN (17547, 28294, 115157, 56591, 132707, 143454, 26322, ...
                                 [144631 total values, not all displayed])
```

The third query computes the final result, returning the count of customers after having applied a filter on both AgeRange and CustomerKey:

```
SELECT
    Customers.AgeRange,
    COUNT()
FROM
    Customers
  WHERE Customers.CustomerKey INB (17547, 28294, 115157, 56591, 132707, 143454, ...
                                 [144631 total values, not all displayed]) AND
       Customers.AgeRange INB ('0-10', '10-20', ...)
```

This last query seems like a repetition. In fact, the FE will simply join these two queries, taking the age ranges from the first one and the count from the second one, in order to build the final dataset.

It is useful to note that, in this case, the SUMMARIZE version outperforms DISTINCTCOUNT, running twice as fast. We had to rewrite the query to take advantage of the fact that we are doing a SUMMARIZE on the same table. The result is worth the effort, but it cannot be easily achieved working on the measure alone—you need to work on the query as a whole to obtain best performance.

You can achieve similar results using DISTINCTOUNT but, in this case, you need to perform the DISTINCTCOUNT on the CustomerKey on the Customers table and not on the fact table.

```
CALCULATETABLE (
    ADDCOLUMNS (
        VALUES ( Customers[AgeRange] ),
        "Result", CALCULATE ( DISTINCTCOUNT ( Customers[CustomerKey] ) )
    ),
    Fact
)
```

The SUMX version of the pattern yields the same results as the SUMMARIZE version, as might be expected:

```
CALCULATETABLE (
    ADDCOLUMNS (
        VALUES ( Customers[AgeRange] ),
        "Result", CALCULATE ( SUMX ( VALUES ( Customers[CustomerKey] ), 1 ) )
    ),
    Fact
)
```

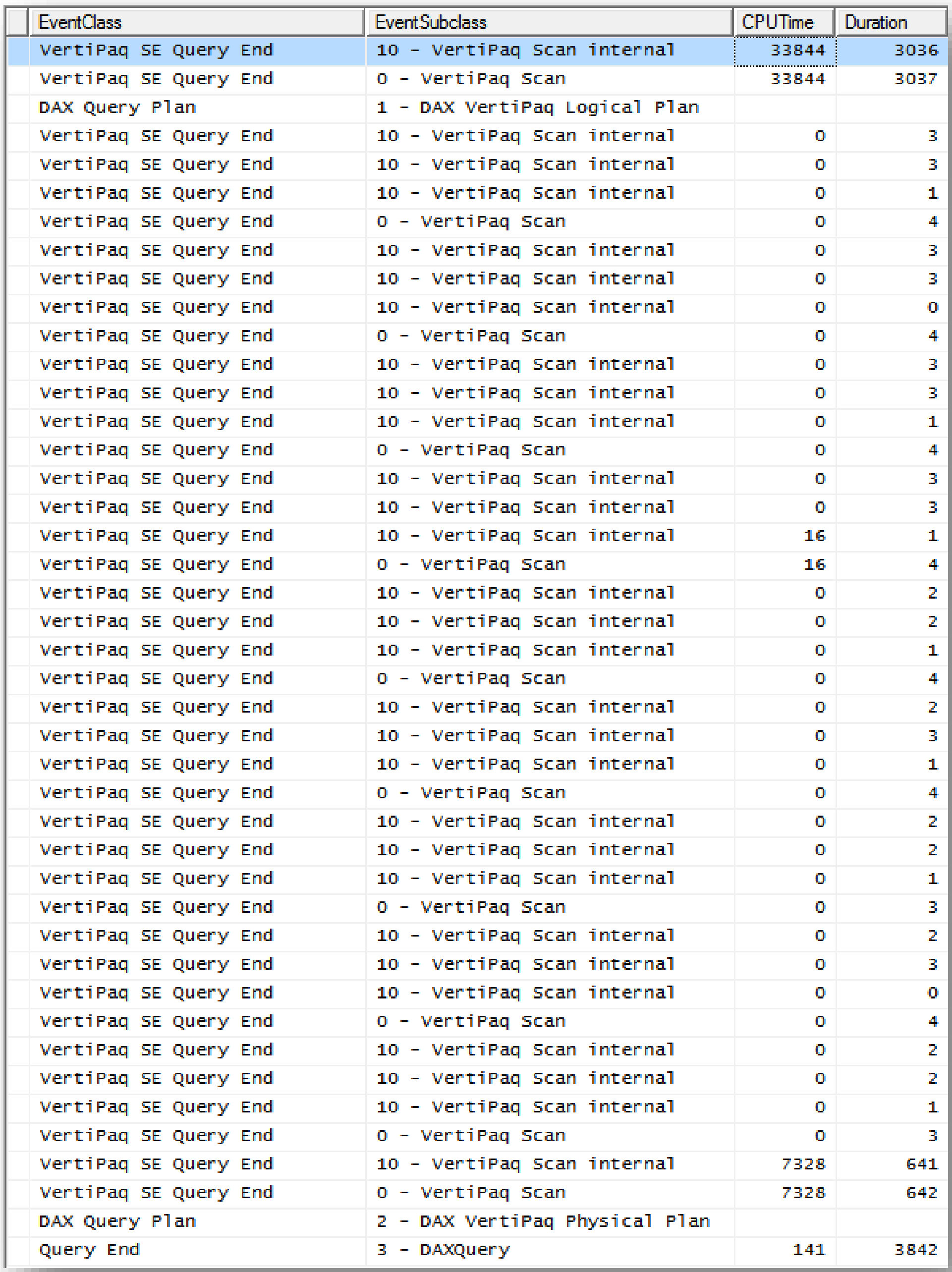
SCD Handling

This query is interesting to look at. The reason is that it is a very common pattern you are likely to encounter if you do not spend time optimizing your queries, and the performance is horrible.

```
SUMMARIZE (
    Fact,
    Customers[AgeRange],
    "Result", CALCULATE ( DISTINCTCOUNT ( Customer[CustomerCode] ), Fact )
)
```

It is very likely that you wrote the DISTINCTCOUNT as a measure and then you used it in a query, without worrying about the performance or, worse, taking for granted that performance cannot be further optimized. In fact, remember that the DISTINCTCOUNT pattern was our top performer for the handling of slowly changing dimensions, when evaluated as a single measure.

Look at the query plan:



EventClass	EventSubclass	CPUTime	Duration
VertiPaq SE Query End	10 - VertiPaq Scan internal	33844	3036
VertiPaq SE Query End	0 - VertiPaq Scan	33844	3037
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	1
VertiPaq SE Query End	0 - VertiPaq Scan	0	4
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	0
VertiPaq SE Query End	0 - VertiPaq Scan	0	4
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	1
VertiPaq SE Query End	0 - VertiPaq Scan	0	4
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	16	1
VertiPaq SE Query End	0 - VertiPaq Scan	16	4
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	1
VertiPaq SE Query End	0 - VertiPaq Scan	0	4
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	1
VertiPaq SE Query End	0 - VertiPaq Scan	0	4
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	1
VertiPaq SE Query End	0 - VertiPaq Scan	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	0
VertiPaq SE Query End	0 - VertiPaq Scan	0	4
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	1
VertiPaq SE Query End	0 - VertiPaq Scan	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	7328	641
VertiPaq SE Query End	0 - VertiPaq Scan	7328	642
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	141	3842

What happened here? Well, this time the query plan is somewhat naïve. The first VertiPaq query returns, as usual, AgeRange and CustomerKey by scanning the fact table but, this time, returns values from the Customer table only. If you carefully look at the queries executed for the previous tests, you will notice that the customer key is taken from the fact table, not from the Customer table.

```
SELECT
    Customers.AgeRange,
    Customers.CustomerKey,
    COUNT()
FROM
    Fact
    LEFT OUTER JOIN Customers
        ON Customers.CustomerKey = Fact.CustomerKey
```

Following this long-running query, there is a set of scans that all take the following form:

```
SELECT
    Customers.CustomerCode,
    COUNT()
FROM
    Customers
    WHERE Customers.CustomerKey INB (17547, 28294, 115157, 56591, 132707, 143454, 84888,
26322, ...
                                    [144631 total values, not all displayed]) VAND
    WHERE Customers.CustomerKey INB (5, 6, 10, 90, ...
                                    [20083 total values, not all displayed])
```

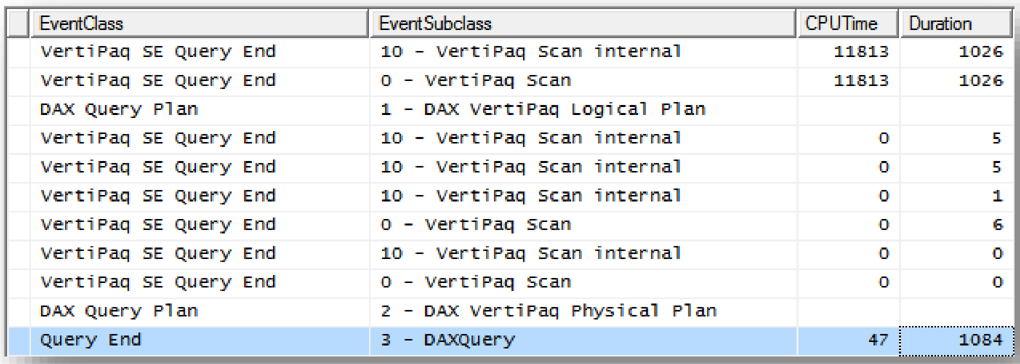
All of these scans are fast but, as you might expect, increasing the number of distinct values in the AgeRange column will affect the query performance, increasing the number of queries.

This time, the engine is really iterating over the different AgeRange values. For each one, it executes a separate VertiPaq query to grab the value of that specific range. FE is making much more work, because it needs to coordinate the results coming from different VertiPaq queries. The final time of 3,842 milliseconds is very high.

Optimizing this query by using ADDCOLUMNS instead of SUMMARIZE dramatically changes the result:

```
ADDCOLUMNS (
    VALUES ( Customers[AgeRange] ),
    "Result", CALCULATE ( DISTINCTCOUNT ( Customer[CustomerCode] ), Fact )
)
```

In fact, the query plan is very different:



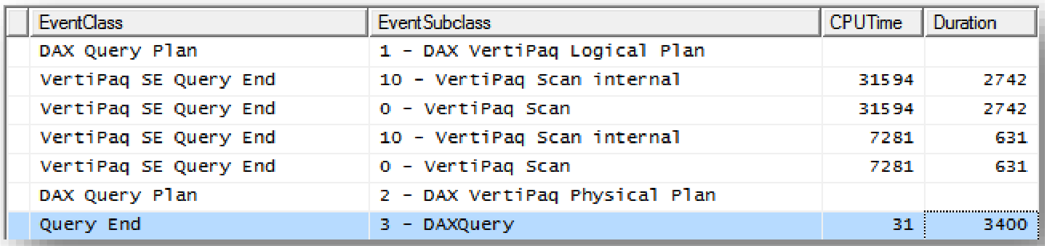
EventClass	EventSubclass	CPUTime	Duration
VertiPaq SE Query End	10 - VertiPaq Scan internal	11813	1026
VertiPaq SE Query End	0 - VertiPaq Scan	11813	1026
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	5
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	5
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	1
VertiPaq SE Query End	0 - VertiPaq Scan	0	6
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	47	1084

There is no need, at this point, to perform a full analysis of the query plan. The pattern used by the optimizer is the same used for the last optimization of the previous section. This time, instead of computing the DISTINCTCOUNT of the CustomerKey, we used the DISTINCTCOUNT of the CustomerCode. DAX used the pattern of identifying the CustomerKeys present in the fact table and later used that set to filter the customers and perform the DISTINCTCOUNT on the Customer table. From this wonderful algorithm follows the timing of only 1,084 milliseconds.

Let us look at what happens if we try the naïve version of the query but, this time, with the SUMMARIZE version of SCD handling:

```
SUMMARIZE (
    Fact,
    Customers[AgeRange],
    "Result", COUNTROWS ( SUMMARIZE ( Fact, Customer[CustomerCode] ) )
)
```

The query plan is the following:



	EventClass	EventSubclass	CPUTime	Duration
	DAX Query Plan	1 - DAX VertiPaq Logical Plan		
	VertiPaq SE Query End	10 - VertiPaq Scan internal	31594	2742
	VertiPaq SE Query End	0 - VertiPaq Scan	31594	2742
	VertiPaq SE Query End	10 - VertiPaq Scan internal	7281	631
	VertiPaq SE Query End	0 - VertiPaq Scan	7281	631
	DAX Query Plan	2 - DAX VertiPaq Physical Plan		
	Query End	3 - DAXQuery	31	3400

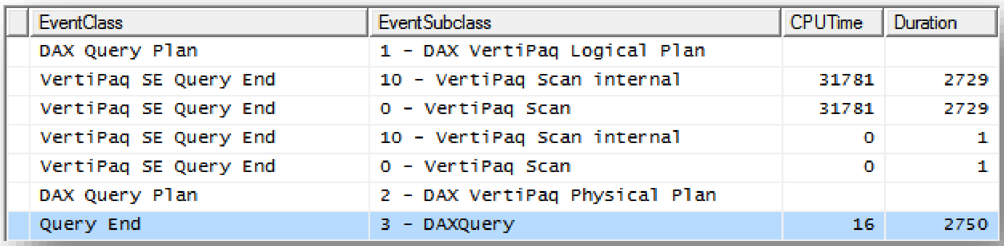
The initial query, using 2,742 milliseconds, is very similar to the DISTINCTCOUNT version of the same query but, this time, it grabs the CustomerCode and not the CustomerKey:

```
SELECT
    Customers.AgeRange,
    Customers.CustomerCode,
    COUNT()
FROM
    Fact
    LEFT OUTER JOIN Customers
        ON Customers.CustomerKey = Fact.CustomerKey
```

The second query gathers the AgeRanges from the fact table and FE performs the final join. The duration is too long, at 3,400 milliseconds, but you already know how to optimize it by using ADDCOLUMNS to avoid the additional scan of the fact table:

```
ADDCOLUMNS (
    VALUES ( Customers[AgeRange] ),
    "Result", CALCULATE ( COUNTROWS ( SUMMARIZE ( Fact, Customer[CustomerCode] ) ) )
)
```

Here is the query plan:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	31781	2729
VertiPaq SE Query End	0 - VertiPaq Scan	31781	2729
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	1
VertiPaq SE Query End	0 - VertiPaq Scan	0	1
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	16	2750

The initial query is identical to the previous one; the missing scan of the fact table (second scan is on the Customers table) makes the difference in duration. Still, it is slower than the DISTINCTCOUNT version. In reality, the DISTINCTCOUNT version became faster when we optimized the full query and, at the end, it had a shape very similar to the current one.

If you write the query in its best shape, it becomes:

```
ADDCOLUMNS (
    VALUES ( Customers[AgeRange] ),
    "Result", CALCULATE (
        COUNTROWS ( SUMMARIZE ( Customers, Customer[CustomerCode] ) ) ),
        Fact
    )
)
```

Again, by forcing DAX to work on the Customers table, we can obtain top performance. This query runs at the same speed as the best DISTINCTCOUNT query (near 1 second).

SUMMARIZE on Columns in Another Table

Using SUMMARIZE on columns in another table is a common usage of distinct counts. In fact, it is often the case that you have many dimensions and you want to produce the distinct count of customers by date, product model, or another dimension column.

In this test we focus on a simple scenario, where we use a single column from a different table. In the next example, we will look at improving performance in the more general case where we have many columns from many tables.

DISTINCTCOUNT

I will not spend much time explaining the query plans in detail, because the basic patterns are similar to queries we have already seen. In fact, if you do not force the engine to work with only the dimension, it always uses the fact table to gather values.

This is evident from these two queries:

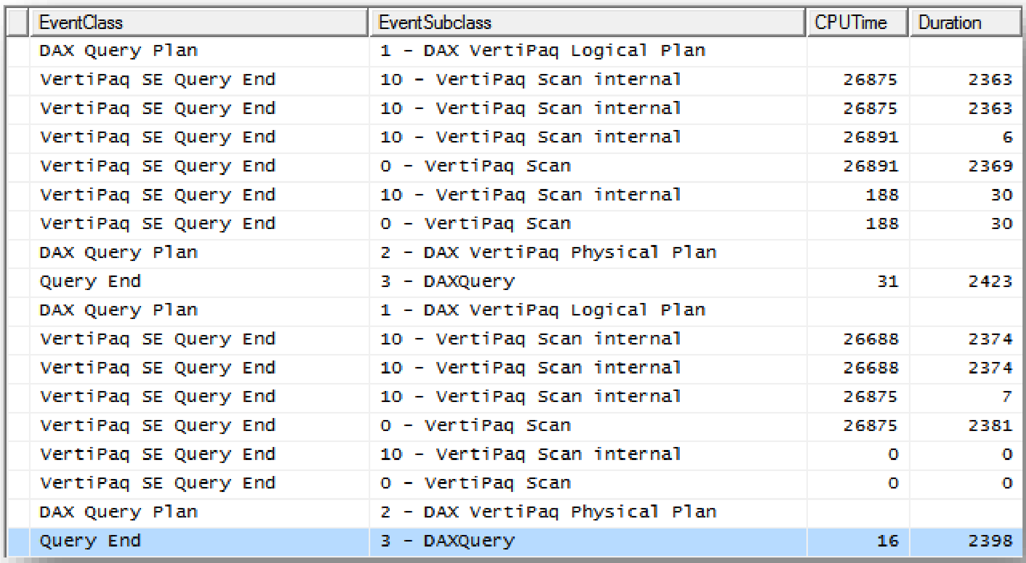
```
SUMMARIZE (
    Fact,
    Calendar[MonthName],
    "Result", DISTINCTCOUNT ( Fact[CustomerKey] )
)
```

And

```
ADDCOLUMNS (
    VALUES ( Calendar[MonthName] ),
    "Result", CALCULATE ( DISTINCTCOUNT ( Fact[CustomerKey] ) )
)
```

They produce identical query plans, following the pattern of running a first query that retrieves MonthName and CustomerKey, using $DCOUNT_DATACACHE to grab the distinct counts grouped by MonthName, and finally performing a JOIN of that result with the different month names from inside FE.

In the next figure, you can see the two queries executed one after the other. The single VertiPaq queries are identical, and the durations are only a few milliseconds apart.



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	26875	2363
VertiPaq SE Query End	10 - VertiPaq Scan internal	26875	2363
VertiPaq SE Query End	10 - VertiPaq Scan internal	26891	6
VertiPaq SE Query End	0 - VertiPaq Scan	26891	2369
VertiPaq SE Query End	10 - VertiPaq Scan internal	188	30
VertiPaq SE Query End	0 - VertiPaq Scan	188	30
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	31	2423
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	26688	2374
VertiPaq SE Query End	10 - VertiPaq Scan internal	26688	2374
VertiPaq SE Query End	10 - VertiPaq Scan internal	26875	7
VertiPaq SE Query End	0 - VertiPaq Scan	26875	2381
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	2398

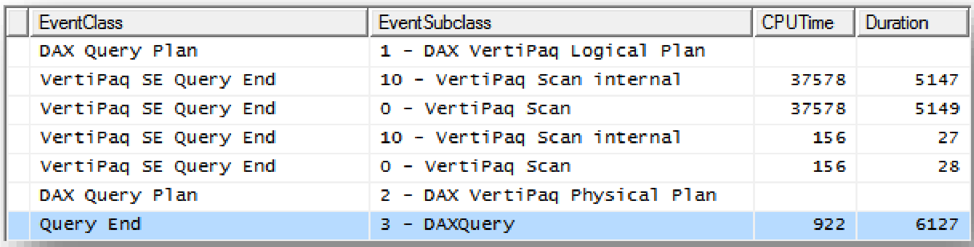
Thus, when working with columns from different tables, it seems that there is no option of using ADDCOLUMNS instead of SUMMARIZE to obtain better performance. This is somewhat expected—there is no way to resolve the distinct count using only a table.

SUMMARIZE

The DISTINCTCOUNT pattern is hard to optimize, but the SUMMARIZE version is even harder. For example, in a naïve query like the following one:

```
SUMMARIZE (
    Fact,
    Calendar[MonthName],
    "Result", COUNTROWS ( SUMMARIZE ( Fact, Fact[CustomerKey] ) )
)
```

The query plan is very simple, yet very slow:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	37578	5147
VertiPaq SE Query End	0 - VertiPaq Scan	37578	5149
VertiPaq SE Query End	10 - VertiPaq Scan internal	156	27
VertiPaq SE Query End	0 - VertiPaq Scan	156	28
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	922	6127

It takes 6,127 milliseconds to run the query. Moreover, 922 milliseconds are spent in the FE. Let us investigate on this query plan in order to understand what is happening. The heaviest SE query is the following one:

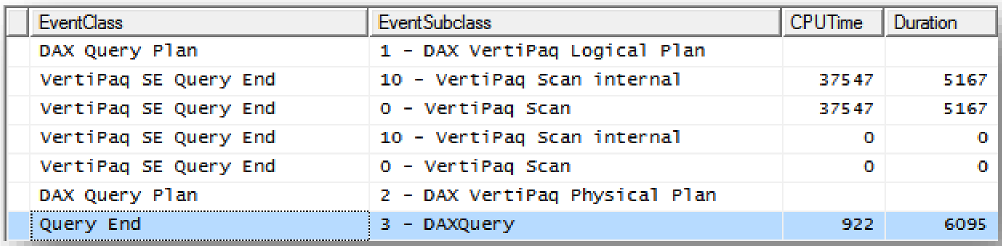
```
SELECT
    Calendar.MonthName,
    Fact.CustomerKey
FROM
    Calendar
  LEFT OUTER JOIN Fact
        ON Fact.CalendarKey = Calendar.CalendarKey
```

It returns, from the fact table, the name of the month and the customer keys. The second VertiPaq query computes the month names from the calendar table in a join with the fact table. Finally, FE performs the aggregation of the result of the first query by using COUNT, to compute the distinct values.

You can optimize the query by removing the 156 milliseconds from the second VertiPaq query by means of using ADDCOLUMNS:

```
ADDCOLUMNS (
    VALUES ( Calendar[MonthName] ),
    "Result", CALCULATE ( SUMMARIZE ( Fact, Fact[CustomerKey] ) )
)
```

The query plan shows that the second VertiPaq query is now much faster:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	37547	5167
VertiPaq SE Query End	0 - VertiPaq Scan	37547	5167
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	922	6095

However, those few milliseconds removed from the overall query plan do not make a substantial difference—the query is still very slow.

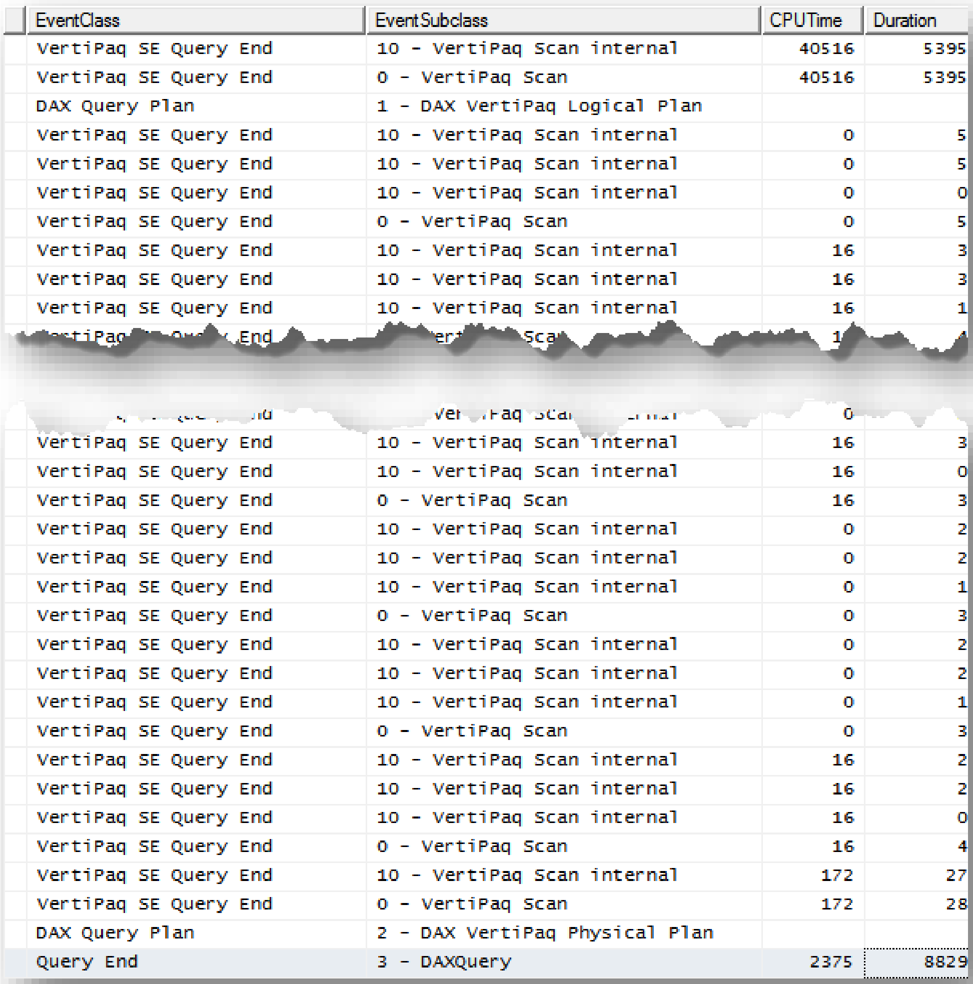
In this scenario, DISTINCTCOUNT is a clear winner: SUMMARIZE is not able to produce the same effect and, worse, it makes a heavy usage of FE, reducing both parallelism and cache options.

SCD Handling

Let us move on the more complex handling of SCD with both SUMMARIZE and DISTINCTCOUNT. We start with this query:

```
SUMMARIZE (
    Fact,
    Calendar[MonthName],
    "Result", CALCULATE ( DISTINCTCOUNT ( Customer[CustomerCode] ), Fact )
)
```

The query plan is somewhat long this time:



EventClass	EventSubclass	CPUTime	Duration
VertiPaq SE Query End	10 - VertiPaq Scan internal	40516	5395
VertiPaq SE Query End	0 - VertiPaq Scan	40516	5395
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	5
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	5
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	0
VertiPaq SE Query End	0 - VertiPaq Scan	0	5
VertiPaq SE Query End	10 - VertiPaq Scan internal	16	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	16	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	16	1
[illegible] End	[illegible] Sca	1	[illegible]
[illegible]	[illegible]	0	[illegible]
VertiPaq SE Query End	10 - VertiPaq Scan internal	16	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	16	0
VertiPaq SE Query End	0 - VertiPaq Scan	16	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	1
VertiPaq SE Query End	0 - VertiPaq Scan	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	1
VertiPaq SE Query End	0 - VertiPaq Scan	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	16	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	16	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	16	0
VertiPaq SE Query End	0 - VertiPaq Scan	16	4
VertiPaq SE Query End	10 - VertiPaq Scan internal	172	27
VertiPaq SE Query End	0 - VertiPaq Scan	172	28
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	2375	8829

Do you recognize the pattern? Iteration is happening. DAX is retrieving the CustomerKey and the MonthName from the first query. Then, for each set of CustomerKeys, it is computing the DISTINCTCOUNT from the customer table using the canonical pattern of $DCOUNT_DATACACHE, which is evident in the figure from the fact that there are three internal scans for each VertiPaq scan.

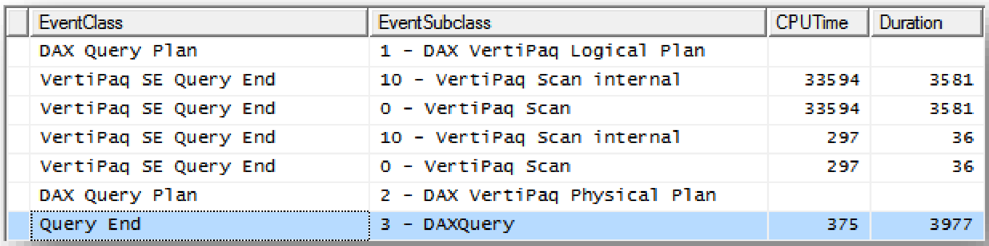
The execution time of 8,829 milliseconds is very high but, again, there is no easy way to force the distinct count to work on a single table because, this time, the fact table is a bridge and needs to be taken into account in a proper way.

Moreover, FE uses 2,375 milliseconds to perform its internal calculations, and we know this time will never be cached.

Let us take a look at the SUMMARIZE version now:

```
SUMMARIZE (
    Fact,
    Calendar[MonthName],
    "Result", COUNTROWS ( SUMMARIZE ( Fact, Customer[CustomerCode] ) )
)
```

The query plan is much shorter, as no iteration is happening:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	33594	3581
VertiPaq SE Query End	0 - VertiPaq Scan	33594	3581
VertiPaq SE Query End	10 - VertiPaq Scan internal	297	36
VertiPaq SE Query End	0 - VertiPaq Scan	297	36
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	375	3977

Interestingly, it took only 3,977 milliseconds, with 375 milliseconds of FE time. The first, and longest, query is the following:

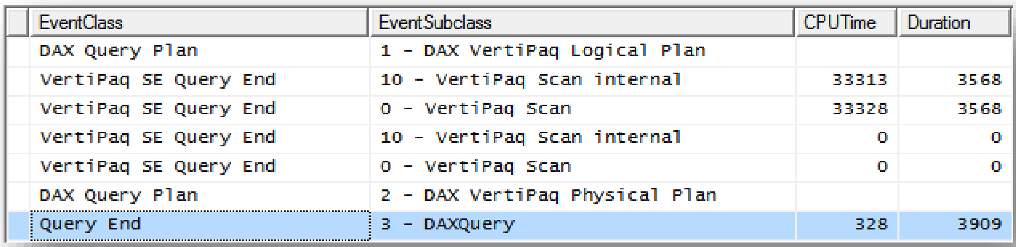
```
SELECT
    Calendar.MonthName,
    Customers.CustomerCode
FROM
    Fact
    LEFT OUTER JOIN Calendar
        ON Fact.CalendarKey = Calendar.CalendarKey
    LEFT OUTER JOIN Customers
        ON Customers.CustomerKey = Fact.CustomerKey
```

Can you see the big difference? Here, with a single VertiPaq query, the engine gathered the customer code and the month name from the fact table with a couple of joins to the dimensions. After that, computing the distinct count is only a matter of aggregating this result, which is the task performed by the FE in those 375 milliseconds.

Moreover, at this point we know that those 36 milliseconds of the second query are spent grabbing the month names from the fact table, and we can avoid them by forcing that scan to work on the dimension:

```
ADDCOLUMNS (
    VALUES ( Calendar[MonthName] ),
    "Result", CALCULATE ( COUNTROWS ( SUMMARIZE ( Fact, Customer[CustomerCode] ) ) )
)
```

The query plan of this last query shows the result:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	33313	3568
VertiPaq SE Query End	0 - VertiPaq Scan	33328	3568
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	328	3909

Now the second VertiPaq query time is gone because, this time, it is scanning the dimension and not the fact table. Not that we saved a huge amount of time but, still, it is better than nothing.

As we have already seen many times, DISTINCTCOUNT and SUMMARIZE perform very differently in different scenarios. In this one, where we have SCD handling with aggregation on a different table, the SUMMARIZE version runs twice as fast as DISTINCTCOUNT.

SUMMARIZE on Columns from Many Tables

Having read this far, you may already know what we will find using SUMMARIZE on columns from many tables. This pattern is similar to the previous one because, as we have seen, it is not easy to optimize the computation if the columns we use for the aggregation do not come from the table we use to compute the distinct count.

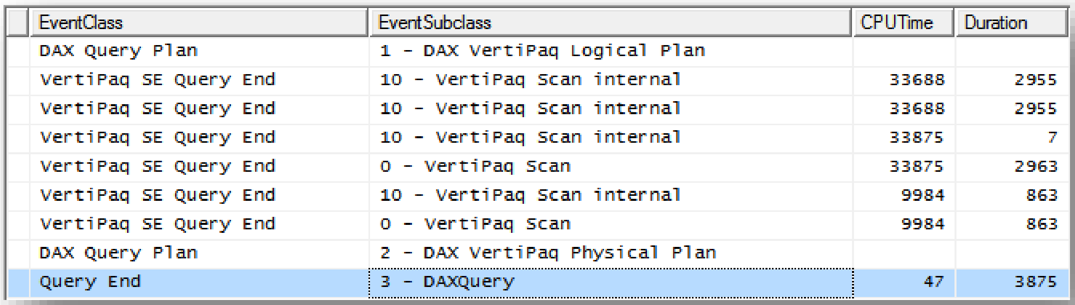
Nonetheless, we will try to optimize the queries when we aggregate from both columns coming from different tables and from columns belonging to the same table used to compute the distinct count.

DISTINCTCOUNT

The basic query pattern is the following:

```
SUMMARIZE (
    Fact,
    Calendar[MonthName],
    Customers[AgeRange],
    "Result", DISTINCTCOUNT ( Fact[CustomerKey] )
)
```

This time, we use one column from the Calendar table and one from the Customer table, aggregating the customer key from the fact table. Here is the query plan:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	33688	2955
VertiPaq SE Query End	10 - VertiPaq Scan internal	33688	2955
VertiPaq SE Query End	10 - VertiPaq Scan internal	33875	7
VertiPaq SE Query End	0 - VertiPaq Scan	33875	2963
VertiPaq SE Query End	10 - VertiPaq Scan internal	9984	863
VertiPaq SE Query End	0 - VertiPaq Scan	9984	863
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	47	3875

The whole query took 3,875 milliseconds, and the pattern should look familiar now. The first query returns month name, age range, and customer key from the fact table with a couple of joins:

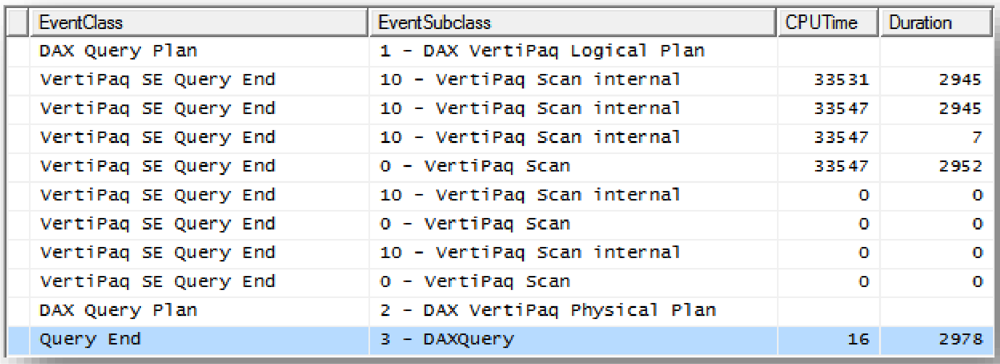
```
SELECT
    Calendar.MonthName,
    Customers.AgeRange,
    Fact.CustomerKey
FROM
    Fact
    LEFT OUTER JOIN Calendar
        ON Fact.CalendarKey = Calendar.CalendarKey
    LEFT OUTER JOIN Customers
        ON Customers.CustomerKey = Fact.CustomerKey
```

Then, it uses the $DCOUNT_DATACACHE to compute the distinct count. The second query retrieves the month name and age range from the fact table. Finally, FE performs the join between these two datasets to produce the result.

We can optimize it by forcing at least the second scan on the dimensions instead of using the fact table:

```
ADDCOLUMNS (
    CROSSJOIN (
        VALUES ( Calendar[MonthName] ),
        VALUES ( Customers[AgeRange] )
    ),
    "Result", CALCULATE ( DISTINCTCOUNT ( Fact[CustomerKey] ) )
)
```

And the query plan clearly shows the improvement:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	33531	2945
VertiPaq SE Query End	10 - VertiPaq Scan internal	33547	2945
VertiPaq SE Query End	10 - VertiPaq Scan internal	33547	7
VertiPaq SE Query End	0 - VertiPaq Scan	33547	2952
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	0
VertiPaq SE Query End	0 - VertiPaq Scan	0	0
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	2978

The first query is identical to the previous test but, this time, we removed 863 milliseconds that were necessary to scan the fact table searching for valid pairs of month name and age range. Of course, this latter query returns combinations that might not be needed, but a simple FILTER will remove them from the resulting set without affecting performance.

It is interesting to note that the CROSSJOIN is computed by separately scanning the two dimensions to gather column values. These two results are then CROSSJOINED, and the resulting set is joined with the result of the fact table scan.

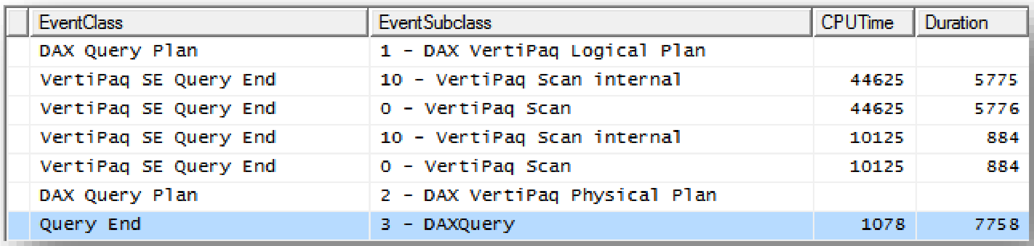
By using the ADDCOLUMNS / CROSSJOIN pattern, you might end up computing values for a large number of combinations that do not exist. As always, optimizing means performing tests to find the best combination of techniques that provide the top performance.

SUMMARIZE

How does the SUMMARIZE version perform in this scenario? Let us start with the naïve query:

```
SUMMARIZE (
    Fact,
    Calendar[MonthName],
    Customers[AgeRange],
    "Result", COUNTROWS ( SUMMARIZE ( Fact, Fact[CustomerKey] ) )
)
```

The query plan shows that performance is not so good:



EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	44625	5775
VertiPaq SE Query End	0 - VertiPaq Scan	44625	5776
VertiPaq SE Query End	10 - VertiPaq Scan internal	10125	884
VertiPaq SE Query End	0 - VertiPaq Scan	10125	884
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	1078	7758

The first VertiPaq query gathers the usual columns from the fact table:

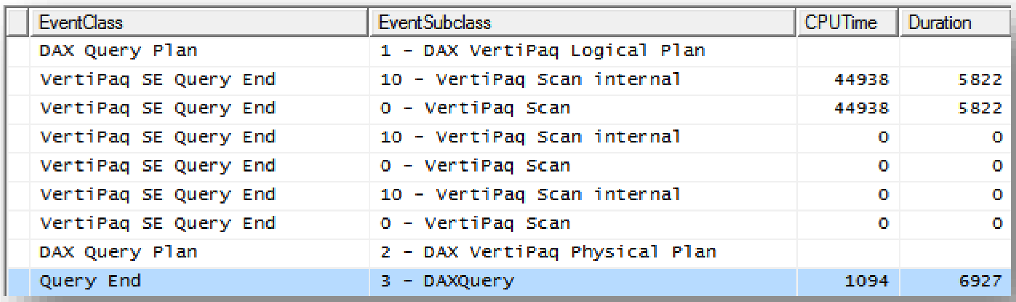
```
SELECT
    Calendar.MonthName,
    Customers.AgeRange,
    Fact.CustomerKey
FROM
    Fact
    LEFT OUTER JOIN Calendar
        ON Fact.CalendarKey = Calendar.CalendarKey
    LEFT OUTER JOIN Customers
        ON Customers.CustomerKey = Fact.CustomerKey
```

Yet this time, the query is taking a lot more time to grab the values. The query is identical to the one produced for the DISTINCTCOUNT calculation but, now, because of several optimizations that happen inside the engine for the DISTINCTCOUNT calculation, it turns out to be much slower than the DISTINCTCOUNT version. Explaining in deeper detail would lead us outside of the focus of this paper.

As shown previously, we can improve performance by removing the scan on the fact table:

```
ADDCOLUMNS (
    CROSSJOIN (
        VALUES ( Calendar[MonthName] ),
        VALUES ( Customers[AgeRange] )
    ),
    "Result", CALCULATE ( COUNTROWS ( SUMMARIZE ( Fact, Fact[CustomerKey] ) ) )
)
```

The query plan shows the improvement, removing the milliseconds used to scan the fact table for month names and age ranges:



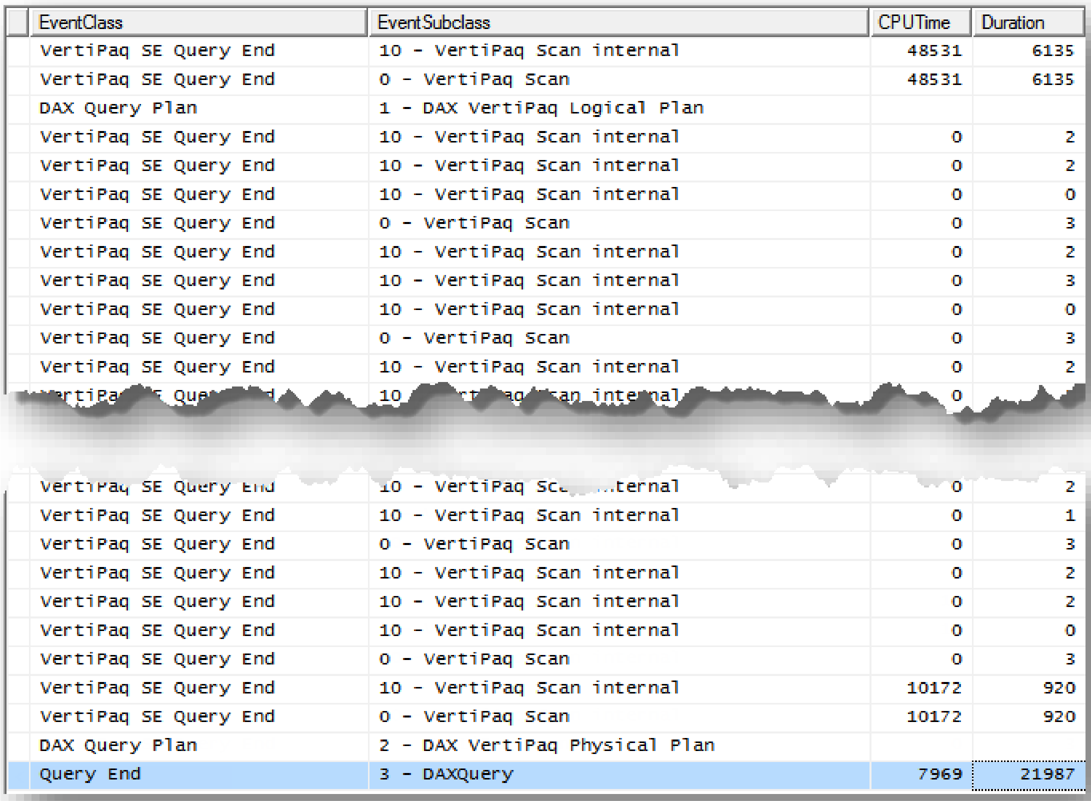
EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	44938	5822
VertiPaq SE Query End	0 - VertiPaq Scan	44938	5822
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	0
VertiPaq SE Query End	0 - VertiPaq Scan	0	0
VertiPaq 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	1094	6927

SCD Handling

It is now time to add SCD handling to our distinct count summarized on many columns. Let us start with the basic pattern for DISTINCTCOUNT:

```
SUMMARIZE (
    Fact,
    Calendar[MonthName],
    Customers[AgeRange],
    "Result", CALCULATE ( DISTINCTCOUNT ( Customers[CustomerCode] ), Fact )
)
```

This query generates a crazy query plan:



EventClass	EventSubclass	CPUTime	Duration
VertiPaq SE Query End	10 - VertiPaq Scan internal	48531	6135
VertiPaq SE Query End	0 - VertiPaq Scan	48531	6135
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	0
VertiPaq SE Query End	0 - VertiPaq Scan	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	0
VertiPaq SE Query End	0 - VertiPaq Scan	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	[illegible]
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	1
VertiPaq SE Query End	0 - VertiPaq Scan	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	2
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	0
VertiPaq SE Query End	0 - VertiPaq Scan	0	3
VertiPaq SE Query End	10 - VertiPaq Scan internal	10172	920
VertiPaq SE Query End	0 - VertiPaq Scan	10172	920
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	7969	21987

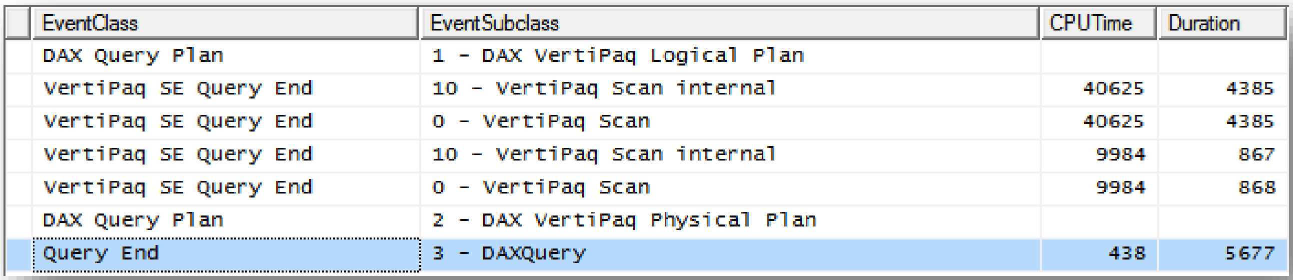
It iterates for each couple of values with very fast VertiPaq queries, but there are a lot of them and, finally, it uses 7,969 milliseconds of FE to perform the final join. The overall execution time of 21,987 milliseconds is huge. The pattern is the same as the DISTINCTOUNT on SUMMARIZE we have seen before, but this time the number of queries is much higher. Using ADDCOLUMNS instead of SUMMARIZE, in this scenario, does not change the plan significantly:

```
ADDCOLUMNS (
    CROSSJOIN (
        VALUES ( Calendar[MonthName] ),
        VALUES ( Customers[AgeRange] )
    ),
    "Result",
        CALCULATE (
            DISTINCTCOUNT ( Customers[CustomerCode] ),
            CALCULATETABLE ( Fact )
        )
)
```

So using the DISTINCTCOUNT over SCD does not look a good idea. But if we try the SUMMARIZE pattern for SCD handling on many columns, as shown below:

```
SUMMARIZE (
    Fact,
    Calendar[MonthName],
    Customers[AgeRange],
    "Result", COUNTROWS ( SUMMARIZE ( Customers, Customers[CustomerCode] ) )
)
```

The result is a much more appealing query plan:



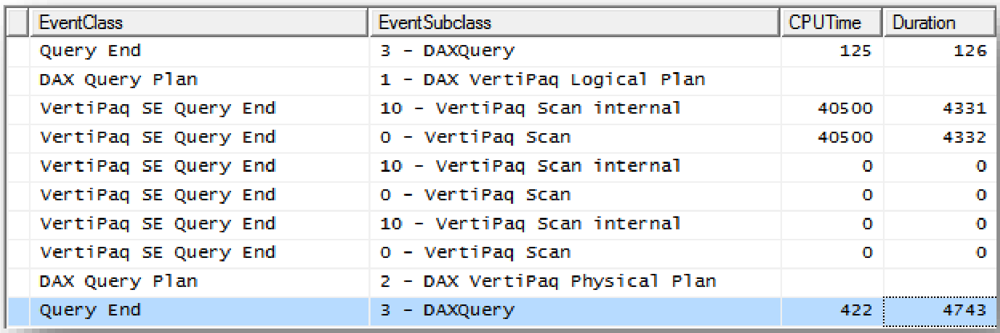
EventClass	EventSubclass	CPUTime	Duration
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	40625	4385
VertiPaq SE Query End	0 - VertiPaq Scan	40625	4385
VertiPaq SE Query End	10 - VertiPaq Scan internal	9984	867
VertiPaq SE Query End	0 - VertiPaq Scan	9984	868
DAX Query Plan	2 - DAX VertiPaq Physical Plan		
Query End	3 - DAXQuery	438	5677

There is nothing new here: the usual scan of the fact table to grab values following the joins in VertiPaq, then a second scan to get the values of month name and age range, again on the fact table, and finally a small amount of time spent in FE to perform the join.

As has often happened in this white paper, the ADDCOLUMNS version of the SUMMARIZE pattern looks even better:

```
ADDCOLUMNS (
   CROSSJOIN (
       VALUES ( Calendar[MonthName] ),
       VALUES ( Customers[AgeRange] )
   ),
   "Result", CALCULATE ( COUNTROWS ( SUMMARIZE ( Customers, Customers[CustomerCode] ) ) )
)
```

The query plan shows, as usual, the elimination of the scanning of the fact table for the values of month name and age range:

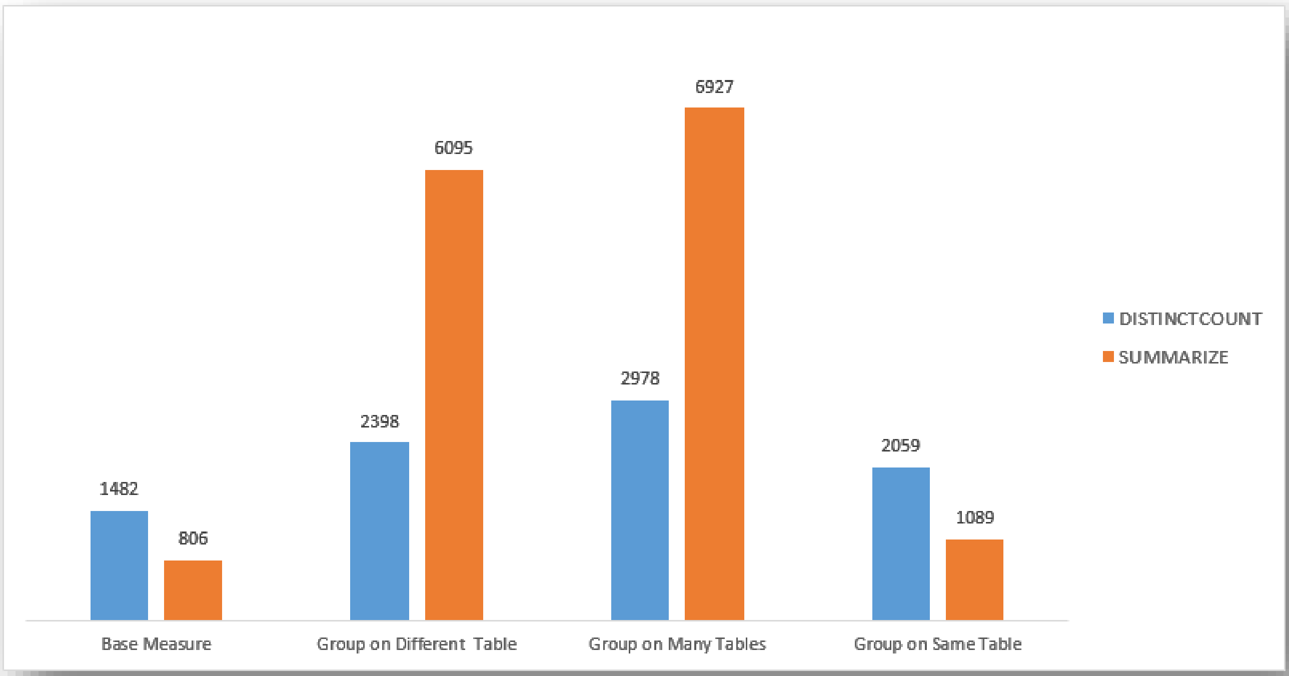


EventClass	EventSubclass	CPUTime	Duration
Query End	3 - DAXQuery	125	126
DAX Query Plan	1 - DAX VertiPaq Logical Plan		
VertiPaq SE Query End	10 - VertiPaq Scan internal	40500	4331
VertiPaq SE Query End	0 - VertiPaq Scan	40500	4332
VertiPaq SE Query End	10 - VertiPaq Scan internal	0	0
VertiPaq SE Query End	0 - VertiPaq Scan	0	0
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	422	4743

Conclusions

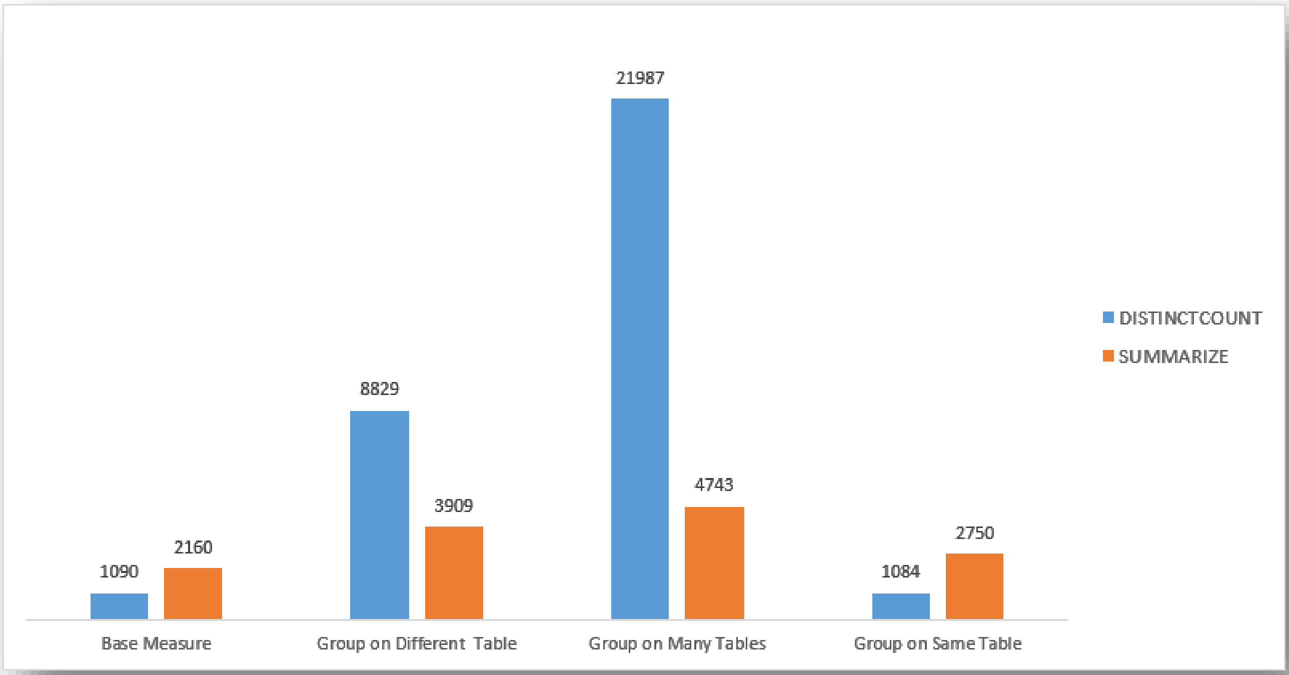
Well, after a long journey through query plans and different patterns for distinct counts, it is now time to draw some conclusions.

The following charts show the best time for each pattern (that is, the most optimized formula I have found using either DISTINCTCOUNT OR SUMMARIZE). If you do not care about SCD handling, then this is the situation:



For the base measure or for grouping on the same table, SUMMARIZE is more than twice as fast. But as soon as you try to group on many tables, which is the most common scenario, then DISTINCTCOUNT is faster. DISTINCTCOUNT provides performance that does not change much from scenario to scenario; its consistency provides the end user with a good experience.

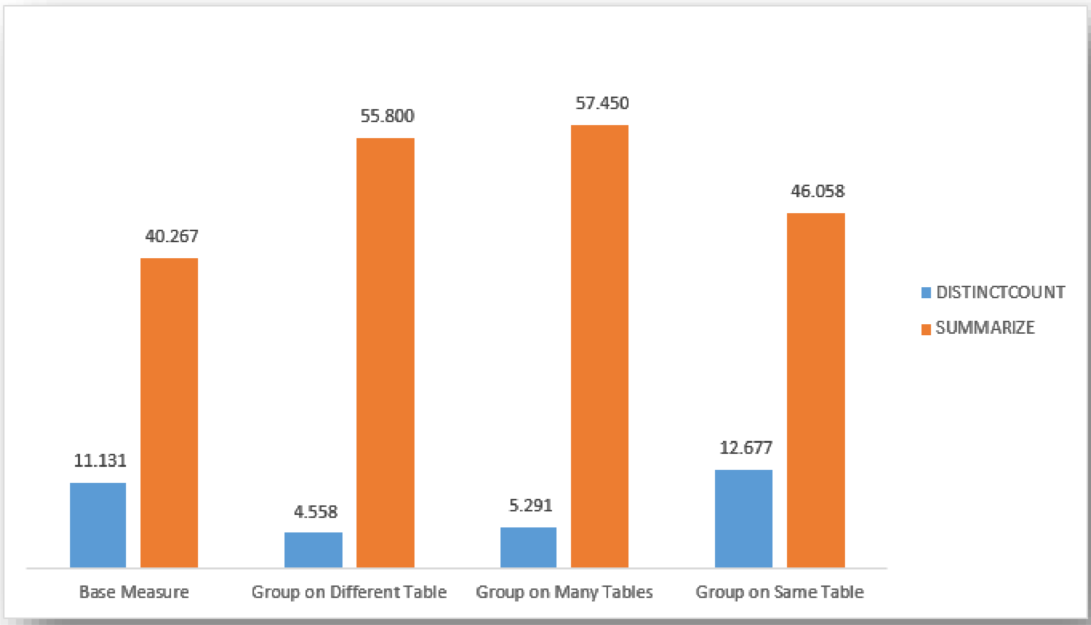
If you need SCD handling, however, then the scenario is totally different:



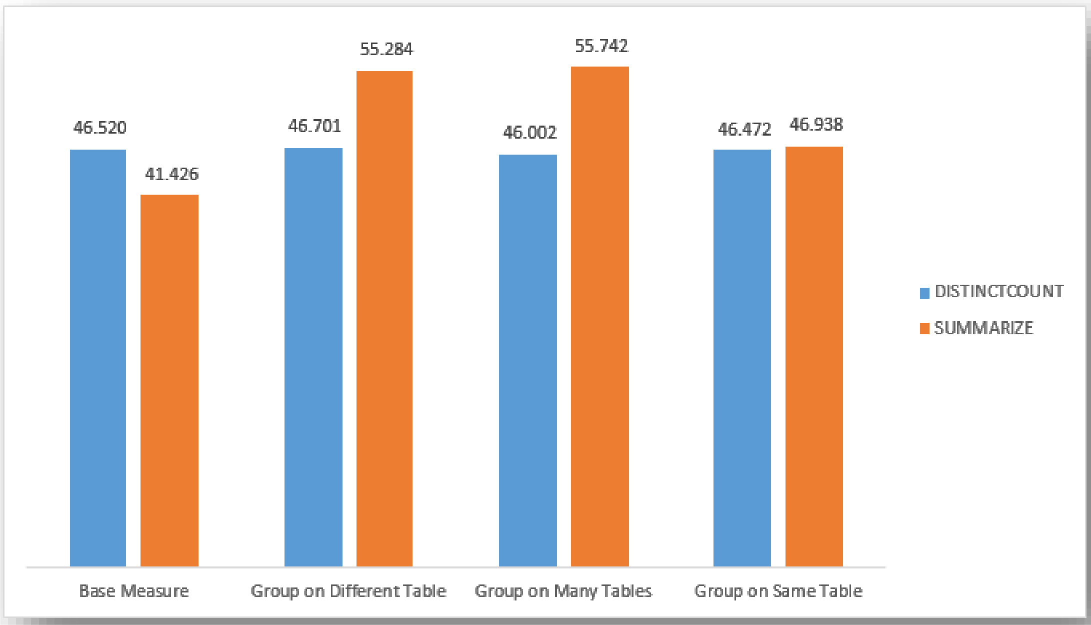
When SCD handling comes into play, DISTINCTCOUNT wins for the base measure and for grouping on the same table, but it is so slow on grouping on different or many tables that it is simply not an option.

Now, the big question is “which pattern should I follow?” The answer, as often happens, is “it depends.” The goal of this paper has never been to show the best pattern for computing distinct values in DAX. In fact, the title is “Understanding Distinct Count in DAX Query Plans,” which is a completely different topic.

As an example, I show below the measurements taken on a different database, with a different distribution of values and a different number of distinct counts for the involved columns. First is the chart without SCD handling:



And next is the chart with SCD handling:



As you can see, the results are very different. In fact, on this database, there is no gain in using the SUMMARIZE version of the DISTINCTCOUNT calculation.

I wanted to show that by using query plans you have the option of understanding the algorithm used by the DAX engine to retrieve values, as well as to show how, by adapting your code to the specific scenario, you can often get great results. For each pattern and for each scenario, there is a huge difference between the worst and the best choice, such as between the naïve query and the one that required some optimization effort.

The best advice I can give you, at this point, is the following: test performance with your data. If you can control the shape of your queries, spend time optimizing them. If you cannot, then go for the pattern that provides the best overall performance, knowing that you will not have a perfect formula, but will have the best you can afford.

What you cannot afford to do is perform some testing on the base measure and then think that your job is done. As we have demonstrated in this paper, the behavior of the base measure is not a good indicator of how well your measure will perform when included in a complex query.

As a final note, remember that we have only scratched the surface. An important part of our job is optimizing customers' data, and we have found many times that the same query, on different datasets, with different data distribution, leads to completely different results.

We cannot overstress this simple fact: DAX can provide astonishing performance, but you need to test the queries on the data, understand the query plan, and find the best formulation of the query for your specific database. Our hope is that, by reading this paper, you now have more weapons to optimize your DAX.