Data Cube, MDX query, and Dashboard

Simone Di Luna 544322

1 Data Cube

This section explains the process followed to build the data cube with Visual Studio 2019.

After specifying the server and database name in the project properties, the Analysis Services database `simonediluna_SSAS_DB` was designed by following the steps below in chronological order:

1. Create the *Data Source* through the *Data Source Wizard* and define the connection string used to connect to the relational database `simonediluna_DB` in the *lds* server.
2. Create a view on all the user-created tables in the `simonediluna_DB` data warehouse via the *Data Source View Wizard*.
3. Create the cube dimensions by means of the *Dimension Wizard*.
4. Create the cube and define its measures through the *Cube Wizard*.
5. Build and deploy the cube.

Since explaining in writing and in detail all the steps and settings used would take too much time and space, we will focus here only on the creation of the *Time dimension*, while briefly describing the hierarchies created for the other dimensions. Eventually, we will also cover measures and measure groups.

Time dimension. It is based solely on the `time_by_day` table of the data warehouse. In particular, all the columns of the `time_by_day` table were used at least once to derive the dimensional attributes. In addition, we added two more attributes obtained by creating two calculated columns:

1. The named calculation `day_of_week` associates the name of the days of the week from the `day_name` column with the corresponding ordinal numbers ranging from 1 to 7. This was done to sort the members of the `Day Name` attribute in the correct order. Specifically, we defined a one-to-one relationship between `Day Name` and `Day Of Week` and changed their properties in the following way:
    - To speed up processing time, we disabled `Day Of Week`, which was created only to order the `Day Name` members. Specifically, we set `AttributeHierarchyEnabled = False`, `AttributeHierarchyOptimizedState = NotOptimized`, `AttributeHierarchyOrdered = False`.
    - In the `Day Name` properties, we set `OrderBy = AttributeKey` and `OrderByAttribute = Day Of Week`.

2. The named calculation `moth_name` associates the ordinal number of each month —taken from the `moth_of_year` column— with the corresponding name of the months.

In addition to these attribute hierarchies, we also created a user-defined balanced hierarchy, called *Calendar*, consisting of four levels. At the root, we have the source attribute `Year Of Calendar` (renamed as `Year` in the hierarchy), which is bound to the `the_year` column of the `time_by_day` table thanks to the *KeyColumns* property.

Immediately below, there is the `Quarter` level, where the source attribute is the `Quarter Of Year`. This attribute is bound through the KeyColumns property to two columns of the source table, namely `the_year` and `the_quarter`. The benefit is that we can now define a functional dependency between `Quarter Of Year` and `Year Of Calendar` in the *Attribute Relationships* tab, thus improving performance. Although each member has two keys, the names are assigned by `the_quarter` column thanks to the *NameColumn* property.

`Month` is the third level of the hierarchy. Its source attribute is `Month Of Year`. We bound this attribute to two columns by defining a composite key column consisting of `the_year` and `month_of_year`. The result is that now the `Month Of Year` attribute is able to uniquely determine `Quarter Of Year` as illustrated by the specific attribute relationship. Furthermore, we also bound the attribute `Month Of Year` to the calculated column `month_of_year` via the *NameColumn* property thus showing the user the actual name of the months instead of their ordinal numbers.

Finally, at the bottom level of the hierarchy, there is the `Date` attribute, bound to the `time_id` column. Since it is the primary key of the source table and since all attributes come from the same table, `Date` determines all other attributes.

Visual Studio warns against having attributes acting simultaneously as flat hierarchies and user-defined hierarchies. The documentation on Microsoft's official Web site also suggests hiding the attribute hierarchies in this scenario. Therefore, for all the attributes in the *Calendar* hierarchy, we set the property `AttributeHierarchyVisible = False`. However, to also have the ability to navigate these attributes outside the user-defined hierarchy, we created new attributes from the same columns of the source table. Specifically, we defined the attributes, `Year`, `Quarter`, `Month`, and `Month Of Year 1`, linked to `the_year`, `the_quarter`, `month_of_year`, and `month_of_year` columns, respectively. The `Month Of Year 1` attribute was created only to sort the member names of the `Month` attribute, so it was disabled with the same procedure used for `Day Of Week`.

It is worth noting that all the attribute relationships we have defined are of *rigid* type. This is because the relationships between attributes in the time dimension cannot change over time. Rigid relationships, unlike *flexible* ones, bring benefits in terms of processing time.

Geography dimension. It is based on the homonymous table of the data warehouse. We created a three-level hierarchy named `GeoHierarchy` which, from top to bottom, is composed of the following attributes: `Continent In Hierarchy` (renamed `Continent`), bound to the `continent` column; `Country In Hierarchy` (renamed `Country`), bound to the `country` column; `Region In Hierarchy` (renamed `Region`), bound to the `region` column.

Each member of a level uniquely determines the member in the upper level. Namely, we defined one rigid attribute relationship between `Region In Hierarchy` and `Country In Hierarchy` and another between `Country In Hierarchy` and `Continent In Hierarchy`.

As with the Time dimension, for all the three attributes used in the `GeoHierarchy`, we set the property `AttributeHierarchyVisible = False`.

Nevertheless, to allow users to use those attributes on their own, we recreated them from other instances of the same columns. These attributes appear to the user as flat hierarchies named, respectively, `Continent` `Country`, and `Region`.

The `Geo Id` attribute is bound to the primary key column `geo_id`. Since it is useless for cube analysis, to improve processing time we modified the following properties: `AttributeHierarchyOptimizedState = NotOptimized`, `AttributeHierarchyOrdered = False`, `AttributeHierarchyVisible = False`. However, we left the attribute enabled to ensure the data consistency check (indeed, it determines all the other attributes).

CPU dimension. It is based on the `cpu_product` table in the data warehouse. For this dimension, we have created two user-defined hierarchies, `BrandSeriesCpu` and `BrandSocketCpu`, to allow navigation from the CPU brand to the corresponding CPU names, either by series or socket.

The `BrandSeriesCpu` hierarchy consists of three levels. At the root is the `Brand Of Series` attribute (renamed `Brand`), bound to the `brand` column. Then there is the `Series In Hierarchy` attribute (renamed `Series`), bound to the `series` column. The lower level is populated by members of the `CPU Name In Hierarchies` attribute, bound to the `cpu_id` column and renamed in the hierarchy simply as `CPU Name`.

The `BrandSocketCpu` hierarchy is very similar to `BrandSeriesCpu` but with two differences:

1. At the root, there is the `Brand Of Socket` attribute, which is another instance of the `brand` column. This was necessary to define the two attribute relationships socket $\rightarrow$ brand and series $\rightarrow$ brand because Visual Studio complains when an attribute is on the right-hand side of two separate attribute relationships.

2. At the second level, we find the `Series In Hierarchy` attribute, bound to the `series column`.

All attributes of the user-defined hierarchies have been hidden by setting `AttributeHierarchyVisible = False`. As for the other dimensions, we have created other attributes from the same columns, so we can also use the flat hierarchies of these attributes. Finally, we also replicated the functional dependencies already designed within the user-defined hierarchies with the new attributes.

Vendor dimension. Based on the homonymous table, this dimension has one only attribute, `Vendor Name`, which is bound to the `vendor_id` column via the *KeyColumns* property, but whose members are named from the `vendor_name` column thanks to the *NameColumn* property.

Measures. Using the *Cube Wizard*, Visual Studio automatically selected the measures from the `Cpu Sales Fact` table and defined the homonymous group of measures. In addition to the existing measures, the wizard also created the `Cpu Sales Fact Count` measure with the following properties: `AggregateFunction = Count` and `FormatString = Standard`.

Regarding the measures `Sales Usd`, `Sales Currency`, and `Cost`, we made sure that `AggregateFunction = Sum` and that `FormatString = Currency`.

We also created another measure group based on the `vendor` table containing the `NVendors` custom measure, that can be used for counting the distinct vendor in a set. Its properties are `AggregateFunction = DistinctCount` and `FormatString = Standard`.

In addition, we also created three calculated measures within the `Cpu Sales Fact` group:

- The `Gross Profit`, defined as `Sales Usd`$-$`Cost`, having `FormatString = Currency`.
- The `Margin`, defined as $\frac{\texttt{Gross Profit}}{\texttt{Sales Usd}}$, with `FormatString = Percent`.
- The `Markup`, defined as $\frac{\texttt{Gross Profit}}{\texttt{Cost}}$, with `FormatString = Percent`.

For what concern the storage mode, we kept the default option unchanged, i.e., MOLAP without proactive caching. This means that no agent is listening for changes in the data warehouse: if a change occurs, all queries on the SSAS database continue to run through the old MOLAP cube, thus providing out-of-date information unless the cube is manually reprocessed.

2 MDX query

The MDX query proposed can be solved in at least three ways (see listing 2), each of which differs in how the previous year's profit is calculated, namely:

1. By means of the `PREVMEMBER` dot function.
2. Via the `CURRENTMEMBER` dot function.
3. Through the `PARALLELPERIOD` function.

During the design phase of the Data Cube in the previous task, we additionally included the `Gross Profit` measure, however, to better embrace the rationale of the question, we pretended that it was not available. Specifically, since this measure is required in all three methods, we created the `Profit` calculated member with a *session-scoped* context (members created via the `WITH` statement have a query scope).

It is worth noting that in all three reports, when, for a specific year, the previous year's profit is missing, the report shows the value `inf`. This is how the engine treats division by 0. We could have treated these cases with the MDX function `IIF` (as shown in listing 1), but there are no meaningful values to use to replace the `inf` value (indeed, both 100% and 0% are not semantically correct. `inf` seems more informative hence we preserved it).

```
WITH MEMBER [% Profit Variation] AS
IIF(
[Time].[Year].PREVMEMBER,
([Profit] - ([Time].[Year].PREVMEMBER, [Profit]))
/ ([Time].[Year].PREVMEMBER, [Profit]),
1
),
FORMAT_STRING = 'Percent'
SELECT
[% Profit Variation] ON 0,
([CPU].[Brand].[Brand], [Time].[Year].[Year]) ON 1
FROM [CPU Sales];
```

Listing 1. Replace the `inf` value by 100%.

```
/* Global calculated member */
CREATE MEMBER [CPU Sales].[Measures].[Profit] AS
   [Measures].[Sales Usd] - [Measures].[Cost];

/* Method 1: PREVMEMBER */
WITH MEMBER [% Profit Variation] AS
   ([Profit] - ([Time].[Year].PREVMEMBER, [Profit]))
      / ([Time].[Year].PREVMEMBER, [Profit]),
   FORMAT_STRING = 'Percent'
SELECT
   [% Profit Variation] ON 0,
   ([CPU].[Brand].[Brand], [Time].[Year].[Year]) ON 1
FROM [CPU Sales];

/* Method 2: LAG */
WITH MEMBER [% Profit Variation] AS
   ([Profit] - ([Time].[Year].CURRENTMEMBER.LAG(1), [Profit]))
      / ([Time].[Year].CURRENTMEMBER.LAG(1), [Profit]),
   FORMAT_STRING = 'Percent'
SELECT
   [% Profit Variation] ON 0,
   ([CPU].[Brand].[Brand], [Time].[Year].[Year]) ON 1
FROM [CPU Sales];

/* Method 3: PARALLELPERIOD */
WITH
   MEMBER [Profit Prev Year] AS
      (
         PARALLELPERIOD(
            [Time].[Calendar].[Year],
            1,
            [Time].[Calendar].CURRENTMEMBER
         ),
         [Profit]
      )
   MEMBER [% Profit Variation] AS
      ([Profit] - [Profit Prev Year]) / [Profit Prev Year],
      FORMAT_STRING = 'Percent'
SELECT
   [% Profit Variation] ON 0,
   ([CPU].[Brand].[Brand], [Time].[Calendar].[Year]) ON 1
FROM [CPU Sales];

/* Delete global member */
DROP MEMBER [CPU Sales].[Measures].[Profit];
```

Listing 2. MDX queries.

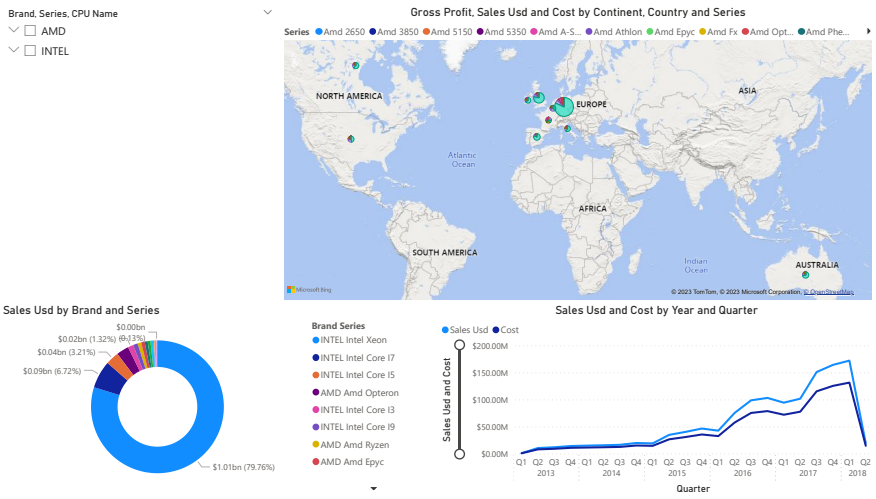

Figure 1. Power BI dashboard.

3 Dashboard

The last assignment of the project asked to reproduce a Power BI dashboard showing:

1. The geographical distribution of sales and costs of the CPU products.
2. How costs and sales change over time at different levels of granularity.

The final dashboard (see 1) is composed of three interactive diagrams and one slicer. Interestingly, by changing the level of granularity either in the slicer or in the other diagrams, the changes affect all the charts simultaneously.

Let's describe briefly the elements in the dashboard:

- The slicer is a filter that allows us to navigate through all the levels of the `BrandSeriesCpu` hierarchy.
- The map shows the geographical distribution of the measures of interest at different level of granularity. Notably, the size of the bubbles represents the information about the gross profit. The gross profit itself is a calculated measure that summarizes both the sales and the costs of the products at the same time. These last two measures have been included in the Tooltips section, so this data is shown in the popup that appears by hovering the mouse over the bubbles. To better fit the assignment requirements, we entered the `Series` attribute in the legend field, thereby displaying product information as well. This transformed the bubbles into pie charts.

- The doughnut chart shows information similar to the bubbles in the map chart but with a higher level of detail. Indeed, it uses the `BrandSeriesCpu` hierarchy that allows sales to be displayed not only by `Series` but also by CPU brand and name.
- Finally, the line chart shows the change in sales and costs within the time hierarchy. It also features a slicer that allows the scale of the Y-axis to be increased or decreased.