VertiPaq vs ColumnStore

PERFORMANCE ANALYSIS OF THE XVELOCITY ENGINE

produced by

sqlbi

VertiPaq vs ColumnStore

Performance Analysis of the xVelocity Engine

Author: Alberto Ferrari

Published: Version 1.0 Revision 2 – August 3, 2012

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

Summary: This paper shows a performance analysis of the two flavors of the xVelocity engine, introduced in SQL 2012 in both SQL Server and SQL Server Analysis Services. During the performance analysis, we sometime indulge in considerations about how to improve performances of the two engines

Acknowledgments: we would like to thank the many peer reviewers that helped us to improve this document: Hrvoje Piasevoli, Darren Gosbell and all of our ssas-insiders friends.

We would also like to give a special thank to Amina Saify, Hugo Kornelis, 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

At the end of June 2012, I was in Amsterdam to present some sessions at Teched Europe 2012 and, while preparing the material for the demos (yes, the best demos are the ones I prepare at the last minute), I decided to make a comparison between the two implementations of xVelocity of SQL 2012, one is the VertiPaq engine in SSAS Tabular and the other one is the ColumnStore index in SQL Server. After some trials, I decided that ColumnStore was a clear loser, because I was not able to see a real improvement in performance for my test queries, involving complex calculations and many-to-many relationships.

But, hey, I am not a SQL expert, I work mostly on the BI side, might it be the case that I don't know how to write a query that takes advantage of ColumnStore?

Being in a place full of SQL gurus from Microsoft and MVPs, I decided to ask for help. Hugo Kornelis was there and I had a chance to talk with him about ColumnStore performance. It turned out that he perfectly knew how to improve the performance of my queries and I was amazed to see that, after his intervention on SQL code, the two engines did run at a comparable speed. Moreover, sometimes ColumnStore was faster than VertiPaq. I suddenly decided to investigate more on the topic. I did not have time to prepare the material for the demos, but, at that point, this whitepaper was born.

At the end, I discovered (many thanks Hugo!) that the first implementation of the ColumnStore indexes in SQL2012 is not perfect and hopefully will be improved in the future. There are several limitations that reduce the effectiveness of ColumnStore indexes, as stated in the following post: http://msdn.microsoft.com/en-us/library/gg492088.aspx. For this reason, the SQL code will sometimes look strange, because we will need to write SQL code that is optimized for the ColumnStore index usage. Nevertheless, once you know how to write them, queries are pretty easy.

Well, enough words for an introduction, it is now time to join me in this amazing trip on performance analysis of two incredible query engines! Hope you will have fun reading, as I had in writing.

DIFFERENT FLAVORS OF XVELOCITY:

In SQL 2012 the xVelocity engine comes in two different flavors:

- **xVelocity in-memory analytics engine**, also known as Vertipaq, is the in-memory engine that runs inside Analysis Services 2012 for Tabular models
- **xVelocity memory optimized ColumnStore index**, is the same technology implemented in the SQL Server engine, in the form of non-clustered columnar indexes

There are some important differences in the two implementations but, for the sake of simplicity, we will refer to both as xVelocity implementation, because both share the same core technology. Moreover, in order to avoid using too long names, we will refer to them as:

- **VertiPaq**, for the xVelocity in-memory analytics engine
- **ColumnStore**, for the xVelocity memory optimized ColumnStore index

In this paper, we are interested in performing an investigation of the performance of the two technologies, comparing the performance of both in similar queries. I assume that the readers of this paper already have

a good understanding of both xVelocity technologies and are familiar with data warehousing, relational and dimensional modeling, SQL Server, the SQL language, the DAX language and the basics of column storage. We don't spend time explaining how the queries work, some of them are pretty complex and, understanding them, is a good exercise by itself. Nevertheless, even less experienced readers can take away the conclusions by simply following the text and ignoring the technical details of the queries.

Because this is the first paper on this topic, don't expect to draw any final conclusion from this paper alone. The goal is to start some investigation and poke your curiosity on this technology, stimulating you to run more tests and analyses.

NOTES ABOUT PERFORMANCE MEASUREMENTS

The focus of this paper is to evaluate performance in a typical data warehouse environment (i.e. queries on large datasets) over relatively simple sets of relationships with reasonable datasets as a result of a complex aggregation. When we say "reasonable" dataset, we mean that there is no point in analyzing queries that returns thousands of rows as a result, because users are normally interested in reports that can be analyzed quickly. Thus, the focus is on the power of computation and aggregation: several billions rows will be summed up in order to retrieve tens of rows as a result.

Finally, when it comes to measure the execution time of queries, we are definitely not interested in milliseconds. We will use the second as the unit of measure because, from the user point of view, there is really no difference between a query that returns in one, two or three seconds when the query aggregates two billions of rows. Even if we completely understand that sometimes a huge work is needed in order to improve the performance of a single query reducing its execution time from 1 second to 0.5 seconds, this is not interesting from the point of view of this document. A query that returns in one second is simply categorized as a "fast" one and if the same query can execute in 0.5 seconds, we consider this improvement as irrelevant. Again, please consider that the focus is on data warehouses only.

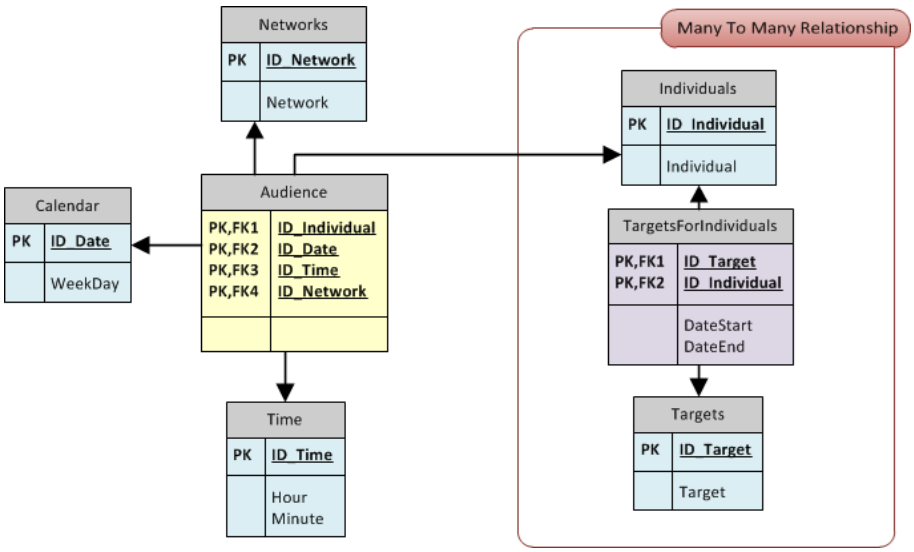
DISCLAIMER

A whitepaper cannot avoid its disclaimer section! Nothing formal here, but I feel the need to state very clearly that the results shown in this paper are measurements made on a very specific database and without the preciseness that would be needed to draw final conclusions. What I wanted to do is to raise curiosity in the BI world about these two engines. Thus, your mileage may vary a lot and the numbers shown here have no official meaning. These numbers cannot be used in any formal communication to show the difference in performance between the two engines. Said in other words, make your tests, don't trust mine and don't deduce, from this document, any conclusion about the two engines.

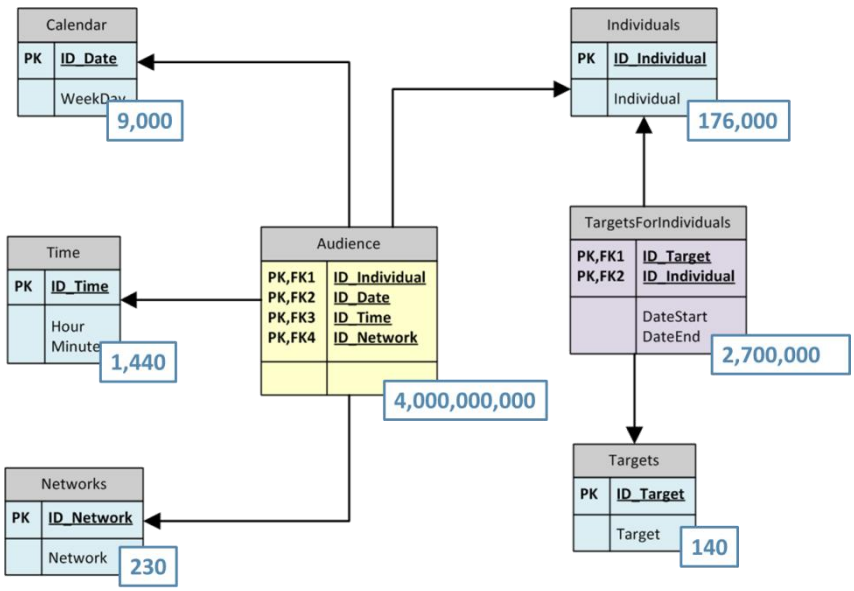
The data model

In order to perform tests, we used a data model with a big fact table and a many-to-many relationship, so to have a good environment to make interesting considerations.

The data model is useful to compute audience analysis of television broadcasts and a simplified version of the data model is the following one:



When it comes to the size of the tables, the biggest one is the fact table, containing more or less 4 billion rows. All the other tables are relatively small:



Inside the fact table, for each individual, there are two columns that we will use, which are Weight and Age. Weight is the “importance” of the individual on that specific date, Age is the age of the individual in that point in time.

The database is not public domain, so please don't ask about downloading it to make your own tests. Trust the results and, if you have a suitable database to make your own trials, use the same techniques on your data, you should observe the same level of performance for each technique. I did not even try to use AdventureWorks to perform any tests, the set of data is so small that no serious measurement can be taken from there.

Query Analysis

Here we start analyzing the performance of the two engines, starting with very simple queries and moving toward ones that are more complex. As you are going to learn, the two engines behave in different ways depending on the complexity of the query.

At the end of this chapter we will summarize all the different queries in a table. Thus, the reader who is eager to look at conclusions can jump to the end, read the conclusions and then dive into the details of what is more interesting for his needs.

THE TEST HARDWARE

The hardware used to test the queries is an 8-core server with 48GB of RAM. SQL Server has 16GB of memory allocated, which is enough to hold the columnar indexes in memory. VertiPaq has enough memory to run all the queries and hold the database.

Thus, both SQL Server and Analysis Services will answer to all of the queries without ever accessing the disk and this has been verified by monitoring disk activity during the tests. The SQL Server queries were executed after warming the cache, so that the necessary pages of the index are already in memory. On the other hand, all DAX queries were executed after clearing the cache, because, otherwise, the VertiPaq cache would return all of them without really executing the query.

The Simplest Aggregation

The simplest query we can run is the SUM of a single column scanning the complete fact table. The DAX query that returns such a value is the following one:

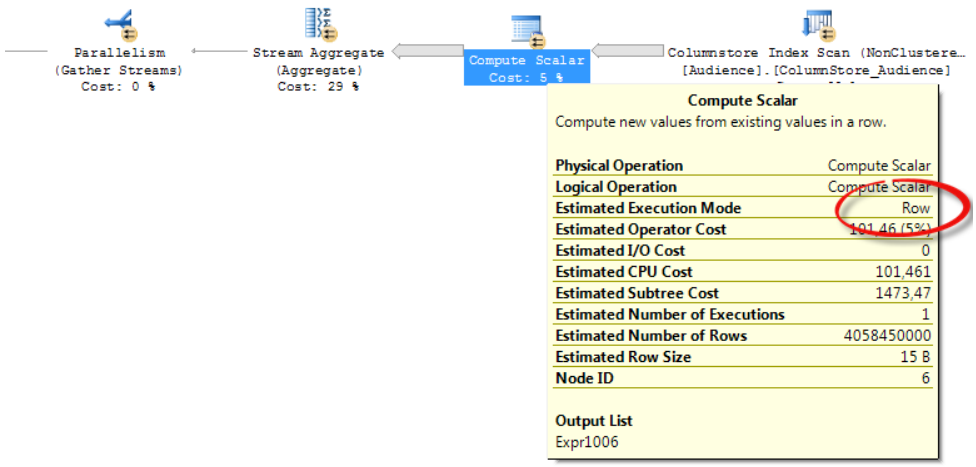
```
EVALUATE
    ROW (
        "Result",
        FORMAT (SUM (Audience[Weight]), "#,#")
    )
```

In VertiPaq this query is very fast, executing in about one second. During the computation, all of the cores are running at 100%. This clearly reflects the power of the Vertipaq engine, a full table scan of 4 billion rows runs in less than one second. There is really nothing we can do to further optimize this query, just because it is too simple to leave space for any optimization.

In SQL Server we have tried this query first:

```
SELECT
    FORMAT (SUM (CAST (Weight AS BIGINT)), '#,#')
FROM
    Audience
```

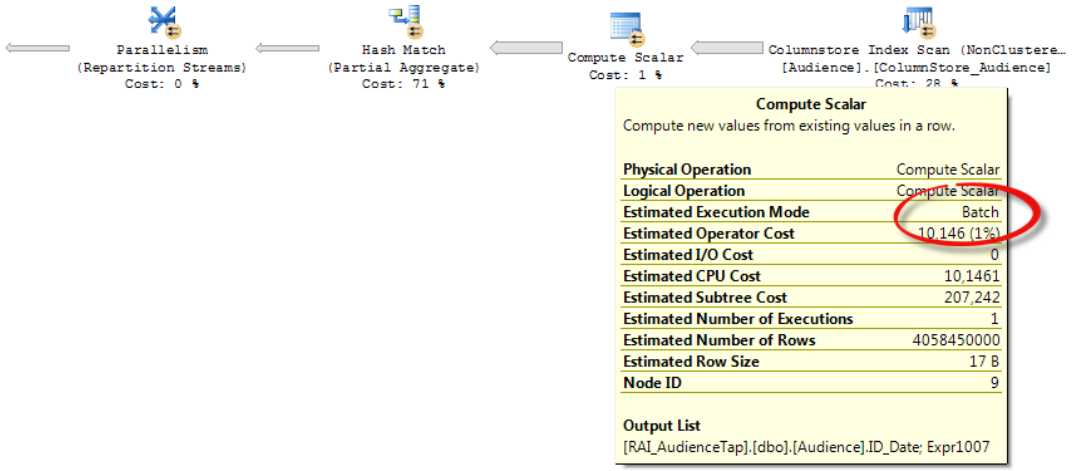
Performances are awful (1 minute and 17 seconds to answer). The reason is that with such a simple query SQL Server does not take advantage of the columnar index and operates in row mode. This can be easily recognized by looking at the query plan:



To obtain good performance in SQL Server, the query needs to be written in a slightly complex way, using a dummy GROUP BY in order to force batch mode in the aggregation operator.

```
SELECT
    FORMAT (SUM (Subtotal), '#,#')
FROM
    (SELECT
        Subtotal = SUM (CAST (Weight AS BIGINT))
     FROM
        Audience
     GROUP BY
        ID_Date) A
```

The query plan of this query clearly shows batch mode working:



In this scenario, Vertipaq is a clear winner. This query runs for 5 seconds before returning the single row result and, during this period, all of the cores are used at 100%. It is useful to note that, in SQL, we needed to do a CAST of the column to a BIGINT in order to compute the values, otherwise the query would raise an out-of-range error.

It is too early to derive any conclusion from these results. This query is useless, returning a single value that does not have a concrete meaning. Yet, it is a good baseline to start investigating the differences between the two engines.

Aggregation in the same table

The first useful query we can try is a simple aggregation of a single measure over a column that belongs to the fact table. Thus, no JOIN is needed, only a scan of the fact table and a subsequent GROUP BY over one of its columns. We have chosen an aggregator column that result in only 240 rows. Thus, this test aggregates 4 billion rows into 240 aggregated values using SUM, with no joins involved.

```
EVALUATE
    SUMMARIZE (
        Audience,
        Audience[ID_Network],
        "Result", FORMAT (SUM (Audience[Weight]), "#,#")
    )
    ORDER BY Audience[ID_Network]
```

On VertiPaq, this query is very fast, executing in 2 seconds. Because there are no calculations and no join to follow, the VertiPaq engine uses all the cores at 100% while running the query.

It is worth to note that the following query runs in a very similar way, returning the same results. We will see that, depending on the shape of the query, moving from SUMMARIZE to ADDCOLUMNS can dramatically change the results. As of now, it is enough to understand the difference in the two versions of the same algorithm.

```
EVALUATE
    ADDCOLUMNS (
        VALUES (Audience[ID_Network]),
        "Result", FORMAT (CALCULATE (SUM (Audience[Weight])), "#,#")
    )
    ORDER BY Audience[ID_Network]
```

The same query on the ColumnStore index is the following one:

```
SELECT
    ID_Network,
    Subtotal = FORMAT (SUM (CAST (Weight AS BIGINT)), '#,#')
FROM
    Audience
GROUP BY
    ID_Network
ORDER BY
    ID_Network
```

This query runs with all of the cores at 100% for 5 seconds before returning the values. It is interesting to note that ColumnStore took the same time to compute a total and a group by, indicating that the time of 5 seconds seems the time required to perform a full scan of the index, whereas the GROUP BY operator seems to use very few resources in order to be executed. After all, we needed to add a dummy GROUP BY even on the previous query in order to let the ColumnStore engine kicks in. This time, because the GROUP BY is natively present in the query, no dummy is required.

Aggregation and Filtering on the same Table

Up to now, we have always queried the full table. This is not a fair test, because data warehouse queries normally apply some kind of filtering to the full table. Thus, in order to make a filtering test, we decided to filter a single month out of the 4 years of data that are present in the fact table.

The following queries are identical to the previous ones but, this time, we added a filter on the ID_Date column that performs the analysis on a single month.

```
EVALUATE
    CALCULATETABLE (
        ADDCOLUMNS (
            VALUES (Audience[ID_Network]),
            "Result", FORMAT (CALCULATE (SUM (Audience[Weight])), "#,#")
        ),
        Audience[ID_Date] >= 8092 && Audience[ID_Date] <= 8119
    )
    ORDER BY Audience[ID_Network]
```

The VertiPaq engine takes more or less one second to answer, the speed is comparable with the full table scan and, again, there is no difference between the same query using ADDCOLUMNS or SUMMARIZE.

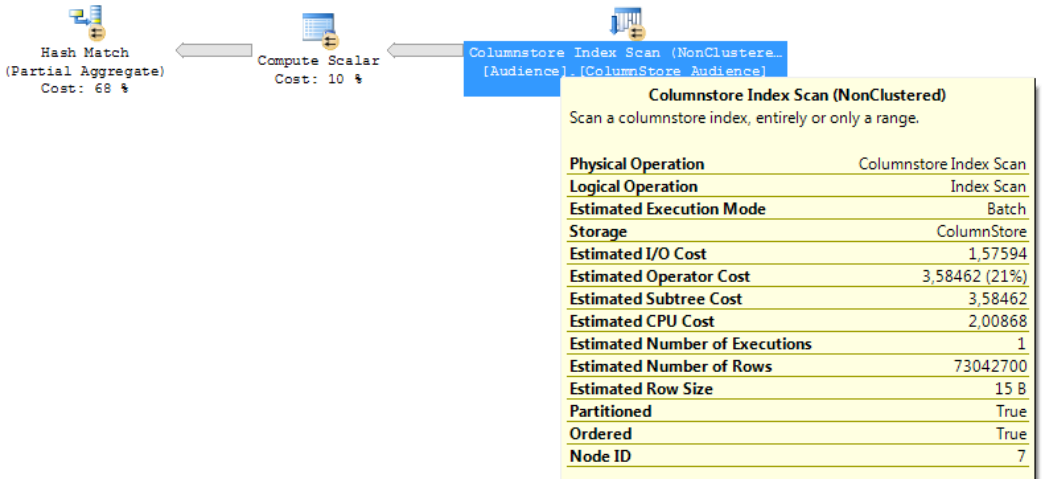
The real surprise is the ColumnStore index, which clearly leverages a very good filtering algorithm.

```
SELECT
    ID_Network,
    Subtotal = FORMAT (SUM (CAST (Weight AS BIGINT)), '#,#')
FROM
    Audience
WHERE
    ID_Date >= 8092 AND ID_Date <= 8119
GROUP BY
    ID_Network
ORDER BY
    ID_Network
```

This query runs in less than one second, which is a great improvement if compared with the five seconds needed to scan the full table. In order to get full information about the real improvement, we would need to analyze numbers below one second but, as we said at the beginning of this document, a response time around 1 second is generally categorized as a “fast” one and we are not interested in very precise results.

Things are getting more interesting now, because the two engines show a comparable speed. It is time to dive into different tests.

We might be tempted to think that the boost in performance of the ColumnStore technology depends from the fact that the filter has been applied over the first column of the clustered index of the table. This is not true, as it can be seen by the query plan, which uses only the ColumnStore index:



In order to remove all the doubts about it, we changed the query grouping by date and filtering over ID_Network, so that the clustered index is of no help here. Let us see the DAX query:

```
EVALUATE
    CALCULATETABLE (
        ADDCOLUMNS (
            VALUES (Audience[ID_Date]),
            "Result", FORMAT (CALCULATE (SUM (Audience[Weight])), "#,#")
        ),
        Audience[ID_Network] = 2
    )
    ORDER BY Audience[ID_Date]
```

And the SQL one

```
SELECT
    ID_Date,
    Subtotal = FORMAT (SUM (CAST (Weight AS BIGINT)), '#,#')
FROM
    Audience
WHERE
    ID_Network = 2
GROUP BY
    ID_Date
ORDER BY
    ID_Date
```

Both queries execute in one second, confirming the huge boost in the performance of ColumnStore when filtering happens, regardless of the column used to perform the filtering. Needless to say, the query plans are absolutely identical. Next time we have a doubt, we will simply trust the query plan! ☺

This behavior of ColumnStore is interesting because it means that you can obtain incredible performances if you do not need to scan the entire fact table, which is a very common scenario. In this case, since ColumnStore does not need to hold the full table in memory, you will be able to run complex queries against huge tables without having a super-server at hand.

Aggregations and Joins

All of the previous queries used a single table as the source for the data. We decided to start from there, to get a first baseline of measurements and draw the first conclusions. Nevertheless, any interesting query in the data warehouse normally uses many joins between the fact table and the dimensions. Filters are imposed on dimensions and are then propagated to the fact table.

The next queries will introduce simple one-to-many joins between the fact table and the dimensions. In Tabular, these join are part of the data model and, in order to use them, it is enough to put a filter on the dimension: the propagation of the filter is automatically handled by the engine. In SQL, we will need to manually set the filters inside the query. The DAX code is very simple:

```
EVALUATE
    CALCULATETABLE (
        SUMMARIZE (
            Audience,
            Individuals[AgeRange],
            "Result", FORMAT (CALCULATE (SUM (Audience[Weight])), "#,#")
        ),
        'Date'[Year] = 2011,
        Time[Period60Minutes]= "08:00 - 08:59",
        Individuals[Gender] = "Male"
    )
    ORDER BY Individuals[AgeRange]
```

This code uses SUMMARIZE. SUMMARIZE is not the evil, but it is mainly useful to perform complex aggregations and subtotals. In this case, the ADDCOLUMNS version is slightly faster. Generally speaking, ADDCOLUMNS is better whenever you do not need the full power of SUMMARIZE.

```
EVALUATE
    CALCULATETABLE (
        ADDCOLUMNS (
            VALUES (Individuals[AgeRange]),
            "Result", FORMAT (CALCULATE (SUM (Audience[Weight])), "#,#")
        ),
        'Date'[Year] = 2011,
        Time[Period60Minutes]= "08:00 - 08:59",
        Individuals[Gender] = "Male"
    )
    ORDER BY Individuals[AgeRange]
```

And the corresponding SQL query is not much more complicated, apart from the need to indicate the JOINs to use:

```
SELECT
    I.AgeRange,
    Subtotal = FORMAT (SUM (CAST (Weight AS BIGINT)), '#,#')
FROM
    Audience A
    INNER JOIN Date D ON A.ID_Date = D.ID_Date
    INNER JOIN Individuals I ON I.ID_Individual = A.ID_Individual
    INNER JOIN Time T ON T.ID_Time = A.ID_Time
WHERE
    D.Year = 2011 AND I.Gender = 'Male' AND T.Hour = '20'
GROUP BY
    AgeRange
ORDER BY
    AgeRange
```

Both queries execute a scan of the fact table using filters that are created on the dimensions, resulting in a typical star join operation.

In terms of performance, the results are interesting: the DAX code runs in 1.5 seconds, whereas the SQL code runs in 1 second. The difference between the two queries is negligible. It is worth pointing out that this query does not perform complex calculations and returns a small set of rows from the fact table, so it is a perfect candidate for SQL to perform better. The interesting point is that SQL Server is very strong in processing JOINs on ColumnStore tables and seems to be able to use dimensions to filter fact tables in a very efficient way, using the STAR JOIN algorithm.

Clearly, we expect STAR JOIN to be natively implemented in VertiPaq, due to the very nature of the SSAS engine, but it is very interesting to see that, as the query becomes more complicated, the SQL engine really shines.

Distinct Count

With the next set of queries, we want to test the performances of distinct counts using both ColumnStore and VertiPaq. We already know that distinct counts are slow in SQL and were a big pain in Multidimensional, whereas they are now super-fast in Tabular thanks to VertiPaq. Did ColumnStore gain the same boost in performance because it is using the same technology?

Because Individuals is a slowly changing dimension (SCD) of type 2, in order to compute the distinct count we need to make the computation on the natural key, not on the surrogate key. Again, this would be hard to compute in Multidimensional, but both ColumnStore and Vertipaq behave very well in this scenario.

Let us start with the DAX query:

```
EVALUATE
    CALCULATETABLE (
        ADDCOLUMNS (
            CROSSJOIN (
                VALUES ('Date'[Year]),
                VALUES (Individuals[Gender]),
                VALUES (Individuals[AgeRange])
            ),
            "Num Of Individuals",
                CALCULATE (
                    COUNTROWS (SUMMARIZE (Audience, Individuals[COD_Individual]))
                )
        ),
        'Date'[Year] = 2010
    )
ORDER BY
    'Date'[Year],
    Individuals[Gender],
    Individuals[AgeRange]
```

As you can see, the complexity of the query comes from the need to compute the distinct count over the natural key of the individual but, after studying the query, you will find it simple. Here we are computing the distinct count of individuals over a full year of data (more or less one billion rows) and doing a group by Year, Gender and AgeRange. The query runs at an excellent speed of 1.5 seconds, an unbelievable result if you compare with the same data model in multidimensional.

Again, it is worth to note that if you express the same query using SUMMARIZE, you get much worse performance. The following query, returning the same dataset, runs in three seconds.

```
EVALUATE
    CALCULATETABLE (
        SUMMARIZE (
            Audience,
            'Date'[Year],
            Individuals[Gender],
            Individuals[AgeRange],
            "Num Of Individuals",
                COUNTROWS (SUMMARIZE (Audience, Individuals[COD_Individual]))
        ),
        'Date'[Year] = 2010
    )
ORDER BY
    'Date'[Year],
    Individuals[Gender],
    Individuals[AgeRange]
```

As the query starts to become complex, DAX requires us to optimize it, using the knowledge we have of the internal implementation of the engine. The DAX era is just at its beginning, so it is not surprising that the optimization of DAX still requires some trial and error, after all no book is yet available about how to optimize DAX.

It is worth to note that the SUMMARIZE version of the query does not need CALCULATE to surround the COUNTROWS, whereas the ADDCOLUMNS one needs it. If this does not seem trivial… well, it might be the case that you want to study a bit more the interaction between CALCULATE and the evaluation contexts, I still need to think twice whenever I write these queries!

VertiPaq is not the only engine that surprised us. The same query, expressed in SQL, is the following one:

```
SELECT
    D.Year,
    I.Gender,
    I.AgeRange,
    NumOfIndividuals = COUNT(DISTINCT (I.COD_Individual))
FROM
    Audience A
    INNER JOIN Date D
        ON A.ID_Date = D.ID_Date
    INNER JOIN dbo.Individuals I
        ON A.ID_Individual = I.ID_Individual
WHERE
    D.Year = 2010
GROUP BY
    Year,
    I.Gender,
    I.AgeRange
ORDER BY
    Year,
    I.Gender,
    I.AgeRange
```

Looking at this query, it is easy to appreciate the elegance of SQL, the code is much easier to write and understand in SQL than it is in DAX. However, for this paper, we are not interested in elegance, the focus is about performance and, not amazingly at this point, the query runs in 3 seconds, exactly the same time used by VertiPaq with SUMMARIZE, but slower than the optimized version of DAX using ADDCOLUMNS.

We even tried to run the query scanning the entire table, i.e. by removing the filter on the year, to further observe the behavior of the two engines. The results are in line with what we have seen up to now: Vertipaq makes a full scan and distinct count in 4 seconds, while ColumnStore takes 12 seconds to perform

the same operation. SQL is clearly a loser here, it seems that most of its power comes from filtering rather than from calculations.

That said… I would like you to stop for one second and repeat with me: “we have been able to compute the distinct count of a slowly changing attribute of a dimension using a 4 billion rows fact table in 4 seconds”. To me, it still seems incredible to be able to achieve this level of performance.

Leaf Level Calculations

Another scenario where the xVelocity engine shines is with leaf level calculations. Computing the weighted average of a column in xVelocity is very fast and, again, in Multidimensional it was a real pain.

In order to perform the test on leaf level calculation we have used a query that computes the weighted average of the age of individuals.

The DAX query is very similar to the previous one, the only difference being the calculated column computed by ADDCOLUMNS. We already know that using SUMMARIZE would only lead to worse performance, so we do not spend time testing it.

```
EVALUATE
    CALCULATETABLE (
        ADDCOLUMNS (
            CROSSJOIN (
                VALUES ('Date'[Year]),
                VALUES (Individuals[Gender]),
                VALUES (Individuals[AgeRange])
            ),
            "Weighted Age",
                CALCULATE (
                    SUMX (Audience, Audience[Weight] * Audience[Age])
                  / SUM (Audience[Weight])
                )
        ),
        'Date'[Year] = 2010
    )
ORDER BY
    'Date'[Year],
    Individuals[Gender],
    Individuals[AgeRange]
```

Running for data of a single year, this query executes in 4 seconds. The code is optimized to take full advantage of the VertiPaq engine, pushing the calculation at the storage engine level. On the full table (i.e. by removing the filter on the year), the query runs in 10 seconds.

- Not all the queries using leaf level calculations can be pushed down to VertiPaq in this way but it is worth to note that, if this can happen, the price to pay in order to perform leaf-level calculations is really negligible. Whenever you prepare a data model on Tabular, you need to think at what to store in the tables in order to take full advantage of the VertiPaq engine.

It is worth pointing out that the result, being a division between two integers, is a floating point value. In DAX you do not need to worry about this, all necessary type conversions happen when needed in a flawless way.

The SQL code is very simple too:

```
SELECT
    D.Year,
    I.Gender,
    I.AgeRange,
    WeightedAge = SUM (CAST (A.Age * A.Weight AS BIGINT))
                / SUM (CAST (A.Weight AS BIGINT))
FROM
    Audience A
    INNER JOIN Date D
        ON A.ID_Date = D.ID_Date
    INNER JOIN dbo.Individuals I
        ON A.ID_Individual = I.ID_Individual
    WHERE D.Year = 2011
GROUP BY
    Year,
    I.Gender,
    I.AgeRange
ORDER BY
    Year,
    I.Gender,
    I.AgeRange
```

Running the query as you see above, it executes in 3 seconds for a single year and in 17 seconds on the full data set. It seems that, when it comes to leaf level calculation, ColumnStore is not as optimized as VertiPaq. However, by reducing the data set to a single year the two engines produces similar response time.

Moreover, in SQL we have to convert types to BIGINT in order to let the query run, because the SUM of all values does not fit into a single integer value. Thus, the result is an integer value, which is not what we want. In order to get the correct result, we needed to change the query by modifying the WeightedAge calculation as

```
WeightedAge = CAST (SUM (CAST (A.Age * A.Weight AS BIGINT)) AS FLOAT)
            / SUM (CAST (A.Weight AS BIGINT))
```

We convert the SUM result to a FLOAT. If we were to convert the Age or the Weight to FLOAT, then all computations would be performed in floating point math, leading to poor performances. Simplicity, in this case, is a point for VertiPaq, which handles this scenario by hiding the complexity of data conversion to the programmer.

Digressions about ColumnStore Performance

Now that you have read some interesting information, it is time to take a break and further elaborate on the introduction. The original query I was working on in Amsterdam was performing the work of the two previous queries, i.e. leaf level calculations and distinct count. As I mentioned at the beginning, I was observing very poor performance for ColumnStore, whereas the same query was running at an amazing speed on VertiPaq.

This is a slightly simplified version of the query I was working on and that lead me to meet Hugo:

```
SELECT
    D.Year,
    I.Gender,
    I.AgeRange,
    NumOfIndividuals = COUNT(DISTINCT (I.COD_Individual)),
    WeightedAge = CAST (SUM (CAST (A.Age * A.Weight AS BIGINT)) AS FLOAT)
                / SUM (CAST (A.Weight AS BIGINT))
FROM
    Audience A
    INNER JOIN Date D
        ON A.ID_Date = D.ID_Date
    INNER JOIN dbo.Individuals I
        ON A.ID_Individual = I.ID_Individual
    WHERE D.Year = 2011
GROUP BY
    Year,
    I.Gender,
    I.AgeRange
ORDER BY
    Year,
    I.Gender,
    I.AgeRange
```

You can see that this query simply computes both the weighted average and the distinct count in the same query. This query executed in 3 minutes and 42 seconds, using all of the cores at 100%, and this query leads me to believe that ColumnStore was a poor performer. At that time, I did not try to simplify the query by removing one of the two calculated columns. I had no idea of the internals of ColumnStore and simply thought that optimization was a task for the optimizer, not for the programmer. Now that you have seen that both queries run very fast (3 seconds each), it is really unbelievable to think that the more complex one takes more than 3 and half a minute to run.

In reality, the problem is mixing distinct counts and other measures in the same query. If you express the same query as the JOIN of two sub-queries, one computing the distinct count and one computing the weighted age, joining the results together after the end, then the same result is obtained in 6 seconds. The fact table is scanned twice, but the result is much better.

Here you can find the resulting query. Thanks to CTE syntax, it is not hard to read and it runs at an amazing speed.

```
WITH    SubQuery1
          AS (SELECT
                D.Year,
                I.Gender,
                I.AgeRange,
                NumOfRows = COUNT(*),
                WeightedAge = SUM(CAST (Weight AS BIGINT) * CAST (Age AS INT))
                            / SUM(CAST (Weight AS BIGINT))
              FROM
                Audience A
                INNER JOIN Date D
                    ON A.ID_Date = D.ID_Date
                INNER JOIN dbo.Individuals I
                    ON A.ID_Individual = I.ID_Individual
              WHERE
                D.Year = 2011
              GROUP BY
                Year,
                I.Gender,
                I.AgeRange
             ),
        SubQuery2
          AS (SELECT
                D.Year,
                I.Gender,
                I.AgeRange,
                NumOfIndividuals = COUNT(DISTINCT (I.COD_Individual))
              FROM
                Audience A
                INNER JOIN Date D
                    ON A.ID_Date = D.ID_Date
                INNER JOIN dbo.Individuals I
                    ON A.ID_Individual = I.ID_Individual
              WHERE
                D.Year = 2011
              GROUP BY
                Year,
                I.Gender,
                I.AgeRange
             )
    SELECT
        S1.Year,
        S1.Gender,
        S1.AgeRange,
        NumOfRows,
        WeightedAge,
        NumOfIndividuals
    FROM
        SubQuery1 S1
        INNER JOIN SubQuery2 S2
            ON S1.AgeRange = S2.AgeRange AND S1.Gender = S2.Gender AND S1.Year = S2.Year
```

The lesson learned here is simple: you cannot know everything and, before going to any kind of conclusion on a topic you do not master, it is always better to ask to somebody who knows the topic better than you do.

Anyway, the scenario is probably going to be improved in further releases of SQL Server, since these kinds of patterns can be easily recognized and optimized by the SQL engine.

Many-to-many Relationships

Sometimes I think that no SQLBI whitepapers can be published without speaking about many-to-many relationships. Of course, this is not an exception. Star joins are good but, in our data model, the many-to-many part is a very important one. For this reason it is important to measure performances of the engines in the many-to-many scenario.

We need to modify the original query using the many-to-many pattern.

```
EVALUATE
    CALCULATETABLE (
          CALCULATETABLE (
             ADDCOLUMNS (
                 CROSSJOIN (
                     VALUES ('Date'[Year]),
                     VALUES (Individuals[Gender]),
                     VALUES (Individuals[AgeRange])
                 ),
                 "Num Of Individuals",
                     CALCULATE (
                         COUNTROWS (SUMMARIZE (Audience, Individuals[COD_Individual]))
                     )
             ),
             BridgeIndividualsTargets
          ),
          Targets[Target] = "Male",
         'Date'[Year] = 2010
    )
```

In this case, we have used the simplest way to express the many-to-many pattern, using the bridge table as one of the parameters of CALCULATE in order to force the filter application (for further information you can see http://gbrueckl.wordpress.com/2012/05/08/resolving-many-to-many-relationships-leveraging-dax-cross-table-filtering/).

We tested various scenarios of filtering here:

- Full table (i.e. no filters): 3.5 seconds
- One year: 2 seconds
- One month: 1 second

The performance is outstanding. The many-to-many pattern is used in a very efficient way. It might be worth remembering again that many-to-many performance can be the main reason to build a prototype in Tabular, because it runs incredibly faster than in Multidimensional. But what about ColumnStore? In order to test it, we have written a query that resembles the same filtering mechanism of VertiPaq:

```
SELECT
    D.Year,
    I.Gender,
    I.AgeRange,
    NumOfIndividuals = COUNT(DISTINCT (I.COD_Individual))
FROM
    Audience A
    INNER JOIN Date D
        ON A.ID_Date = D.ID_Date
    INNER JOIN dbo.Individuals I
        ON A.ID_Individual = I.ID_Individual
    INNER JOIN (SELECT
                    ID_Individual
                FROM
                    dbo.BridgeIndividualsTargets B
                    INNER JOIN dbo.Targets T
                        ON B.ID_Target = T.ID_Target
                WHERE
                    T.Target = 'MASCHI'
                GROUP BY
                    ID_Individual
               ) B
        ON A.ID_Individual = B.ID_Individual
WHERE
    D.Year = 2010
GROUP BY
    Year,
    I.Gender,
    I.AgeRange
```

We measured performance using different filters:

- Full table (i.e. no filters): 10 seconds
- One year: 3 seconds
- One month: 1 second

The behavior of the engine is exceptional also with ColumnStore, even if not so good as VertiPaq. Moreover, when the filter is applied to a single month, ColumnStore is as fast as VertiPaq, confirming the power of the filtering algorithm implemented in ColumnStore.

An interesting note here is that the way we have expressed the query is derived from our knowledge of the DAX language, because we tried to express the same algorithm used in DAX to express the query in SQL. It might be the case that a different formulation of the query will result in a different performance. The code we have used to express this query has been optimized to leverage the ColumnStore index.

Other Considerations

In the preceding sections, we have spoken about query time performance. However, there are other measurements and topics that need to be taken into account when analyzing which technology best fits your needs.

In this section, I will outline some thoughts about different measurements and capabilities that might drive you toward one solution or the other.

Processing Time

When it comes to measuring processing time, ColumnStore is a great winner. A full process of the Analysis Services database takes more or less 6 hours to read, compress and store all 4 billion rows. As you can imagine, dimension processing is negligible, but transferring all the rows from SQL Server to Analysis Services to complete the processing is a time consuming operation.

The complete rebuild of the ColumnStore index, on the other hand, runs in 1 hour, i.e. 6 times faster. It is clear that this is not an important measurement, because in any case we would use partitions to improve processing performance. However, it is important to know that processing a ColumnStore index is much faster than processing an SSAS database.

It is worth to remember that putting a ColumnStore on a table makes it read-only and, if you want to update its content, you will need to use partition techniques.

MetaData Layer

The Tabular project acts as a metadata layer on top of data. Not only does it provide the capability of being queried using DAX and MDX, it also provides metadata to client tools like Excel, so that the data in Analysis Services is consumed directly by end users, without the need to write a custom application that queries the database.

SQL does not provide the same richness of metadata as SSAS. Thus, in the end, it might be the case that ColumnStore indexes are simply not a viable option for many scenarios. But, in the case where you do not need this metadata layer, then you still have the option to choose which technology works best for your specific case.

Moreover, when we speak about metadata layer, we do not only refer to the description of columns, hierarchies, format strings and other cosmetic information. Using SSAS you have the ability to define measures and calculated columns inside the data model, making it much easier for users to use the data in your database.

Memory Usage

In terms of memory usage, SQL is by far the best option. Vertipaq needs to hold the entire data model in memory, whereas SQL Server can dynamically load data in memory to answer queries and unload unused information. Thus, if your database does not fit in memory, then SQL is your only option. And, as we have seen, it is not a bad one. Yes, you are missing metadata and many good features of VertiPaq, but by

carefully designing the data model and optimizing the queries, you will be able to obtain good performances on your existing server.

DirectQuery over ColumnStore

At this point of the whitepaper, I guess some of you are wondering whether a viable solution can be that of using a Tabular data model built using the DirectQuery technology. This would create a metadata layer on top of an existing SQL database, where we can create ColumnStore indexes in order to improve performance.

We do not expect great performance from such a solution. The problem is that the DAX queries are translated by DirectQuery in complex SQL ones and they are not aware of the limitations that currently exist in ColumnStore. Thus, most of the DAX queries you execute in DirectQuery cannot leverage the full power of the ColumnStore technology, resulting in poor performances.

In order to speed up DirectQuery, ColumnStore seems not a good option. Clearly, your mileage may vary, it depends from a huge number of factors. From the tests we carried on, it seems that SQL queries generated by DirectQuery are not well optimized by the SQL query engine. Of course, this might change in future releases and/or service packs.

Cache Usage

Usage of cache is another great point for VertiPaq, which caches the results of Storage Engine queries in a very efficient way. The response time is immediate if the same request to the Storage Engine has been already executed in the past (and it is still in cache, of course).

Most of the queries we have used in this whitepaper make extensive usage of the Storage Engine and very few operations use the Formula Engine. In this scenario, the cache makes a big difference when the system is in production.

SQL Server, on the other hand, does not cache results (which is obvious, because SQL Server is not intended to be an analytical system, it has a wider range of applications). Each time you run a query, you will need to pay the same query time to obtain a result. Possible improvements might be observed just because the first query gets data from disk, but once ColumnStore data is loaded in memory, there are no differences in following executions.

The Language

I am a big fan of DAX. It is simple, elegant, powerful and it has a great potential. That said, I think that in terms of expressivity, elegance and power, SQL is a clear winner. Being on the market for so many years, SQL is a very mature language and there are on the market much more skilled SQL programmers than DAX ones.

Moreover, by leveraging SQL as the language to query your data warehouse, we would be able to use stored procedure, complex joins, functions with parameters, views and, in general, all the features already present in the SQL Server engine. As of now, not all the operators in SQL are capable of running in batch mode and exploiting the power of the xVelocity engine, but it is predictable that, in the future, the situation will only be improved.

Conclusions

Is it ColumnStore better than VertiPaq, or is it the case that VertiPaq is the winner? The answer, as you might imagine, is always the same: it depends.

Let us put down a list of the features tested and mark the winner.

	Vertipaq	Tie	ColumnStore
Simple Aggregation	✓		
Aggregation in same table	✓		
Filtering same table		✓	
Star Join		✓	
Distinct Count	✓		
Leaf Level Calculations	✓		
Many-to-many Relationships	✓		
Processing Time			✓
MetaData Layer	✓		
Memory Usage			✓
Cache Usage	✓		
Language Expressivity			✓

If we were forced to draw a single conclusion, we would say that this is very near to a tie, with VertiPaq with a slight advantage. Yes, VertiPaq is faster on some queries, especially when a full scan of the table is required. The reasons is probably due to the fact that, even if data is already in memory due to the cache, SQL needs some work in order to discover it and cannot leverage the fact that no I/O will ever be needed. VertiPaq is even faster when it comes to do heavy calculations like distinct counts, again confirming that the optimizations in the calculation engine have a good effect on performances.

On the other hand, when data is filtered (which happens most of the time) the SQL algorithm shines and reaches the same speed of VertiPaq. Moreover, using SQL you are not limited to the available memory: data is loaded when needed and discarded when not needed anymore. This difference alone can be the driver that makes you select SQL against VertiPaq. We have to remember that, in case data needs to be loaded from disk, all the timings will be very different.

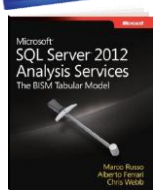
Finally, also consider that SSAS integrates a rich metadata layer and the capability to use DAX and MDX to query the database, along with calculated columns, measures and a very good client-tool interface. Thus, depending on your specific scenario, you now have arguments both in favor and in contrast with each technology.

LINKS

- http://www.sqlbi.com/articles/vertipaq-vs-columnstore/ : the project home page for this paper and related resources
- http://www.sqlbi.com : community dedicated to Business Intelligence with SQL Server
- http://sqlblog.com/blogs/alberto_ferrari : blog of Alberto Ferrari (author)
- http://sqlblog.com/blogs/marco_russo : blog of Marco Russo
- http://www.sqlbi.com/articles/many2many/ : the Many-to-Many Revolution 2.0 Paper
- http://www.sqlbi.com/articles/sqlbi-methodology/ : the SQLBI Methodology paper for best practices in BI solution design

- http://www.amazon.com/gp/product/1847197221/?tag=se04-20 : the book "Expert Cube Development with Microsoft SQL Server 2008 Analysis Services" written by Marco Russo, Alberto Ferrari and Chris Webb

- http://www.amazon.com/dp/0735640580/?tag=se04-20 : the book "PowerPivot for Excel 2010: Give your data meaning" written by Marco Russo and Alberto Ferrari.



- http://www.amazon.com/dp/0735658188/?tag=se04-20 : the book "Microsoft SQL Server 2012 Analysis Services: The BISM Tabular Model" written by Marco Russo, Alberto Ferrari and Chris Webb.
- http://www.powerpivotworkshop.com : Two-day workshop on PowerPivot and the DAX language, the perfect place where to touch the Karma of DAX with the author of this paper.
- http://www.ssasworkshop.com : Two-day workshop on SQL Server Analysis Services BISM Tabular, the perfect introduction to the new world of Tabular.