Measures and Measure Groups in Microsoft Analysis Services: Part 1

by Chris Webb | July 2009 | Microsoft

In this two-part article by **Chris Webb**, we will look at measures and measure groups, ways to control how measures aggregate up, and how dimensions can be related to measure groups. In this part, will cover useful properties of measures, along with built-in measure aggregation types and dimension calculations.

**Measures and aggregation**

**Measures** are the numeric values that our users want to aggregate, slice, dice and otherwise analyze, and as a result, it's important to make sure they behave the way we want them to. One of the fundamental reasons for using Analysis Services is that, unlike a relational database it allows us to build into our cube design business rules about measures: how they should be formatted, how they should aggregate up, how they interact with specific dimensions and so on. It's therefore no surprise that we'll spend a lot of our cube development time thinking about measures.

**Useful properties of Measures**

Apart from the *AggregateFunction* property of a measure, which we'll come to next, there are two other important properties we'll want to set on a measure, once we've created it.

**Format string**

The Format String property of a measure specifies how the raw value of the measure gets formatted when it's displayed in query results. Almost all client tools will display the formatted value of a measure, and this allows us to ensure consistent formatting of a measure across all applications that display data from our cube.

*A notable exception is Excel 2003 and earlier versions, which can only display raw measure values and not formatted values. Excel 2007 will display properly formatted measure values in most cases, but not all. For instance, it ignores the fourth section of the Format String which controls formatting for nulls. Reporting Services can display formatted values in reports, but doesn't by default; this blog entry describes how you can make it do so:*[***http://tinyurl.com/gregformatstring***](http://tinyurl.com/gregformatstring)*.*

There are a number of built-in formats that you can choose from, and you can also build your own by using syntax very similar to the one used by **Visual BASIC for Applications** (**VBA**) for number formatting. The Books Online topic *FORMAT\_STRING Contents* gives a complete description of the syntax used.

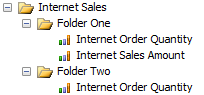
Here are some points to bear in mind when setting the Format String property:

* If you're working with percentage values, using the % symbol will display your values multiplied by one hundred and add a percentage sign to the end. Note that only the display value gets multiplied by hundred—the real value of the measure will not be, so although your user might see a value of 98% the actual value of the cell would be 0.98.
* If you have a measure that returns null values in some circumstances and you want your users to see something other than null, don't try to use a MDX calculation to replace the nulls—this will cause severe query performance problems. You can use the fourth section of the Format String property to do this instead—for example, the following:  
  *#,#.00;#,#.00;0;NA*   
  will display the string **NA** for null values, while keeping the actual cell value as null without affecting performance.
* Be careful while using the **Currency** built-in format: it will format values with the currency symbol for the locale specified in the *Language* property of the cube. This combination of the **Currency** format and the*Language* property is frequently recommended for formatting measures that contain monetary values, but setting this property will also affect the way number formats are displayed: for example, in the UK and the USA, the comma is used as a thousands separator, but in continental Europe it is used as a decimal separator. As a result, if you wanted to display a currency value to a user in a locale that didn't use that currency, then you could end up with confusing results. The value €100,101 would be interpreted as a value just over one hundred Euros to a user in France, but in the UK, it would be interpreted as a value of just over one hundred thousand Euros. You can use the desired currency symbol in a Format String instead, for example '$#,#.00', but this will not have an effect on the thousands and decimal separators used, which will always correspond to the Language setting. You can find an example of how to change the language property using a scoped assignment in the MDX Script here: <http://tinyurl.com/gregformatstring>.
* Similarly, while Analysis Services 2008 supports the translation of captions and member names for users in different locales, unlike in previous versions, it will not translate the number formats used. As a result, if your cube might be used by users in different locales you need to ensure they understand whichever number format the cube is using.

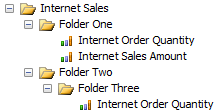
**Display folders**

Many cubes have a lot of measures on them, and as with dimension hierarchies, it's possible to group measures together into folders to make it easier for your users to find the one they want. Most, but not all, client tools support display folders, so it may be worth checking whether the one you intend to use does.

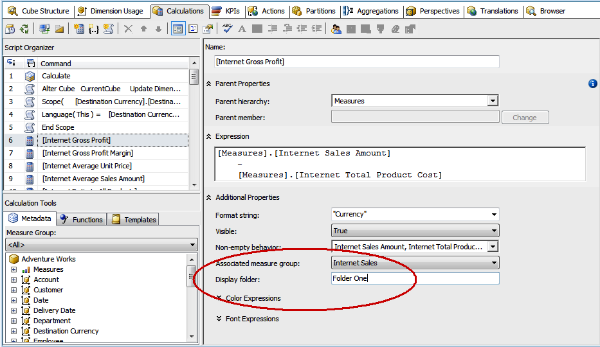
By default each measure group in a cube will have its own folder containing all of the measures on the measure group; these top level measure group folders cannot be removed and it's not possible to make a measure from one measure group appear in a folder under another measure group. By entering a folder name in a measure's *Display Folder* property, you'll make the measure appear in a folder underneath its measure group with that name; if there isn't already a folder with that name, then one will be created, and folder names are case-sensitive. You can make a measure appear under multiple folders by entering a semi-colon delimited list of names as follows: *Folder One*; *Folder Two*.



You can also create a folder hierarchy by entering either a forward-slash / or back-slash delimited list (the documentation contradicts itself on which is meant to be used—most client tools that support display folders support both) of folder names as follows: *Folder One; Folder TwoFolder Three*.



Calculated measures defined in the MDX Script can also be associated with a measure group, through the *Associated\_Measure\_Group* property, and with a display folder through the *Display\_Folder* property. These properties can be set either in code or in Form View in the **Calculations** tab in the Cube Editor:



If you don't associate a calculated measure with a measure group, but do put it in a folder, the folder will appear at the same level as the folders created for each measure group.

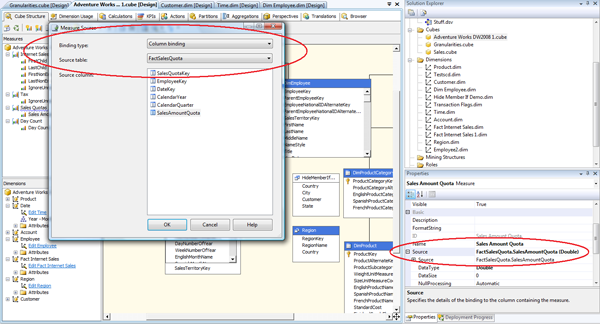
**Built-in measure aggregation types**

The most important property of a measure is *AggregateFunction*; it controls how the measure aggregates up through each hierarchy in the cube. When you run an MDX query, you can think of it as being similar to a SQL *SELECT* statement with a *GROUP BY* clause—but whereas in SQL you have to specify an aggregate function to control how each column's values get aggregated, in MDX you specify this for each measure when the cube is designed.

**Basic aggregation types**

Anyone with a passing knowledge of SQL will understand the four basic aggregation types available when setting the *AggregateFunction* property:

* **Sum** is the commonest aggregation type to use, probably the one you'll use for 90% of all the measures. It means that the values for this measure will be summed up.
* **Count** is another commonly used property value and aggregates either by counting the overall number of rows from the fact table that the measure group is built from (when the **Binding Type** property, found on the Measure Source dialog that appears when you click on the **ellipses** button next to the Source property of a measure, is set to **Row Binding**), or by counting non-null values from a specific measure column (when **Binding Type** property is set to **Column Binding**).



* **Min** and **Max** return the minimum and maximum measure values.

There isn't a built-in Average aggregation type—as we'll soon see, **AverageOfChildren** does not do a simple average—but it's very easy to create a calculated measure that returns an average by dividing a measure with AggregateFunction **Sum** by one with AggregateFunction **Count**, for example:

**CREATE MEMBER CURRENTCUBE.[Measures].[Average Measure Example] AS  
IIF([Measures].[Count Measure]=0, NULL,  
[Measures].[Sum Measure]/[Measures].[Count Measure]);**

**Distinct Count**

The **DistinctCount** aggregation type counts the number of distinct values in a column in your fact table, similar to a *Count(Distinct)* in SQL. It's generally used in scenarios where you're counting some kind of key, for example, finding the number of unique **Customers** who bought a particular product in a given time period. This is, by its very nature, an expensive operation for Analysis Services and queries that use**DistinctCount** measures can perform worse than those which use additive measures. It is possible to get distinct count values using MDX calculations but this almost always performs worse; it is also possible to use many-to-many dimensions to get the same results and this may perform better in some circumstances; see the section on "Distinct Count" in the "Many to Many Revolution" white paper, available at [**http://tinyurl.com/m2mrev**](http://tinyurl.com/m2mrev).

When you create a new distinct count measure, BIDS will create a new measure group to hold it automatically. Each distinct count measure needs to be put into its own measure group for query performance reasons, and although it is possible to override BIDS and create a distinct count measure in an existing measure group with measures that have other aggregation types, we strongly recommend that you do not do this.

**None**

The **None** aggregation type simply means that no aggregation takes place on the measure at all. Although it might seem that a measure with this aggregation type displays no values at all, that's not true: it only contains values at the lowest possible granularity in the cube, at the intersection of the key attributes of all the dimensions. It's very rarely used, and only makes sense for values such as prices that should never be aggregated.

*If you ever find that your cube seems to contain no data even though it has processed successfully, check to see if you have accidentally deleted the Calculate statement from the beginning of your MDX Script. Without this statement, no aggregation will take place within the cube and you'll only see data at the intersection of the leaves of every dimension, as if every measure had AggregateFunction****None****.*

**Semi-additive aggregation types**

The semi-additive aggregation types are:

* **AverageOfChildren**
* **FirstChild**
* **LastChild**
* **FirstNonEmpty**
* **LastNonEmpty**

They behave the same as measures with aggregation type **Sum** on all dimensions except Time dimensions. In order to get Analysis Services to recognize a **Time** dimension, you'll need to have set the dimension's *Type* property to Time in the Dimension Editor.

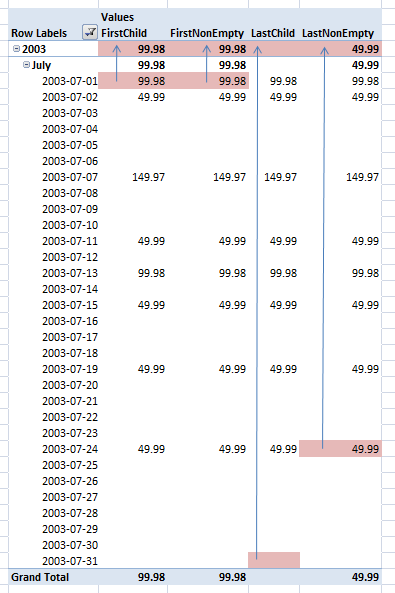
*Sometimes you'll have multiple, role-playing Time dimensions in a cube, and if you have semi-additive measures, they'll be semi-additive for just one of these Time dimensions. In this situation, Analysis Services 2008 RTM uses the first Time dimension in the cube that has a relationship with the measure group containing the semi-additive measure. You can control the order of dimensions in the cube by dragging and dropping them in the Dimensions pane in the bottom left-hand corner of the Cube Structure tab of the Cube Editor; the following blog entry describes how to do this in more detail:****[http://tinyurl.com/gregsemiadd](http://tinyurl.com/gregsemiadd" \t "_blank)****. However, this behavior has changed between versions in the past and may change again in the future.*

Semi-additive aggregation is extremely useful when you have a fact table that contains snapshot data. For example, if you had a fact table containing information on the number of items in stock in a warehouse, then it would never make sense to aggregate these measures over time: if you had ten widgets in stock on January 1, eleven in stock on January 2, eight on January 3 and so on, the value you would want to display for the whole of January would never be the sum of the number of items in stock on each day in January. The value you do display depends on your organization's business rules.

Let's take a look at what each of the semi-additive measure values actually do:

* **AverageOfChildren** displays the average of all the values at the lowest level of granularity on the Time dimension. So, for example, if **Date** was the lowest level of granularity, when looking at a **Year** value, then Analysis Services would display the average value for all days in the year.
* **FirstChild** displays the value of the first time period at the lowest level of granularity, for example, the first day of the year.
* **LastChild** displays the value of the last time period at the lowest level of granularity, for example, the last day of the year.
* **FirstNonEmpty** displays the value of the first time period at the lowest level of granularity that is not empty, for example the first day of the year that has a value.
* **LastNonEmpty** displays the value of the last time period at the lowest level of granularity that is not empty, for example the last day of the year that has a value. This is the most commonly used semi-additive type; a good example of its use would be where the measure group contains data about stock levels in a warehouse, so when you aggregated along the Time dimension what you'd want to see is the amount of stock you had at the end of the current time period.

The following screenshot of an Excel pivot table illustrates how each of these semi-additive aggregation types works:



Note that the semi-additive measures only have an effect above the lowest level of granularity on a Time dimension. For dates like July 17th in the screenshot above, where there is no data for the **Sum** measure, the **LastNonEmpty** measure still returns null and not the value of the last non-empty date.

[**Expert Cube Development with Microsoft SQL Server 2008 Analysis Services**](http://www.packtpub.com/expert-cube-development-with-microsoft-sql-server-2008-analysis-services/book/sl/expertcube-abr1/0709?utm_source=sl_expertcube_abr1_0709&utm_medium=content&utm_campaign=sanjiv)

|  |  |
| --- | --- |
| **[Expert Cube Development with Microsoft SQL Server 2008 Analysis Services](http://www.packtpub.com/expert-cube-development-with-microsoft-sql-server-2008-analysis-services/book/sl/expertcube-abr1/0709?utm_source=sl_expertcube_abr1_0709&utm_medium=content&utm_campaign=sanjiv)** | Design and implement fast, scalable and maintainable cubes     * A real-world guide to designing cubes with Analysis Services 2008 * Model dimensions and measure groups in BI Development Studio * Implement security, drill-through, and MDX calculations * Learn how to deploy, monitor, and performance-tune your cube * Filled with best practices and useful hints and tips |

The semi-additive measure aggregation types (and **None**) are only available in Enterprise Edition; it's possible to recreate the same functionality using MDX but query performance will not be quite as good. Here's an example of how you could overwrite the value of a measure with the *AggregateFunction* **Sum** by using an MDX Script assignment to make it behave in the same way as a **LastNonEmpty** measure:

**SCOPE([Measures].[Sales Amount]);  
THIS = TAIL(  
NONEMPTY(  
{EXISTING [Date].[Date].[Date].MEMBERS}   
\* [Measures].[Sales Amount])  
,1).ITEM(0);  
END SCOPE;**

We may also find that the performance of **LastNonEmpty** is still not good enough, especially on very large cubes. It is not as fast as **LastChild**, so one trick we could try is to calculate the last non empty value for a measure in a new column in your fact table of your ETL. We can then create a new measure from this column, use the **LastChild** measure as its aggregation type, then set its *Visible* property to false and use an MDX Script something like the following to display the new measure's value above the leaf level for the original measure:

**SCOPE([Measures].[Sales Amount], [Date].[Date].[All]);  
THIS = [Measures].[LastChild];  
END SCOPE;**

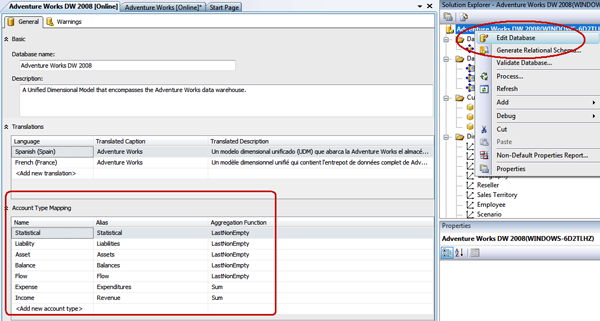
This assignment forces each time period to show the value of the last date within that time period. Typically, though, at the tail of your Date dimension you will have dates that have not occurred yet and which cannot contain any data—for example, if today's date is April 16th 2009, you might only have data in the cube up to April 15th, but the Date dimension will probably contain all of the dates up to December 31st 2009. For these future dates you'll either have to ensure your ETL populates the value of the**LastChild** measure up until the end of the current year or use another MDX Script assignment that is similar to the one described at the end of this blog entry: [**http://tinyurl.com/gregsemiadd**](http://tinyurl.com/gregsemiadd).

**By Account**

A **Chart of Accounts** dimension is a feature common to many cubes containing financial data; in many ways it has a lot in common with the measures dimension, because each member on a Chart of Accounts dimension represents a different type of value such as profit and loss or balance sheet values for example. A Chart of Accounts dimension is often implemented with a parent/child hierarchy, and there are often complex business rules that govern how the members on the hierarchy should aggregate up, if they even do at all. Analysis Services provides several features to help you build and manage such a dimension, although it's widely accepted that building financial applications in Analysis Services is not as easy as it should be, or indeed as easy as it is in other OLAP servers.

The **ByAccount** aggregation type (available in Enterprise Edition only) is the first of these features we'll discuss, and it allows us to define different semi-additive behavior for a single measure for different members on the main hierarchy of a Chart of Accounts dimension. The steps to get it working are as follows:

* Create your Chart of Accounts dimension.
* Create another attribute that will be used to flag which members on the main hierarchy will use which semi-additive aggregation type. This should have one member for each type of attribute, where each member represents a different form of semi-additive behavior. Call it something like 'Account Type'.
* Set the following properties:
  1. On the dimension itself, set the *Type* property to **Accounts**.
  2. On the main hierarchy, set the *Type* property to **Account**.
  3. On the Account Type attribute, set the *Type* property to **Account Type**.
* Define how each account type should aggregate. This can be done by right-clicking on the name of the project in the **Solution Explorer** in BIDS and selecting **Edit Database**. In this screen, in the **Account Type Mapping** table, the Name column contains the name of the built-in or user-defined account types and the **Alias** column contains a comma-delimited list of member names on the Account Type hierarchy that map to each account type, and **Aggregation Function** allows you to select either **Sum** or one of the semi-additive aggregation types for each account type.



Most of this work can also be done by running the **Define Account Intelligence** wizard. To do this, go to the **Dimension** menu and select **Add Business Intelligence** and then select '**Define Account Intelligence**' from the list of available enhancements.

*In our experience, the****ByAccount****aggregation type is very rarely used. This is partly because cube developers do not know about it, and partly because most financial applications need to implement more complex logic for aggregation than the built-in semi-additive aggregation types allow. This means that aggregation logic is usually implemented using MDX Script assignments, which give complete flexibility although at the expense of a lot of complexity and do not perform as well as the built-in aggregation types. Probably, the best compromise is to start off by using****ByAccount****and then for the accounts which need more complex aggregation, configure them to use****Sum****in the****Edit Database****screen and then use custom MDX for them.*

**Dimension calculations**

In addition to the *AggregateFunction* property of a measure, there are many other ways of controlling how measures aggregate up. Apart from writing code in the MDX Script, these methods involve different types of dimension calculations.

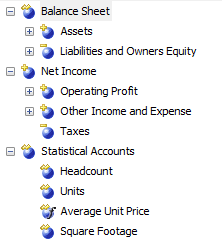
**Unary operators and weights**

In a similar way to how you can control semi-additive aggregation with the **ByAccount** aggregation type, you can control how members on a hierarchy (usually parent/child, but not necessarily) aggregate up to their parents by defining unary operators. To do this, you must create a new column in your dimension table to hold the unary operator values and then set the attribute's *UnaryOperatorColumn* property to point to it.

The unary operators supported are:

* +, which means that the member's value contributes as a positive value to its parent's total
* -, which means that the member's value contributes as a negative value to its parent's total
* \*, which means that the member's value is multiplied with the value of the previous member in the level
* /, which means that the member's value is divided by the value of the previous member in the level
* ~ , which means the member's value is ignored when calculating the value of the parent
* Any numeric value, which works the same as + but where the numeric value is multiplied by the member's value first

Let's take a look at a simple example of this working. Once you've defined some unary operators, you can see which members have which operators in the **Browser** tab of the **Dimension Editor**:

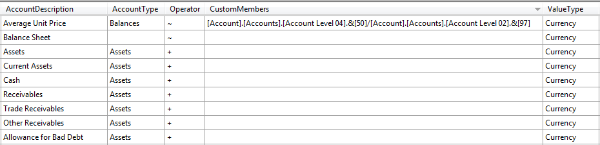


From this, we can see that the **All** member has three children. They are **Balance Sheet**, **Net Income** and**Statistical Accounts**. Since both **Balance Sheet** and **Statistical Accounts** have unary operator ~, **All**member's value is going to be the same as that of **Net Income**. Similarly, we can see that the value of **Net Income** itself will be calculated as follows: + (Operating Profit) + (Other Income and Expense) – (Taxes).

The distributive unary operators, that is + and -, perform much better than calculated members that do the equivalent addition and subtraction in MDX formulas.

**Custom Member Formulas**

Where unary operators do not give you enough control, you can assign an MDX expression to use to calculate the value for each member on an attribute hierarchy using Custom Member Formulas. To do this, we need to create a new column on our dimension table to hold the MDX expressions and then set the *CustomRollupColumn* property of the attribute to this column.



We can also create another column to hold property values that can be applied to each member on the hierarchy too, which then needs to be assigned in the *CustomRollupPropertiesColumn* property. The contents of this column takes the form of a comma-delimited list of property values, just as you would use at the end of a calculated member definition; for example, if a member had the following expression associated with it in the dimension table:

**FORMAT\_STRING='$#,#.00', BACK\_COLOR=RGB(255,0,0)**

then all values for that measure would have the Format String *'$#,#.00'* applied to them, and they would be displayed with a red cell background.

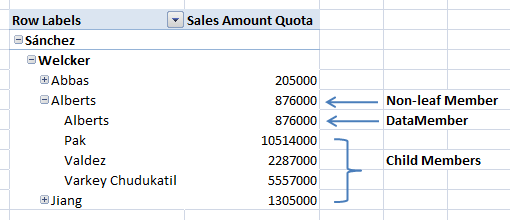
Custom Member Formulas offer something very similar to what you can do with MDX Script assignments, so what are the advantages and disadvantages of using them?

* In terms of query performance, Custom Member Formulas and MDX Script assignments are the same.
* As dimension calculations, Custom Member Formulas differ from MDX Script assignments in one important respect: they use the 'Closest Pass Wins' rule rather than the 'Last Pass Wins' rule when a cell's value can be calculated in two different ways from two different calculations. You don't have the flexibility you have with MDX Script assignments, but you can always be sure that a member's value will be calculated using the formula you specify and not be accidentally overwritten by another assignment.
* Using Custom Member Formulas means the dimension becomes, in some respects, self-documenting. If the formulas are made visible to the end user somehow, perhaps by creating another attribute with *AttributeHierarchyEnabled=False* in the formula column, and the formulas are relatively simple, then users should be able to view and understand them.
* If a dimension with Custom Member Formulas is shared across multiple cubes, then these calculations are automatically applied to all of the cubes. With MDX Script assignments, the code would have to be duplicated across cubes.
* The major drawback to using Custom Member Formulas is that if you need to edit a calculation, you need to edit a value in a relational table. Not only is this inconvenient, it makes debugging calculations much more difficult and it means that your MDX code can be split between two places: the dimension table and the MDX Script.

**Non-aggregatable values**

In some rare cases, you may encounter non-aggregatable measures, that's to say measures whose aggregated values have to be supplied from the data warehouse and cannot be derived from lower granularity values. It's always worth asking why measures are non-aggregatable: in many cases, it's the result of some kind of pre-aggregation or calculation taking place during the ETL phase. While Analysis Services can handle non-aggregatable measures, it's much better at handling additive data, so if you can build your cube from the additive data, we recommend you do so, even if the data volumes end up being much larger.

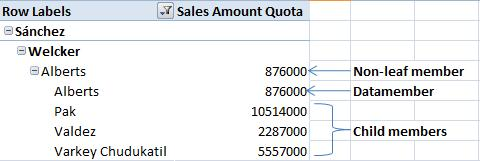
However, if you do have to work with non-aggregatable measures, one way of handling them is by using parent/child hierarchies. Every non-leaf member on a parent/child hierarchy has an extra, system-generated child called a datamember, and you can control the visibility of these datamembers by setting the *MembersWithData* property on the parent/child hierarchy. If your fact table contains values for a non-leaf member on a parent/child hierarchy, then these values will in fact be assigned to the member's datamember; by default, the real non-leaf member's values will then be aggregated from its children and its datamember. This can be seen in the following screenshot from Excel:



In order to display non-aggregatable values, all you need to do is to use an MDX Script assignment to make each non-leaf member on your parent/child hierarchy display just the value of its datamember, for example:

**SCOPE([Measures].[Sales Amount Quota]);  
THIS=[Employee].[Employees].CURRENTMEMBER.DATAMEMBER;  
END SCOPE;**

This code would have the following effect on our Excel example:



As you can see, the value of the non-leaf member **Alberts** is now no longer the sum of its children and its datamember, but that of its datamember alone.

*For more details on working with non-aggregatable values and parent/child hierarchies, see the following paper written by Richard Tkachuk:*[***http://tinyurl.com/nonaggdata***](http://tinyurl.com/nonaggdata)

**Summary**

In this first part, we covered useful properties of measures, basic aggregation types, Custom Member Formulas along with their advantages and disadvantages. In the next part, we will focus on [**measure groups and dimension/measure group relationships**](http://www.packtpub.com/article/measures-and-measure-groups-microsoft-analysis-services-part2).

Measures and Measure Groups in Microsoft Analysis Services: Part 2

by Chris Webb | July 2009 | Content Management

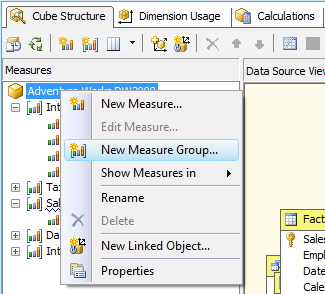
In the previous part of this article by **Chris Webb**, we covered useful [**properties of measures, along with built-in measure aggregation types and dimension calculations**](http://www.packtpub.com/article/measures-and-measure-groups-microsoft-analysis-services-part1?utm_source=sl_expertcube_abr2_0709&utm_medium=content&utm_campaign=sanjiv). In this part, we will focus on measure groups and dimension/measure group relationships.

**Measure groups**

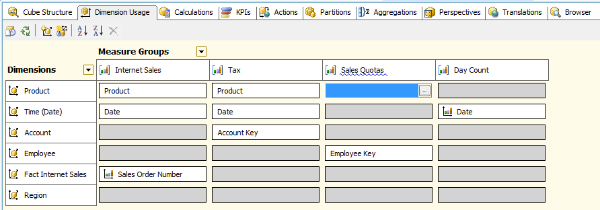
All but the simplest data warehouses will contain multiple fact tables, and Analysis Services allows you to build a single cube on top of multiple fact tables through the creation of multiple measure groups. These measure groups can contain different dimensions and be at different granularities, but so long as you model your cube correctly, your users will be able to use measures from each of these measure groups in their queries easily and without worrying about the underlying complexity.

**Creating multiple measure groups**

To create a new measure group in the **Cube Editor**, go to the **Cube Structure** tab and right-click on the**cube** name in the **Measures** pane and select **'New Measure Group'**. You'll then need to select the fact table to create the measure group from and then the new measure group will be created; any columns that aren't used as foreign key columns in the DSV will automatically be created as measures, and you'll also get an extra measure of aggregation type **Count**. It's a good idea to delete any measures you are not going to use at this stage.



Once you've created a new measure group, BIDS will try to set up relationships between it and any existing dimensions in your cube based on the relationships you've defined in your DSV. Since doing this manually can be time-consuming, this is another great reason for defining relationships in the DSV. You can check the relationships that have been created on the **Dimension Usage** tab of the **Cube Editor**:



In Analysis Services 2005, it was true in some cases that query performance was better on cubes with fewer measure groups, and that breaking a large cube with many measure groups up into many smaller cubes with only one or two measure groups could result in faster queries. This is no longer the case in Analysis Services 2008. Although there are other reasons why you might want to consider creating separate cubes for each measure group, this is still something of a controversial subject amongst Analysis Services developers. The advantages of a single cube approach are:

* All of your data is in one place. If your users need to display measures from multiple measure groups, or you need to create calculations that span measure groups, everything is already in place.
* You only have one cube to manage security and calculations on; with multiple cubes the same security and calculations might have to be duplicated.

The advantages of the multiple cube approach are:

* If you have a complex cube but have to use Standard Edition, you cannot use Perspectives to hide complexity from your users. In this case, creating multiple cubes might be a more user-friendly approach.
* Depending on your requirements, security might be easier to manage with multiple cubes. It's very easy to grant or deny a role access to a cube; it's much harder to use dimension security to control which measures and dimensions in a multi-measure group cube a role can access.
* If you have complex calculations, especially MDX Script assignments, it's too easy to write a calculation that has an effect on part of the cube you didn't want to alter. With multiple cubes, the chances of this happening are reduced.

**Creating measure groups from dimension tables**

Measure groups don't always have to be created from fact tables. In many cases, it can be useful to build measure groups from dimension tables too. One common scenario where you might want to do this is when you want to create a measure that counts the number of days in the currently selected time period, so if you had selected a year on your Time dimension's hierarchy, the measure would show the number of days in the year. You could implement this with a calculated measure in MDX, but it would be hard to write code that worked in all possible circumstances, such as when a user multi-selects time periods. In fact, it's a better idea to create a new measure group from your Time dimension table containing a new measure with *AggregateFunction* **Count**, so you're simply counting the number of days as the number of rows in the dimension table. This measure will perform faster and always return the values you expect. This post on Mosha Pasumansky's blog discusses the problem in more detail:[**http://tinyurl.com/moshadays**](http://tinyurl.com/moshadays)

***MDX formulas vs pre-calculating values*** *If you can somehow model a calculation into the structure of your cube, or perform it in your ETL, you should do so in preference to doing it in MDX only so long as you do not compromise the functionality of your cube. A pure MDX approach will be the most flexible and maintainable since it only involves writing code, and if calculation logic needs to change, then you just need to redeploy your updated MDX Script; doing calculations upstream in the ETL can be much more time-consuming to implement and if you decide to change your calculation logic, then it could involve reloading one or more tables. However, an MDX calculation, even one that is properly tuned, will of course never perform as well as a pre-calculated value or a regular measure. The day count measure, discussed in the previous paragraph, is a perfect example of where a cube-modeling approach trumps MDX. If your aim was to create a measure that showed average daily sales, though, it would make no sense to try to pre-calculate all possible values since that would be far too time-consuming and would result in a non-aggregatable measure. The best solution here would be a hybrid: create real measures for sales and day count, and then create an MDX calculated measure that divided the former by the latter. However, it's always necessary to consider the type of calculation, the volume of data involved and the chances of the calculation algorithm changing in the future before you can make an informed decision on which approach to take.*

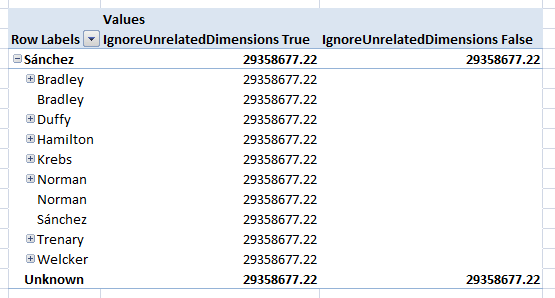
**Handling different dimensionality**

When you have different measure groups in a cube, they are almost always going to have different dimensions associated with them; indeed, if you have measure groups that have identical dimensionality, you might consider combining them into a single measure group if it is convenient to do so. As we've already seen, the **Dimension Usage** tab shows us which dimensions have relationships with which measure groups.

When a dimension has a relationship with a measure group it goes without saying that making a selection on that dimension will affect the values that are displayed for measures on that measure group. But what happens to measures when you make a selection on a dimension that has no relationship with a measure group? In fact, you have two options here, controlled by the *IgnoreUnrelatedDimensions*property of a measure group:

* *IgnoreUnrelatedDimensions=False* displays a null value for all members below the root (the intersection of all of the All Members or default members on every hierarchy) of the dimension, except the Unknown member, or
* *IgnoreUnrelatedDimensions=True* repeats the value displayed at the root of the dimension for every member on every hierarchy of the dimension. This is the default state.

The screenshot below shows what happens for two otherwise identical measures from measure groups which have *IgnoreUnrelatedDimensions* set to **True** and to **False** when they're displayed next to a dimension they have no relationship with:

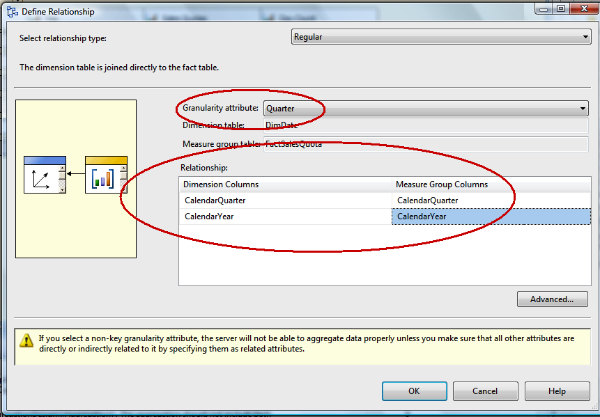


It's usually best to keep *IgnoreUnrelatedDimensions* set to **True** since if the users are querying measures from multiple measure groups, then they don't want some of their selected measures suddenly returning null if they slice by a dimension that has a regular relationship with their other selected measures.

**Handling different granularities**

Even when measure groups share the same dimensions, they may not share the same granularity. For example, we may hold sales information in one fact table down to the day level, but also hold sales quotas in another fact table at the quarter level. If we created measure groups from both these fact tables, then they would both have regular relationships with our Time dimension but at different granularities.

Normally, when you create a regular relationship between a dimension and a measure group, Analysis Services will join the columns specified in the *KeyColumns* property of the key attribute of the dimension with the appropriate foreign key columns of the fact table (note that during processing, Analysis Services won't usually do the join in SQL, it does it internally). However, when you have a fact table of a higher granularity, you need to change the granularity attribute property of the relationship to choose the attribute from the dimension you do want to join on instead:



In the previous screenshot, we can see an amber warning triangle telling us that by selecting a non-key attribute, the server may have trouble aggregating measure values. What does this mean exactly? Let's take a look at the attribute relationships defined on our Time dimension again:

Measures and Measure Groups in Microsoft Analysis Services: Part 2

If we're loading data at the Quarter level, what do we expect to see at the Month and Date level? We can only expect to see useful values at the level of the granularity attribute we've chosen, and for only those attributes whose values can be derived from that attribute; this is yet another good reason to make sure your attribute relationships have been optimized. Below the granularity attribute, we've got the same options regarding what gets displayed as we had with dimensions that have no relationship at all with a measure group: either repeated values or null values. The *IgnoreUnrelatedDimensions* property is again used to control this behavior.

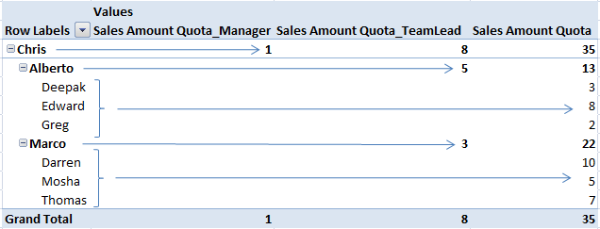
Unfortunately, the default **True** setting for *IgnoreUnrelatedDimensions* is usually not the option you want to use in this scenario (users usually prefer to see nulls below the granularity of a measure in our experience) and this may conflict with how we want to set *IgnoreUnrelatedDimensions* to control the behavior of dimensions which have no relationship with a measure group. There are ways of resolving this conflict such as using MDX Script assignments to set cell values to null or by using the*ValidMeasure()* MDX function, but none are particularly elegant.

**Non-aggregatable measures: a different approach**

We've already seen how we can use parent/child hierarchies to load non-aggregatable measure values into our cube. However, given the problems associated with using parent/child hierarchies and knowing what we now know about measure groups, let's consider a different approach to solving this problem.

A non-aggregatable measure will have, by its very nature, data stored for many different granularities of a dimension. Rather than storing all of these different granularities of values in the same fact table, we could create multiple fact tables for each granularity of value. Having built measure groups from these fact tables, we would then be able to join our dimension to each of them with a regular relationship but at different granularities.

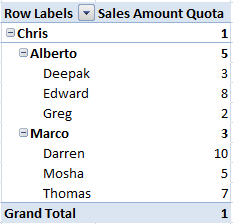
We'd then be in the position of having multiple measures representing the different granularities of a single, logical measure. What we actually want is a single non-aggregatable measure, and we can get this by using MDX Script assignments to combine the different granularities. Let's say we have a regular (non-parent/child) dimension called **Employee** with three attributes **Manager**, **Team Leader** and **Sales Person**, and a logical non-aggregatable measure called **Sales Quota** appearing in three measure groups as three measures called **Sales Amount Quota\_Manager**, **Sales Amount Quota\_TeamLead** and**Sales Amount Quota** for each of these three granularities. Here's a screenshot showing what a query against this cube would show at this stage:



We can combine the three measures into one like this:

**SCOPE([Measures].[Sales Amount Quota]);  
SCOPE([Employee].[Salesperson].[All]);  
THIS=[Measures].[Sales Amount Quota\_TeamLead];  
END SCOPE;  
SCOPE([Employee].[Team Lead].[All]);  
THIS=[Measures].[Sales Amount Quota\_Manager];  
END SCOPE;  
END SCOPE;**

This code takes the lowest granularity measure **Sales Amount Quota**, and then overwrites it twice: the first assignment replaces all of the values above the **Sales Person** granularity with the value of the measure containing **Sales Amount Quota** for **Team Leaders**; the second assignment then replaces all of the values above the **Team Leader** granularity with the value of the measure containing **Sales Quotas** for Managers. Once we've set *Visible*=**False** for the **Sales Amount Quota\_TeamLead** and **Sales Amount Quota\_Manager** measures, we're left with just the **Sales Amount Quota** measure visible, thus displaying the non-aggregatable values that we wanted. The user would then see this:



**Using linked dimensions and measure groups**

Creating linked dimensions and measure groups allows you to share the same dimensions and measure groups across separate Analysis Services databases, and the same measure group across multiple cubes. To do this, all you need to do is to run the 'New Linked Object' wizard from the Cube Editor, either by clicking on the button in the toolbar on the Cube Structure or Dimension Usage tabs, or by selecting it from the right-click menu in the Measures pane of the Cube Structure tab.

Doing this has the advantage of reducing the amount of processing and maintenance needed: instead of having many identical dimensions and measure groups to maintain and keep synchronized, all of which need processing separately, you can have a single object which only needs to be changed and processed once. At least that's the theory—in practice, linked objects are not as widely used as they could be because there are a number of limitations in their use:

* Linked objects represent a static snapshot of the metadata of the source object, and any changes to the source object are not passed through to the linked object. So for example, if you create a linked dimension and then add an attribute to the source dimension, you then have to delete and recreate the linked dimension—there's no option to refresh a linked object.
* You can also import the calculations defined in the MDX Script of the source cube using the wizard. However, you can only import the entire script and this may include references to objects present in the source cube that aren't in the target cube, and which may need to be deleted to prevent errors. The calculations that remain will also need to be updated manually when those in the source cube are changed, and if there are a lot, this can add an unwelcome maintenance overhead.
* A linked measure group can only be used with dimensions from the same database as the source measure group. This isn't a problem when you're sharing measure groups between cubes in the same database, but could be if you wanted to share measure groups across databases.
* As you would expect, when you query a linked measure group, your query is redirected to the source measure group. If the source measure group is on a different server, this may introduce some latency and hurt query performance. Analysis Services does try to mitigate this by doing some caching on the linked measure group's database, though. By default, it will cache data on a per-query basis, but if you change the *RefreshPolicy* property from **ByQuery** to **ByInterval** you can specify a time limit for data to be held in cache.

Linked objects can be useful when cube development is split between multiple development teams, or when you need to create multiple cubes containing some shared data, but, in general, we recommend against using them widely because of these limitations.

**Role-playing dimensions**

It's also possible to add the same dimension to a cube more than once, and give each instance a different relationship to the same measure group. For example, in our Sales fact table, we might have several different foreign key columns that join to our Time dimension table: one which holds the date an order was placed on, one which holds the date it was shipped from the warehouse, and one which holds the date the order should arrive with the customer. In Analysis Services, we can create a single physical Time dimension in our database, which is referred to as a **database dimension**, and then add it three times to the cube to create three 'cube dimensions', renaming each cube dimension to something like Order Date, Ship Date and Due Date. These three cube dimensions are referred to as role-playing dimensions: the same dimension is playing three different roles in the same cube.

Role playing dimensions are a very useful feature. They reduce maintenance overheads because you only need to edit one dimension, and unlike linked dimensions, any changes made to the underlying database dimension are propagated to all of the cube dimensions that are based on it. They also reduce processing time because you only need to process the database dimension once. However, there is one frustrating limitation with role-playing dimensions and that is that while you can override certain properties of the database dimension on a per-cube dimension basis, you can't change the name of any of the attributes or hierarchies of a cube dimension. So if you have a user hierarchy called 'Calendar' on your database dimension, all of your cube dimensions will also have a user hierarchy called 'Calendar', and your users might find it difficult to tell which hierarchy is which in certain client tools (Excel 2003 is particularly bad in this respect) or in reports. Unfortunately, we have seen numerous cases where this problem alone meant role-playing dimensions couldn't be used.

**Dimension/measure group relationships**

So far we've seen dimensions either having no relationship with a measure group or having a regular relationship, but that's not the whole story: there are many different types of relationships that a dimension can have with a measure group. Here's the complete list:

* No relationship
* Regular
* Fact
* Referenced
* Many-to-Many
* Data Mining

**Fact relationships**

Fact or degenerate dimensions are dimensions that are built directly from columns in a fact table, not from a separate dimension table. From an Analysis Services dimension point of view, they are no different from any other kind of dimension, except that there is a special fact relationship type that a dimension can have with a measure group. There are in fact very few differences between a fact relationship and a regular relationship, and they are:

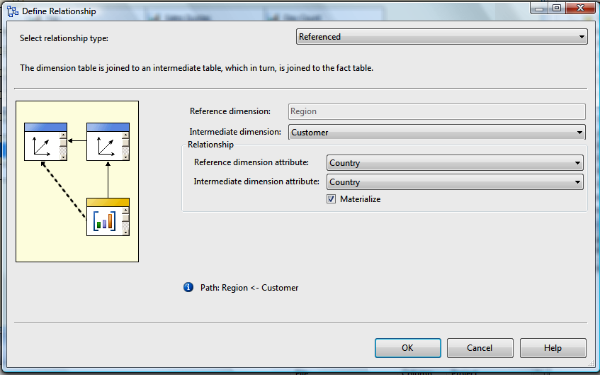
* A fact relationship will result in marginally more efficient SQL being generated when the fact dimension is used in ROLAP drillthrough.
* Fact relationships are visible to client tools in the cube's metadata, so client tools may choose to display fact dimensions differently.
* A fact relationship can only be defined on dimensions and measure groups that are based on the same table in the DSV.
* A measure group can only have a fact relationship with one database dimension. It can have more than one fact relationship, but all of them have to be with cube dimensions based on the same database dimension.

It still makes sense though to define relationships as fact relationships when you can. Apart from the reasons given above, the functionality might change in future versions of Analysis Services and fact relationship types might be further optimized in some way.

**Referenced relationships**

A referenced relationship is where a dimension joins to a measure group through another dimension. For example, you might have a Customer dimension that includes geographic attributes up to and including a customer's country; also, your organization might divide the world up into international regions such as North America, Europe, Middle East and Africa (EMEA), Latin America (LATAM) and Asia-Pacific and so on for financial reporting, and you might build a dimension for this too. If your sales fact table only contained a foreign key for the Customer dimension, but you wanted to analyze sales by international region, you would be able to create a referenced relationship from the Region dimension through the Customer dimension to the Sales measure group.

When setting up a referenced relationship in the **Define Relationship** dialog in the **Dimension Usage** tab, you're asked to first choose the dimension that you wish to join through and then which attribute on the reference dimension joins to which attribute on the intermediate dimension:



When the join is made between the attributes you've chosen on the reference dimension, once again it's the values in the columns that are defined in the *KeyColumns* property of each attribute that you're in fact joining on.

The **Materialize** checkbox is automatically checked, and this ensures maximum query performance by resolving the join between the dimensions at processing time, which can lead to a significant decrease in processing performance. Unchecking this box means that no penalty is paid at processing time but query performance may be worse.

The question you may well be asking yourself at this stage is: why bother to use referenced relationships at all? It is in fact a good question to ask, because, in general, it's better to include all of the attributes you need in a single Analysis Services dimension built from multiple tables rather than use a referenced relationship. The single dimension approach will perform better and is more user-friendly: for example, you can't define user hierarchies that span a reference dimension and its intermediate dimension.

That said, there are situations where referenced relationships are useful because it's simply not feasible to add all of the attributes you need to a dimension. You might have a Customer dimension, for instance, that has a number of attributes representing dates—the date of a customer's first purchase, the date of a customer's tenth purchase, the date of a customer's last purchase and so on. If you had created these attributes with keys that matched the surrogate keys of your Time dimension, you could create multiple, referenced (but not materialized) role-playing Time dimensions joined to each of these attributes that would give you the ability to analyze each of these dates. You certainly wouldn't want to duplicate all of the attributes from your Time dimension for each of these dates in your Customer dimension. Another good use for referenced relationships is when you want to create multiple parent/child hierarchies from the same dimension table

**Data mining relationships**

The data mining functionality of Analysis Services is outside the scope of this article, so we won't spend much time on the data mining relationship type. Suffice to say that when you create an Analysis Services mining structure from data sourced from a cube, you have the option of using that mining structure as the source for a special type of dimension, called a data mining dimension. The wizard will also create a new cube containing linked copies of all of the dimensions and measure groups in the source cube, plus the new data mining dimension, which then has a data mining relationships with the measure groups.

**Summary**

In this part, we focused on how to create new measure groups and handle the problems of different dimensionality and granularity, and looked at the different types of relationships that are possible between dimensions and measure groups.