TUTORIAL: Introduction to Multidimensional Expressions (MDX)

Summary: This tutorial introduces multidimensional expressions (MDX), a highly functional expression syntax for querying multidimensional data in Microsoft SQL Server OLAP Services. It also discusses the structure of OLAP Services cubes and explores the features of MDX.

Contents

Introduction

Microsoft SQL Server OLAP Services provides an architecture for access to multidimensional data. This data is summarized, organized, and stored in multidimensional structures for rapid response to user queries. Through OLE DB for OLAP, a PivotTable Service provides client access to this multidimensional online analytical processing (OLAP) data. For expressing queries to this data, OLE DB for OLAP employs a full-fledged, highly functional expression syntax: multidimensional expressions (MDX).

OLAP Services supports MDX functions as a full language implementation for creating and querying cube data. The querying capabilities of this language are the focus of this article. To demonstrate these capabilities, we will utilize whenever possible simple real-world sample expressions. These are based on the Sales cube in the sample FoodMart database that is installed with OLAP Services. The MDX expressions can thus be run to view the actual output.

OLE DB for OLAP is a set of Component Object Model (COM) interfaces designed to extend OLE DB for efficient access to multidimensional data. ADO has been extended with new objects, collections, and methods that are designed to take advantage of OLE DB for OLAP. These extensions are collectively known as ADO MD (multidimensional) and are designed to provide a simple, high-level object model for accessing tabular and OLAP data.

This description of MDX assumes the reader is familiar with multidimensional data warehousing and OLAP terms.

To run the sample queries you will need:

- Microsoft SQL Server 2000 Analysis Services (or Microsoft SQL Server 7 OLAP Services) properly installed
- Foodmart 2000 (or Foodmart) sample database
- MDX Sample Query Application (usually found on: Start / Programs / Microsoft SQL Server / Analysis Services (or Start / Programs / Microsoft SQL Server / OLAP Services)

0. Multidimensional Expressions

Before talking about MDX and how it queries data, it is worthwhile to give a brief description of the structure of a cube. In addition, we will outline the cube structure of the sample FoodMart Database Sales cube, since all the samples in this article are designed to operate against this sample.

Cube Concepts

Cubes are key elements in online analytic processing. They are subsets of data from the OLAP store, organized and summarized into multidimensional structures. These data summaries provide the mechanism that allows rapid and uniform response times to complex queries.

The fundamental cube concepts to understand are *dimensions* and *measures*.

- Dimensions provide the categorical descriptions by which the measures are separated for analysis.
- Measures identify the numerical values that are summarized for analysis, such as price, cost, or quantity sold.

An important point to note here: The collection of measures forms a dimension, albeit a special one, called "Measures."

Each cube dimension can contain a hierarchy of levels to specify the categorical breakdown available to users. For example, a Store dimension might include the following level hierarchy: Country, State, City, and Store Name. Each level in a dimension is of a finer grain than its parent. Similarly, the hierarchy of a time dimension might include levels for year, quarter, and month.

Multiple hierarchies can exist for a single dimension. For example, take the common breakdown of time. One may want to view a Time dimension by calendar or fiscal periods. In this case the time dimension could contain the time hierarchies fiscal period and calendar year. The fiscal period hierarchy (defined as Time.FiscalYear) could contain the levels Fiscal Year, Fiscal Quarter, and Month. The calendar hierarchy (defined as Time.Calendar) could contain the levels Calendar Year, Calendar Quarter, and Month.

A dimension can be created for use in an individual cube or in multiple cubes. A dimension created for an individual cube is called a *private dimension*, whereas a

dimension that can be used by multiple cubes is called a *shared dimension*. Shared dimensions enable the standardization of business metrics among cubes within a database.

One final important item of note is the concept of a *member*. A member is nothing more than an item in a dimension or measure. A *calculated member* is a dimension member whose value is calculated at run time using a specified expression. Calculated members can also be defined as measures. Only the definitions for calculated members are stored; values are calculated in memory when needed to answer a query. Calculated members enable you to add members and measures to a cube without increasing its size. Although calculated members must be based on a cube's existing data, you can create complex expressions by combining this data with arithmetic operators, numbers, and a variety of functions.

Although the term "cube" suggests three dimensions, a cube can have up to 64 dimensions, including the Measures dimension. In this article the expressions will at most retrieve two dimensions of data, rows and columns. This is analogous to the concept of a two-dimensional matrix.

FoodMart Sales Cube

To enable you to execute the code samples in this article (via the MDX sample application that comes with OLAP Services), they will all be written for the Sales cube in the sample FoodMart database that is installed with OLAP Services. This cube was been designed for the analysis of a chain of grocery stores and their promotions, customers, and products. Tables 1 and 2 outline the dimensions and measures associated with this cube.

Table 1. Sales Cube Dimensions

Dimension name	Level(s)	Description
Customers	Country, State or Province, City, Name	Geographical hierarchy for registered customers of our stores.
Education Level	Education Level	Education level of customer, such as "Graduate Degree" or "High School Degree."
Gender	Gender	Customer gender: "M" or "F"
Marital Status	Marital Status	Customer marital status: "S" or "M"

Product	Product Family Product Department Product Category Product Subcategory Brand Name Product Name	The products that are on sale in the FoodMart stores.
Promotion Media	Media Type	The media used for a promotion, such as Daily Paper, Radio, or Television.
Promotions	Promotion Name	Identifies promotion that triggered the sale.
Store	Store Country Store State Store City Store Name	Geographical hierarchy for different stores in the chain (country, state, city).
Store Size in SQFT	Store Square Feet	Area occupied by store, in square feet.
Store Type	Store Type	Type of store, such as "Deluxe Supermarket" or "Small Grocery."
Time	Years, Quarters, Months	Time period when the sale was made.
Yearly Income	Yearly Income	Income of customer.

Table 2. Sales Cube Measures

Measure name	Description
Unit Sales	Number of units sold.
Store Cost	Cost of goods sold.
Store Sales	Value of sales transactions.
Sales Count	Number of sales transactions.
Store Sales Net	Value of sales transactions less cost of goods sold.
Sales Average	Store sales/sales count. (This is a calculated measure.)

1. Getting Started with MDX

Let's start by outlining one of the simplest forms of an MDX expression, bearing in mind this is for an outline of an expression returning two cube dimensions:

```
SELECT axis specification ON COLUMNS,
axis specification ON ROWS
FROM cube_name
WHERE slicer_specification
```

The axis specification can also be thought of as the member selection for the axis. If a single dimension is required, using this notation, COLUMNS must be returned. For more dimensions, the named axes would be PAGES, CHAPTERS and, finally, SECTIONS. If you desire more generic axis terms over the named terms, you can use the AXIS(index) naming convention. The index will be a zero-based reference to the axis.

The slicer specification on the WHERE clause is actually optional. If not specified, the returned measure will be the default for the cube. Unless you actually query the Measures dimension (as will the first few expressions), you should always use a slicer specification. Doing so defines the slice of the cube to be viewed, hence the term. More will be said about this later.

The simplest form of an axis specification or member selection involves taking the **MEMBERS** of the required dimension, including those of the special Measures dimension:

💻 Query# 1.1

```
SELECT Measures.MEMBERS ON COLUMNS,
[Store].MEMBERS ON ROWS
FROM [Sales]
```

This expression satisfies the requirement to query the recorded measures for each store along with a summary at every defined summary level. Alternatively, it displays the measures for the stores hierarchy. In running this expression, you will

see a row member named "All Stores." The "All" member is generated by default and becomes the default member for the dimension.

The square brackets are optional, except for identifiers with embedded spaces, where they are required. The axis definition can be enclosed in braces, which are used to denote sets. (They are needed only when enumerating sets.)

In addition to taking the **MEMBERS** of a dimension, a single member of a dimension can be selected. If an enumeration of the required members is desired, it can be returned on a single axis:

💻 Query# 1.2

```
SELECT Measures.MEMBERS ON COLUMNS,
{[Store].[Store State].[CA], [Store].[Store State].[WA]} ON ROWS
FROM [Sales]
```

This expression queries the measures for the stores summarized for the states of California and Washington. To actually query the measures for the members making up both these states, one would query the **CHILDREN** of the required members:

💻 Query# 1.3

```
SELECT Measures.MEMBERS ON COLUMNS,
{[Store].[Store State].[CA].CHILDREN,
   [Store].[Store State].[WA].CHILDREN} ON ROWS
FROM [Sales]
```

When running this expression, it is interesting to note that the row set could be expressed by either of the following expressions:

```
[Store State].[CA].CHILDREN
[Store].[CA].CHILDREN
```

The expression uses fully qualified or unique names. Fully qualified member names include their dimension and the parent member of the given member at all levels.

When member names are uniquely identifiable, fully qualified member names are not required. Currently, the unique name property is defined as the fully qualified name. To remove any ambiguities in formulating expressions, the use of unique names should always be adopted.

At this point one should be comfortable with the concept of both **MEMBERS** and **CHILDREN**. To define these terms, the **MEMBERS** function returns the members for the specified dimension or dimension level, and the **CHILDREN** function returns the child members for a particular member within the dimension. Both functions are used often in formulating expressions, but do not provide the ability to drill down to a lower level within the hierarchy. For this task, a function called **DESCENDANTS** is required. This function allows one to go to any level in depth. The syntax for the **DESCENDANTS** function is:

```
DESCENDANTS(member, level [, flags])
```

By default, only members at the specified level will be included. By changing the value of the flag, one can include or exclude descendants or children before and after the specified level. Using **DESCENDANTS** it becomes possible to query the cube for information at the individual store city level, in addition to providing a summary at the state level. Using California as an example:

💻 Query# 1.4

```
SELECT Measures.MEMBERS ON COLUMNS,
{[Store].[Store State].[CA],
   DESCENDANTS([Store].[Store State].[CA], [Store City])} ON ROWS
FROM [Sales]
```

The value of the optional flag can be SELF (the default value whose value can be omitted), BEFORE, AFTER, or BEFORE_AND_AFTER. The statement

```
DESCENDANTS([Store].[Store State].[CA], [Store City], AFTER)
```

would also return the level after the store cities, the actual store names. In this case, specifying BEFORE would return information from the state level.

One final function does require mention for calculated members: **ADDCALCULATEDMEMBERS**. Calculated members are not enumerated if one requests the dimensions members. Calculated members must be explicitly requested by using the **ADDCALCULATEDMEMBERS** function:

💻 Query# 1.5

```
SELECT ADDCALCULATEDMEMBERS(Measures.MEMBERS) ON COLUMNS,
{[Store].[Store State].[CA],
   DESCENDANTS([Store].[Store State].[CA], [Store City])} ON ROWS
FROM [Sales]
```

Slicer specifications

You define the slicer specification with the WHERE clause, outlining the slice of the cube to be viewed. Usually, the WHERE clause is used to define the measure that is being queried. Because the cube's measures are just another dimension, selecting the desired measure is achieved by selecting the appropriate slice of the cube.

If one were required to query the sales average for the stores summarized at the state level, cross referenced against the store type, one would have to define a slicer specification. This requirement can be expressed by the following expression:

💻 Query# 1.6

```
SELECT {[Store Type].[Store Type].MEMBERS} ON COLUMNS,
{[Store].[Store State].MEMBERS} ON ROWS
FROM [Sales]
WHERE (Measures.[Sales Average])
```

As stated, the slicer specification in the WHERE clause is actually a dimensional slice of the cube. Thus the WHERE clause can, and often does, extend to other dimensions. If one were only interested in the sales averages for the year 1997, the WHERE clause would be written as:

```
WHERE (Measures.[Sales Average], [Time].[Year].[1997])
```

It is important to note that slicing is not the same as filtering. Slicing does not affect selection of the axis members, but rather the values that go into them. This is different from filtering, because filtering reduces the number of axis members.

2. Core MDX Functionality

Although the basics of MDX are enough to provide simple queries against the multidimensional data, there are many features of the MDX implementation that make MDX a rich and powerful query tool. These features allow more useful analysis of the cube data and are the focus of the remainder of this article.

Calculated Members and Named Sets

An important concept when working with MDX expressions is that of calculated members and named sets. Calculated members allow one to define formulas and treat the formula as a new member of a specified parent. Within an MDX expression, the syntax for a calculated member is to put the following construction in front of the SELECT statement:

```
WITH MEMBER parent.name AS 'expression'
```

Here, *parent* refers to the parent of the new calculated member name. Because dimensions are organized as a hierarchy, when defining calculated members, their position inside the hierarchy must be defined.

Similarly, for named sets the syntax is:

```
WITH SET set_name AS 'expression'
```

If you need to have named sets and calculated members available for the life of a session and visible to all queries in that session, you can use the CREATE statement with a SESSION scope. As this is part of the cube definition syntax, we will only concentrate on query-specific calculated members and named sets.

The simplest use of calculated members is in defining a new measure that relates already defined measures. This is a common practice for such questions as percentage profit for sales, by defining the calculated measure Profit Percent.

```
WITH MEMBER Measures.ProfitPercent AS
'(Measures.[Store Sales] - Measures.[Store Cost]) /
 (Measures.[Store Cost])', FORMAT_STRING = '#.00%'
```

For defining calculated members there are two properties that you need to know: **FORMAT_STRING** and **SOLVE_ORDER**. **FORMAT_STRING** informs the MDX expression of the display format to use for the new calculated member. The format

expression takes the form of the Microsoft Visual Basic? format function. The use of the percent symbol (%) specifies that the calculation returns a percentage and should be treated as such, including multiplication by a factor of 100.

The **SOLVE_ORDER** property is used in defining multiple calculated members or named sets. This property helps decide the order in which to perform the evaluations. The calculated member or named set with the highest solve order value will be the first to be evaluated. The default value for the solve order is zero.

With calculated members one can easily define a new Time member to represent the first and second halves of the year:

```
WITH MEMBER [Time].[First Half 97] AS
   '[Time].[1997].[Q1] + [Time].[1997].[Q2]'
MEMBER [Time].[Second Half 97] AS
   '[Time].[1997].[Q3] + [Time].[1997].[Q4]'
```

Using all this, if one were required to display the individual store's sales percentage profit for each quarter and half year, the MDX expression would read:

💻 Query# 2.1

```
WITH MEMBER Measures.ProfitPercent AS
'(Measures.[Store Sales] - Measures.[Store Cost]) /
 (Measures.[Store Cost])', FORMAT_STRING = '#.00%', SOLVE_ORDER = 1
MEMBER [Time].[First Half 97] AS
   '[Time].[1997].[Q1] + [Time].[1997].[Q2]'
MEMBER [Time].[Second Half 97] AS
   '[Time].[1997].[Q3] + [Time].[1997].[Q4]'
SELECT {[Time].[First Half 97], [Time].[Second Half 97],
   [Time].[1997].CHILDREN} ON COLUMNS,
{[Store].[Store Name].MEMBERS} ON ROWS
FROM [Sales]
```

```
WHERE (Measures.ProfitPercent)
```

The use of the **SOLVE_ORDER** property deserves explanation. Since the **SOLVE_ORDER** for the calculated measure, profit percent, is defined as being greater than zero, its value will be evaluated first. In evaluating the percentage profit over the new calculated Time members, First Half 97 and Second Half 97, the values for store sales and store cost over these time periods are calculated. The calculation for profit percent will then be correctly based on the values of store sales and store cost over the new calculated Time members.

If the solve order were defined to evaluate the new calculated time members first, the profit percent would be calculated for each quarter prior to adding the quarter's results. This would have the effect of adding together the two percentages for each quarter rather than calculating the percent for the whole defined time period. This would obviously yield incorrect results.

In all these expressions, the new calculated member has been directly related to a dimension. This doesn't have to be the case; calculated members can also be related to a member within the hierarchy, as the next sample shows:

💻 Query# 2.2

```
WITH MEMBER [Time].[1997].[H1] AS
   '[Time].[1997].[Q1] + [Time].[1997].[Q2]'
MEMBER [Time].[1997].[H2] AS
   '[Time].[1997].[Q3] + [Time].[1997].[Q4]'
SELECT {[Time].[1997].[H1], [Time].[1997].[H2]} ON COLUMNS,
[Store].[Store Name].MEMBERS ON ROWS
FROM [Sales]
WHERE (Measures.Profit)
```

The definition of named sets follows the exact same syntax as that for calculated members. A named set could be defined that contains the first quarter for each year, within the time dimension. Using this, you can display store profit for the first quarter of each year:

💻 Query# 2.3

```
WITH SET [Quarter1] AS
   'GENERATE([Time].[Year].MEMBERS, {[Time].CURRENTMEMBER.FIRSTCHILD})'
SELECT [Quarter1] ON COLUMNS,
[Store].[Store Name].MEMBERS ON ROWS
FROM [Sales]
WHERE (Measures.[Profit])
```

The function **FIRSTCHILD** takes the first child of the specified member, in this case the first quarter of each year. A similar function called **LASTCHILD** also exists, which will take the last child of a specified member.

In the following sections, more will be said about calculated members and named sets, including the use of the **CURRENTMEMBER** function. In using calculated members and named sets, the ability to perform hierarchical navigation is what really extends their usage. The next section covers this capability.

Hierarchical Navigation

In constructing MDX expressions, it is often necessary to relate a *current members* value to others in the hierarchy. MDX has many methods that can be applied to a member to traverse this hierarchy. Of these, the most commonly used methods are **PREVMEMBER**, **NEXTMEMBER**, **CURRENTMEMBER**, and **PARENT**. Others also exist, including **FIRSTCHILD** and **LASTCHILD**.

Consider the common business need for calculating the sales of a product brand as a percentage of the sales of that product within its product subcategory. To satisfy this requirement, you must calculate the percentage of the sales of the current product, or member, compared to that of its parent. The expression for this calculated member can be derived using the **CURRENTMEMBER** and **PARENT** functions.

💻 Query# 2.4

```
WITH MEMBER MEASURES.PercentageSales AS
   '([Product].CURRENTMEMBER, Measures.[Unit Sales]) /
```

```
   ([Product].CURRENTMEMBER.PARENT, Measures.[Unit Sales])',
   FORMAT_STRING = '#.00%'
SELECT {MEASURES.[Unit Sales], MEASURES.PercentageSales} ON COLUMNS,
[Product].[Brand Name].MEMBERS ON ROWS
FROM [Sales]
```

The **CURRENTMEMBER** function returns the current member along a dimension during an iteration. The **PARENT** function returns the parent of a member.

In this expression, the **PARENT** of the **CURRENTMEMBER** of the product brand name is the product subcategory (see Table 1). If you needed to rewrite this calculation to query product sales by brand name as a percentage of the sales within the product department, you could define the calculated measure as:

```
WITH MEMBER MEASURES.PercentageSales AS
   '([Product].CURRENTMEMBER, Measures.[Unit Sales]) /
   ([Product].CURRENTMEMBER.PARENT.PARENT.PARENT,
      Measures.[Unit Sales])',
```

The repeated use of the **PARENT** function can be replaced by calculating the appropriate ancestor of the **CURRENTMEMBER**. The appropriate function for this is **ANCESTOR**, which returns the ancestor of a member at the specified level:

```
WITH MEMBER MEASURES.PercentageSales AS
   '([Product].CURRENTMEMBER, Measures.[Unit Sales]) /
   (ANCESTOR([Product].CURRENTMEMBER, [Product Category]),
   MEASURES.[Unit Sales])'
```

If this analysis were needed for promotion rather than for products, there could be an issue with the fact that a member exists for promotions representing sales for which no promotion applies. In this case, the use of named sets and the function **EXCEPT** will easily allow an expression to be formulated that shows the percentage of sales for each promotion compared only to other promotions:

💻 Query# 2.5

```
WITH SET [PromotionSales] AS
   'EXCEPT({[Promotions].[All Promotions].CHILDREN},
   {[Promotions].[No Promotion]})'
MEMBER Measures.PercentageSales AS
   '([Promotions].CURRENTMEMBER, Measures.[Unit Sales]) /
   SUM([PromotionSales], MEASURES.[Unit Sales])',
   FORMAT_STRING = '#.00%'
SELECT {Measures.[Unit Sales], Measures.PercentageSales} ON COLUMNS,
[PromotionSales] ON ROWS
FROM [Sales]
```

The **EXCEPT** function finds the difference between two sets, optionally retaining duplicates. The syntax for this function is:

```
EXCEPT(set1, set2 [, ALL])
```

Duplicates are eliminated from both sets prior to finding the difference. The optional ALL flag retains duplicates.

The concept of taking the current member within a set is also useful when using the **GENERATE** function. The **GENERATE** function iterates through all the members of a set, using a second set as a template for the resultant set.

Consider the requirement to query unit sales for promotions for each year and, in addition, break down the yearly information into its corresponding quarter details. Not knowing what years there are to display, one would need to generate the axis information based on the members of the time dimension for the yearly level, along with the corresponding members of the quarterly level:

💻 Query# 2.6

```
SELECT {GENERATE([Time].[Year].MEMBERS,
   {[Time].CURRENTMEMBER, [Time].CURRENTMEMBER.CHILDREN})} ON COLUMNS,
[Promotions].[All Promotions].CHILDREN ON ROWS
```

```
FROM [Sales]

WHERE (Measures.[Unit Sales])
```

Similarly, if you wanted to display the unit sales for all promotions and stores within the states of California and Washington, you would need to enumerate all the stores for each state. Not only would this be a lengthy expression to derive, but modifying the states in the expression would require extensive rework. The **GENERATE** function can be used to allow new states to be easily added to or removed from the expression:

Query# 2.7

```
SELECT {GENERATE({[Store].[CA], [Store].[WA]},
DESCENDANTS([Store].CURRENTMEMBER, [Store Name]))} ON COLUMNS,

[Promotions].[All Promotions].CHILDREN ON ROWS

FROM [Sales]

WHERE (Measures.[Unit Sales])
```

Another common business problem involves the need to show growth over a time period, and here the **PREVMEMBER** function can be used. If one needed to display sales profit and the incremental change from the previous time member for all months in 1997, the MDX expression would read:

Query# 2.8

```
WITH MEMBER Measures.[Profit Growth] AS

   '(Measures.[Profit])  - (Measures.[Profit], [Time].PREVMEMBER)',

   FORMAT_STRING = '###,###.00'

SELECT {Measures.[Profit], Measures.[Profit Growth]} ON COLUMNS,

{DESCENDANTS([Time].[1997], [Month])} ON ROWS

FROM [Sales]
```

Using **NEXTMEMBER** in this expression would show sales for each month compared with those of the following month. You can also use the **LEAD** function, which returns the member located a specified number of positions following a specific

member along the member's dimension. The syntax for the **LEAD** function is as follows:

```
member.LEAD(number)
```

If the number given is negative a prior member is returned; if it is zero the current member is returned. This capability allows for replacing the **PREV**, **NEXT**, and **CURRENT** navigation with the more generic **LEAD(-1)**, **LEAD(1)**, and **LEAD(0)**. A similar function called **LAG** exists, such that **LAG(n)** is equivalent to **LEAD(-n)**.

Time Series Functions

This last sample expression brings up an important part of data analysis: time period analysis. Here we can examine how we did this month compared to the same month last year, or how we did this quarter compared to the last quarter. MDX provides a powerful set of time series functions for time period analysis.

While they are called time series functions and their most common use is with the Time dimension, most of them work equally well with any other dimension, and scenarios exist where these functions can be useful on other dimensions. The *x*TD (**YTD**, **MTD**, **QTD**, **WTD**) functions are exceptions to this flexibility; they are only applicable to the Time dimension. These functions refer to Year-, Quarter-, Month-, and Week-to-date periods and will be discussed later in this section.

Including the *x*TD functions, the important time series functions that we demonstrate here are **PARALLELPERIOD**, **CLOSINGPERIOD**, **OPENINGPERIOD**, and **PERIODSTODATE**.

Continuing in the vein of time period analysis, **PARALLELPERIOD** allows one to easily compare member values of a specified member with those of a member in the same relative position in a prior period. (The prior period is the prior member value at a higher specified level in the hierarchy.) For example, one would compare values from one month with those of the same relative month in the previous year. The expression using the **PREVMEMBER** function compared growth with that of the previous month; **PARALLELPERIOD** allows for an easy comparison of growth with that of the same period in the previous quarter:

Query# 2.9

```
WITH MEMBER Measures.[Profit Growth] AS '(Measures.[Profit]) –
   (Measures.[Profit], PARALLELPERIOD([Time].[Quarter]))',
   FORMAT_STRING = '###,###.00'
SELECT {Measures.[Profit], Measures.[Profit Growth]} ON COLUMNS,
{DESCENDANTS([Time].[1997], [Month])} ON ROWS
FROM [Sales]
```

If you were to run this expression, for the first quarter the profit growth would be equivalent to the profit. Because the sales cube only holds sales for 1997 and 1998, the quarter growth for the first quarter cannot really be measured. In these situations, an appropriate value of zero is used for parallel periods beyond the cube's range.

The exact syntax for **PARALLELPERIOD** is:

```
PARALLELPERIOD(level, numeric_expression, member)
```

All parameters are optional. The numeric expression allows one to specify how many periods one wishes to go back. One could just as easily have written the previous expression to traverse back to the same month in the previous half year:

```
PARALLELPERIOD([Time].[Quarter], 2)
```

The functions **OPENINGPERIOD** and **CLOSINGPERIOD** have similar syntax:

```
OPENINGPERIOD(level, member)
CLOSINGINGPERIOD(level, member)
```

Their purpose is to return the first or last sibling among the descendants of a member at a specified level. All function parameters are optional. If no member is specified, the default is **[Time].CURRENTMEMBER**. If no level is specified, it is the level below that of member that will be assumed.

For seasonal sales businesses, it might be important to see how much sales increase after the first month in the season. Using a quarter to represent the season, one can measure the unit sales difference for each month compared with the opening month of the quarter:

💻 Query# 2.10

```
WITH MEMBER Measures.[Sales Difference] AS
   '(Measures.[Unit Sales]) – (Measures.[Unit Sales],
   OPENINGPERIOD([Time].[Month], [Time].CURRENTMEMBER.PARENT))',
   FORMAT_STRING = '###,###.00'
SELECT {Measures.[Unit Sales], Measures.[Sales Difference]} ON COLUMNS,
{DESCENDANTS([Time].[1997], [Month])} ON ROWS
FROM [Sales]
```

In deriving the calculated member "Sales Difference," the opening period at the month level is taken for the quarter in which the month resides. Replacing **OPENINGPERIOD** with **CLOSINGPERIOD** will show sales based on the final month of the specified season.

The final set of time series functions we'll discuss are the *x*TD functions. Before describing these functions, however, we need to consider the **PERIODSTODATE** function, as the *x*TD functions are merely special cases of **PERIODSTODATE**.

PERIODSTODATE returns a set of periods (members) from a specified level starting with the first period and ending with a specified member. This function becomes very useful when combined with such functions as **SUM**, as will be shown shortly. The syntax for the **PERIODSTODATE** function is:

```
PERIODSTODATE(level, member)
```

If member is not specified, the member **[Time].CURRENTMEMBER** is assumed. As a simple example, to define a set of all the months up to and including the month of June for the year 1997, the following definition could be used:

```
PERIODSTODATE([Time].[Year], [Time].[1997].[Q2].[6])
```

Before we continue with this discussion, the **SUM** function warrants a brief description. This function returns the sum of a numeric expression evaluated over a set. For example, one can easily display the sum of unit sales for the states of California and Washington with the following simple expression:

```
SUM({[Store].[Store State].[CA], [Store].[Store State].[WA]},
   Measures.[Unit Sales])
```

More will be said about **SUM** and other numeric functions shortly. Using the functions **SUM** and **PERIODSTODATE**, it becomes easy to define a calculated member that displays year-to-date information. Take for example the requirement to query monthly year-to-date sales for each product category in 1997. The measure to be displayed is the sum of the current time member over the year level:

```
PERIODSTODATE([Time].[Year], [Time].CURRENTMEMBER)
```

This is easily abbreviated by the expression **YTD()**:

💻 Query# 2.11

```
WITH MEMBER Measures.YTDSales AS
   'SUM(YTD(), Measures.[Store Sales])', FORMAT_STRING = '#.00'
SELECT {DESCENDANTS([Time].[1997], [Month])} ON COLUMNS,
{[Product].[Product Category].MEMBERS} ON ROWS
FROM [Sales]
WHERE (Measures.YTDSales)
```

Calculating quarter-to-date information is easily achieved by using the **QTD()** instead of **YTD()** function. It can also be achieved by using the **PERIODSTODATE** function with the level defined as [Time].[Quarter]. Similar rules apply for using the **MTD()** and **WTD()** functions. Using **PERIODSTODATE** may be somewhat long-winded compared to using the *x*TD functions, but does offer greater flexibility. In addition, it can also be used with non-time dimensions.

Tuples and CROSSJOIN

In many cases a combination of members from different dimensions are enclosed in brackets. This combination is known as a *tuple* and is used to display multiple dimensions onto a single axis. In the case of a single member tuple, the brackets can be omitted.

The main advantage of tuples becomes apparent when more than two axes are required. Say that one needs to query unit sales for the product categories to each city for each quarter. This query is easily expressed by the following MDX expression, but unless one is mapping the data into a 3D graph, is impossible to display.

💻 Query# 2.11 /* non-displayable on MDX Sample App */

```
SELECT [Product].[Product Family].MEMBERS ON COLUMNS,
[Customers].[City].MEMBERS ON ROWS,
[Time].[Quarter].MEMBERS ON PAGES
FROM [Sales]
WHERE (Measures.[Unit Sales])
```

To allow this query to be viewed in a matrix format one would need to combine the customer and time dimensions onto a single axis. This is a tuple—a combination of dimension members coming from different dimensions. The syntax for a tuple is as follows:

```
(member_of_dim_1, member_of_dim_2, ..., member_of_dim_n)
```

The only problem here is to enumerate all the possible combinations of customer cities and yearly quarters. Fortunately, MDX supports this operation through the use of the **CROSSJOIN** function. This function produces all combinations of two sets. The previous expression can now be rewritten to display the results on two axes as follows:

💻 Query# 2.12

```
SELECT [Product].[Product Family].MEMBERS ON COLUMNS,
{CROSSJOIN([Customers].[City].MEMBERS, [Time].[Quarter].MEMBERS)}
   ON ROWS
FROM [Sales]
WHERE (Measures.[Unit Sales])
```

Filtering and Sorting

As mentioned earlier, the concept of slicing and filtering are very distinct. Expectably, filtering actually reduces the number of members on an axis. However, all slicing does is affect the values that go into the axis members, and does not actually reduce their number.

Quite often an MDX expression will retrieve axis members when there are no values to be placed into them. This brings up the easiest form of filtering—removing empty members from the axis. You can achieve this filtering through the use of the NON EMPTY clause. You just need to use NON EMPTY as part of the axis definition. Looking at the previous expression, you can easily remove empty tuples from the axis by reformulating the MDX expression as:

💻 Query# 2.13

```
SELECT [Product].[Product Family].MEMBERS ON COLUMNS,
NON EMPTY {CROSSJOIN([Customers].[City].MEMBERS,
   [Time].[Quarter].MEMBERS)} ON ROWS
FROM [Sales]
WHERE (Measures.[Unit Sales])
```

For more specific filtering, MDX offers the **FILTER** function. This function returns the set that results from filtering according to the specified search condition. The format of the **FILTER** function is:

```
FILTER(set, search_condition)
```

Consider the following simple expression comparing sales profit in 1997 for each city based against the store type:

💻 Query# 2.14

```
SELECT {[Store Type].[Store Type].MEMBERS} ON COLUMNS,
{[Store].[Store City].MEMBERS} ON ROWS
FROM [Sales]
```

```
WHERE (Measures.[Profit], [Time].[Year].[1997])
```

If one were only interested in viewing top-performing cities, defined by those whose unit sales exceed 25,000, a filter would be defined as:

💻 Query# 2.15

```
SELECT NON EMPTY {[Store Type].[Store Type].MEMBERS} ON COLUMNS,
FILTER({[Store].[Store City].MEMBERS},
   (Measures.[Unit Sales], [Time].[1997])>25000) ON ROWS
FROM [Sales]
WHERE (Measures.[Profit], [Time].[Year].[1997])
```

This can easily be extended to query those stores with a profit percentage less than that for all stores in their state; which stores' profit margins are falling behind the state average for each store type:

💻 Query# 2.16

```
WITH MEMBER Measures.[Profit Percent] AS
   '(Measures.[Store Sales]-Measures.[Store Cost]) /
   (Measures.[Store Cost])', FORMAT_STRING = '#.00%'
SELECT NON EMPTY {[Store Type].[Store Type].MEMBERS} ON COLUMNS,
FILTER({[Store].[Store City].MEMBERS},
   (Measures.[Profit Percent], [Time].[1997]) <
   (Measures.[Profit Percent], [Time].[1997],
   ANCESTOR([Store].CURRENTMEMBER, [Store State]))) ON ROWS
FROM [Sales]
WHERE (Measures.[Profit Percent], [Time].[Year].[1997])
```

During cube queries, all the members in a dimension have a natural order. This order can be seen when one utilizes the inclusion operator, a colon. Consider the simple expression displaying all measures for the store cities:

💻 Query# 2.17

```
SELECT Measures.MEMBERS ON COLUMNS,
[Store].[Store City].MEMBERS ON ROWS
FROM [Sales]
```

The first city listed is Vancouver, followed by Victoria and then Mexico City. The natural order is not very apparent, because we do not know the member from the parent level. If you were only interested in the cities listed between Beverly Hills and Spokane (the cities in the United States), you could write:

💻 Query# 2.18

```
SELECT Measures.MEMBERS ON COLUMNS,
{[Store].[Store City].[Beverly Hills]:[Spokane]} ON ROWS
FROM [Sales]
```

What if you wanted all the stores in this list sorted by the city name? It's a common reporting requirement. MDX provides this functionality through the **ORDER** function. The full syntax for this function is:

```
ORDER(set, expression [, ASC | DESC | BASC | BDESC])
```

The expression can be numeric or a string expression. The default sort order is ASC. The "B" prefix indicates the hierarchical order can be broken. Hierarchized ordering first arranges members according to their position in the hierarchy, and then it orders each level. The nonhierarchized ordering arranges members in the set without regard to the hierarchy.

Working on the previous sample to order the set regardless of the hierarchy, we see the following:

💻 Query# 2.19

```
SELECT Measures.MEMBERS ON COLUMNS,
ORDER({[Store].[Store City].[Beverly Hills]:[Spokane]},
```

```
   [Store].CURRENTMEMBER.Name, BASC) ON ROWS
FROM [Sales]
```

Here the property **Name** is used. This returns the name of a level, dimension, member, or hierarchy. A similar property, **UniqueName**, exists to return the corresponding unique name.

More often than not, the actual ordering is based on an actual measure. The same query can easily be changed to display the all-city information based on the sales count; you query the cities' sales information ordered by their sales performance:

💻 Query# 2.20

```
SELECT Measures.MEMBERS ON COLUMNS,
ORDER({[Store].[Store City].MEMBERS},
   Measures.[Sales Count], BDESC) ON ROWS
FROM [Sales]
```

Top and Bottom Performance Analysis

When displaying information such as the best-selling cities based on unit sales, it may be beneficial to limit the query to, say, the top dozen. MDX can support this operation using a function called **HEAD**. This function is very simple and returns the first members in the set based on the number that one requests. A similar function called **TAIL** exists that returns a subset from the end of the set. Taking the previous expression of best-selling stores as an example, the top dozen store cities can be queried by the expression:

💻 Query# 2.21

```
SELECT Measures.MEMBERS ON COLUMNS,
HEAD(ORDER({[Store].[Store City].MEMBERS},
   Measures.[Sales Count], BDESC), 12) ON ROWS
FROM [Sales]
```

Expectably, because this is such a common request, MDX supports a function called **TOPCOUNT** to perform such a task. The syntax for the **TOPCOUNT** function is:

```
TOPCOUNT(set, count, numeric_expression)
```

The previous expression can easily be rewritten:

💻 Query# 2.22

```
SELECT Measures.MEMBERS ON COLUMNS,
TOPCOUNT({[Store].[Store City].MEMBERS}, 12,
   Measures.[Sales Count]) ON ROWS
FROM [Sales]
```

This expression is very simple, but it doesn't have to be. An MDX expression can be calculated that displays the top half-dozen cities, based on sales count, and how much all the other cities combined have sold. This expression also shows another use of the **SUM** function, in addition to named sets and calculated members:

💻 Query# 2.23

```
WITH SET TopDozenCities AS
   'TOPCOUNT([Store].[Store City].MEMBERS, 12, [Sales Count])'
MEMBER [Store].[Other Cities] AS
   '([Store].[All Stores], Measures.CURRENTMEMBER) –
   SUM(TopDozenCities, Measures.CURRENTMEMBER)'
SELECT {Measures.MEMBERS} ON COLUMNS,
{TopDozenCities, [Store].[Other Cities]} ON ROWS
FROM [Sales]
```

Other functions exist for the top filter processing. They are **TOPPERCENT**, returning the top elements whose cumulative total is at least a specified percentage, and **TOPSUM**, returning the top elements whose cumulative total is at least a specified

value. There is also a series of **BOTTOM** functions, returning the bottom items in the list.

The preceding expression can easily be modified to display the list of cities whose sales count accounts for 50 percent of all the sales:

💻 Query# 2.24

```
SELECT Measures.MEMBERS ON COLUMNS,
TOPPERCENT({[Store].[Store City].MEMBERS}, 50,
   Measures.[Sales Count]) ON ROWS
FROM [Sales]
```

In all of these expressions, the row axis has been defined as the members of the Measures dimension. Again, this does not have to be the case. One can easily formulate an expression that shows the breakdown of the sales counts for the store types:

💻 Query# 2.25

```
SELECT [Store Type].MEMBERS ON COLUMNS,
TOPPERCENT({[Store].[Store City].MEMBERS}, 50,
   Measures.[Sales Count]) ON ROWS
FROM [Sales]
WHERE (Measures.[Unit Sales])
```

Numeric Functions and Conditional Expressions

MDX supports many numeric functions. The **SUM** function is an example that we have seen already in this article. **COUNT** is another important function, which simply counts the number of tuples in a set.

The **COUNT** function has two options: including and excluding empty cells. **COUNT** is useful for such operations as deriving the number of customers that purchased a

particular product category. Looking at unit sales, products within the number of customers who purchased products can be derived by counting the number of tuples of the unit sales and customer names. Excluding empty cells is necessary to restrict the count to those customers for which there are unit sales within the product category:

💻 Query# 2.26

```
WITH MEMBER Measures.[Customer Count] AS
   'COUNT(CROSSJOIN({Measures.[Unit Sales]},
   [Customers].[Name].MEMBERS), EXCLUDEEMPTY)'
SELECT {Measures.[Unit Sales], Measures.[Customer Count]} ON COLUMNS,
[Product].[Product Category].MEMBERS ON ROWS
FROM [Sales]
```

Other numeric functions exist for such operations as calculating the average, median, maximum, minimum, variance, and standard deviation of tuples in a set based on a numeric value. These functions are **AVG**, **MEDIAN**, **MAX**, **MIN**, **VAR**, and **STDDEV**, respectively. The format for all these functions is the same:

```
FUNCTION(set, numeric_value_expression)
```

Taking the **MAX** function as an example, one can easily analyze each product category to see the maximum unit sales for a one-month period in a particular year:

💻 Query# 2.27

```
WITH MEMBER Measures.[Maximum Sales] AS
   'MAX(DESCENDANTS([Time].CURRENTMEMBER, [Time].[Month]),
   Measures.[Unit Sales])'
SELECT {[Time].[1997]} ON COLUMNS,
[Product].[Product Category].MEMBERS ON ROWS
FROM [Sales]
WHERE (Measures.[Maximum Sales])
```

Replacing **MAX** with **AVG** would give the average unit sales for the product category in a month for the given time period. Because the **AVG** function is dependent on a count of the members for which the average is being taken, it is important to note that the function excludes empty cells.

Taking the previous sample further, if one needed to not only view the unit sales for 1997 and the average for the month, but also see the month count for which the average is derived, the MDX expression would be formulated as:

💻 Query# 2.28

```
WITH MEMBER [Time].[Average Sales] AS
   'AVG(DESCENDANTS([Time].[1997], [Time].[Month]))'
MEMBER [Time].[Average Count] AS
   'COUNT(DESCENDANTS([Time].[1997], [Time].[Month]),EXCLUDEEMPTY)'
SELECT {[Time].[1997], [Time].[Average Sales], [Time].[Average Count]}
   ON COLUMNS,
[Product].[Brand Name].MEMBERS ON ROWS
FROM [Sales]
WHERE (Measures.[Unit Sales])
```

In addition to these built-in functions, MDX allows you to create and register your own functions that operate on multidimensional data. These functions, called "user-defined functions" (UDFs), can accept arguments and return values in the MDX syntax. UDFs can be created in any language that supports COM. In addition to this, MDX supports many functions in the Microsoft Visual Basic for Applications (VBA) Expression Services library. This library is included with OLAP Services and is automatically registered.

Take, for example, the VBA function **INSTR**. This function compares two strings to return the position of the second string within the first. With this function you can query the measures for the store types where the actual store type contains the word "supermarket," stores defined as a type of supermarket:

Query# 2.29

```
SELECT Measures.MEMBERS ON COLUMNS,
FILTER({[Store Type].[Store Type].MEMBERS},
   VBA!INSTR(1, [Store Type].CURRENTMEMBER.Name, "Supermarket") > 0)
   ON ROWS
FROM [Sales]
```

Here the VBA clause is not needed. The purpose of this clause is to fully qualify the origin of the function. This qualification is only needed when the same function exists in multiple declared libraries.

MDX supports the conditional clauses **Immediate IF**, or **IIF**. The **IIF** function is used to test any search condition and choose one value or another based on whether the test is true or false.

Consider an earlier expression that queried profit growth over the previous time period. If one needed to rewrite this query to display the growth percentage, the expression could be formulated as:

Query# 2.30

```
WITH MEMBER Measures.[Profit Growth] AS '(Measures.[Profit]) /
   (Measures.[Profit], [Time].PREVMEMBER)', FORMAT_STRING = '#.00%'
SELECT {Measures.[Profit], Measures.[Profit Growth]} ON COLUMNS,
{DESCENDANTS([Time].[1997], [Month])} ON ROWS
FROM [Sales]
```

The only problem with this expression would be that the first time period causes a division by zero. This problem can easily be avoided by using **IIF** and checking for the existence of an empty cell:

Query# 2.31

```
WITH MEMBER Measures.[Profit Growth] AS
   'IIF(ISEMPTY([Time].PREVMEMBER), 1, (Measures.[Profit]) /
   (Measures.[Profit], [Time].PREVMEMBER))', FORMAT_STRING = '#.00%'
SELECT {Measures.[Profit], Measures.[Profit Growth]} ON COLUMNS,
{DESCENDANTS([Time].[1997], [Month])} ON ROWS
FROM [Sales]
```

This functionality can also be achieved with a function called **COALESCEEMPTY**, which coalesces an empty cell value to a number or a string and returns the coalesced value. In this case, the empty cell from the previous time member would be coalesced to the value for the current time member:

💻 Query# 2.32

```
WITH MEMBER Measures.[Profit Growth] AS '(Measures.[Profit]) /
   COALESCEEMPTY((Measures.[Profit], [Time].PREVMEMBER),
   Measures.[Profit])', FORMAT_STRING = '#.00%'
SELECT {Measures.[Profit], Measures.[Profit Growth]} ON COLUMNS,
{DESCENDANTS([Time].[1997], [Month])} ON ROWS
FROM [Sales]
```

The function returns the first (from the left) nonempty value expression in the list of value expressions. If all are empty, it returns the empty cell value.

3. Conclusion

This has been an overview of the capabilities of MDX for data querying and analysis. Although all the sample expressions are given for a sales database, this can very easily be mapped over to Inventory Figures, Manufacturing Data, Financial Information, or Web Site Logs. The possibilities are endless. The type of data analysis that can be easily performed by MDX should be more than enough to warrant the effort in transforming data into OLAP cubes.

Remember that MDX can perform very sophisticated queries. Building good MDX skills will enable one to write elegant-looking queries that will get the result set quickly. In the OLAP space, the developers who are effective will be those who are intimate with MDX.