Лекция 5

Тема « The WITH Clause and Calculated Members »

The WITH Clause and Calculated Members

Often business needs involve calculations that must be formulated within the scope of a specific query. The MDX WITH clause provides you with the ability to create such calculations and use them within the context of the query. In addition, you can also retrieve data from outside the context of the current cube using the LookupCube MDX function.

Typical calculations that are created using the WITH clause are named sets and calculated members. In addition to these, the WITH clause provides you with functionality to define cell calculations, load a cube into an Analysis Server cache for improving query performance, alter the contents of cells by calling functions in external libraries, and additional advanced capabilities such as solve order and pass order. You learn about named sets, calculated members, and calculated measures in this chapter. Chapter 10 covers the rest.

The syntax of the WITH clause is :

[WITH < formula_expression > [, < formula_expression > ...]]

You can specify several calculations in one WITH clause. The formula_expression will vary depending upon the type of calculations. Calculations are separated by commas.

Named Sets

As you learned earlier, a set is a collection of tuples. A set expression, even though simple, can often be quite lengthy and this might make the query appear to be complex and unreadable. MDX provides you with the capability of dynamically defining sets with a specific name so that the name can be used within the query. Think of it as an alias for the collection of tuples in the set. This is called a *named set* . A named set is nothing but an alias for an MDX set expression that can be used anywhere within the query as an alternative to specifying the actual set expression.

Consider the case where you have customers in various countries. Suppose you want to retrieve the Sales information for customers in Europe. Your MDX query would look like this:

```
SELECT Measures.[Internet Sales Amount] ON COLUMNS,
{[Customer].[Country].[Country]. & [France],
[Customer].[Country].[Country]. & [Germany],
[Customer].[Country].[Country]. & [United Kingdom]} ON ROWS
FROM [Adventure Works]
```

This query is not too lengthy, but you can imagine a query that would contain a lot of members and functions being applied to this specific set several times within the query. Instead of specifying the complete set every time it ' s used in the query, you can create a named set and then use it in the query as follows:

```
WITH SET [EUROPE] AS '{[Customer].[Country].[Country]. & [France],
[Customer].[Country].[Country]. & [Germany],[Customer].[Country].[Country].
& [United Kingdom]}'
SELECT Measures.[Internet Sales Amount] ON COLUMNS,
[EUROPE] ON ROWS
FROM [Adventure Works]
```

The formula_expression for the WITH clause with a named set is

Formula_expression := [DYNAMIC] SET < set_alias_name > AS ['] < set > [']

The set_alias_name can be any alias name and is typically enclosed within square brackets. Note the keywords SET and AS that are used in this expression to specify a named set. The keyword DYNAMIC is optional. The actual set of tuples does not have to be enclosed within single quotes. The single quotes are still available for backward compatibility with Analysis Services 2000.

You can create named sets within an MDX query using the WITH clause shown in this section. You can also create them within a session using the CREATE SET option. Additionally, you can create them globally in MDX scripts using CREATE statements. Sets can be evaluated statically or dynamically at query execution time. Hence the keyword DYNAMIC is typically used within MDX scripts to be evaluated at the query execution time. Chapter 10 shows how to create a DYNAMIC set.

Calculated Members

Calculated members are calculations specified by MDX expressions. They are resolved as a result of MDX expression evaluation rather than just by the retrieval of the original fact data. A typical example of a calculated member is the calculation of year - to - date sales of products. Let ' s say the fact data only contains sales information of products for each month and you need to calculate the year - to - date sales. You can do this with an MDX expression using the WITH clause.

The formula_expression of the WITH clause for calculated members is :
Formula_expression := MEMBER < MemberName > AS [‘] < MDX_Expression > [‘],
[, SOLVE_ORDER = < integer >]
[, < CellProperty > = < PropertyExpression >]
MDX uses the keywords MEMBER and AS in the WITH clause for creating calculated members. The MemberName should be a fully qualified member name that includes the dimension, hierarchy, and level under which the specific calculated member needs to be created. The MDX_Expression should return a value that calculates the value of the member. The SOLVE_ORDER, which is an optional parameter, should be a positive integer value if specified. It determines the order in which the members are evaluated when multiple calculated members are defined. The CellProperty is also optional and is used to specify cell properties for the calculated member such as the text formatting of the cell contents including the background color.
All the measures in a cube are stored in a special dimension called Measures. Calculated members can also be created on the measures dimension. In fact, most of the calculated members that are used for business are created on the measures dimension. Calculated members on the measures dimension are referred to as calculated measures. The following are examples of calculated member statements:
Example 1 :

```
WITH MEMBER MEASURES.[Profit] AS [Measures].[Internet Sales Amount]-
[Measures].[Internet Standard Product Cost]
SELECT measures.profit ON COLUMNS,
[Customer].[Country].MEMBERS ON ROWS
FROM [Adventure Works]
```

In Example 1 a calculated member, Profit, has been defined as the difference of the measures [Internet Sales Amount] and [Internet Standard Product Cost]. When the query is executed, the Profit value will be calculated for every country based on the MDX expression.

Example 2 :

```
WITH
SET [Product Order] AS ‘Order([Product].[Product Line].MEMBERS,
[Internet Sales Amount], BDESC)’
MEMBER [Measures].[Product Rank] AS ‘Rank([Product].[Product Line].CURRENTMEMBER,
[Product Order])’
SELECT {[Product Rank], [Sales Amount]} ON COLUMNS,
[Product Order] ON ROWS
from [Adventure Works]
```

Example 2 includes creation of a named set and a calculated member within the scope of the query. The query orders the Products based on the Internet Sales Amount and returns the sales amount of each product along with the rank. The named set [Product Order] is created so that the members within this set are ordered based on the Sales. This is done by using an MDX function called Order (you can learn more about Order in Appendix A, available online on this book’s page at www.wrox.com). To retrieve the rank of each product, a calculated member, [Product Rank], is created using the MDX function Rank .
The result of the preceding query on the Adventure Works cube from the Adventure Works DW 2008 sample database is:

Product	**Rank**	**Sales Amount**
All Products	1	$109,809,274.20
Road	2	$48,262,055.15
Mountain	3	$42,456,731.56
Touring	4	$16,010,837.10
Accessory	5	$2,539,401.59
Components	6	$540,248.80

Example 3 :

```
WITH MEMBER Measures.[Cumulative Sales] AS ‘Sum(
YTD(),[Internet Sales Amount])’
SELECT {Measures.[Internet Sales Amount],Measures.[Cumulative Sales]} ON 0,
[Date].[Calendar].[Calendar Semester].MEMBERS ON 1
FROM [Adventure Works]
```

In Example 3 a calculated member is created so that you can analyze the [Internet Sales Amount] of each half year along with the cumulative sales for the whole year. For this, two MDX functions are used: Sum and YTD . The YTD MDX function is called without any parameters so that the default Time member at that level is used in the calculation. The Sum function is used to aggregate the sales amount for that specific level. The result of the preceding query on the sample Analysis Services database is shown in the following table. You can see that the Cumulative

Sales corresponding for the members H2 CY 2002, H2 CY 2003, and H2 CY 2004 show the sum of Internet Sales Amount for that member and the previous half year.

Example 4 :

```
WITH MEMBER [Date].[Calendar].[%Change] AS
100* (([Date].[Calendar].[Calendar Quarter].[Q2 CY 2002] -
[Date].[Calendar].[Calendar Quarter].[Q1 CY 2002])/
[Date].[Calendar].[Calendar Quarter].[Q2 CY 2002])
SELECT {[Date].[Calendar].[Calendar Quarter].[Q1 CY 2002],
[Date].[Calendar].[Calendar Quarter].[Q2 CY 2002],
[Date].[Calendar].[%Change]} ON COLUMNS,
Measures.[Internet Sales Amount] ON ROWS
FROM [Adventure Works]
```

This query shows an example of a calculated member defined in the Date dimension to return a quarter - over - quarter comparison of the sales amount. In this example, quarter 1 and quarter 2 of the year 2002 are used. The result of this query is:

Q1 CY 2002 Q2 CY 2002 %Change

Internet Sales Amount $1,791,698.45 $2,014,012.13 11.0383486486941

MDX Expressions

MDX expressions are partial MDX statements that evaluate to a value. They are typically used in calculations or in defining values for objects such as default members and default measures, or for defining security expressions to allow or deny access. MDX expressions typically take a member, a tuple, or a set as a parameter and return a value. If the result of the MDX expression evaluation is no value, a Null value is returned. Following are some examples of MDX expressions:

Example 1

```
Customer.[Customer Geography].DEFAULTMEMBER
```

This example returns the default member specified for the Customer Geography hierarchy of the Customer dimension.

Example 2

```
(Customer.[Customer Geography].CURRENTMEMBER, Measures.[Sales Amount]) -
(Customer.[Customer Geography].Australia, Measures.[Sales Amount)
```

This MDX expression is used to compare the sales to customers of different countries with sales of customers in Australia.

Such an expression is typically used in a calculated measure. Complex MDX expressions can include various operators in the MDX language along with the combination of the functions available in MDX. One such example is shown in Example 3.

Example 3

```
COUNT(INTERSECT( DESCENDANTS( IIF( HIERARCHIZE(EXISTS[Employee].
[Employee].MEMBERS,
STRTOMEMBER("[Employee].[login].[login]. & ["+USERNAME+"]")),
POST).ITEM(0).ITEM(0).PARENT.DATAMEMBER is
HIERARCHIZE(EXISTS([Employee].[Employee].MEMBERS,
STRTOMEMBER("[Employee].[login].[login]. & ["+USERNAME+"]")),
POST).ITEM(0).ITEM(0),
HIERARCHIZE(EXISTS([Employee].[Employee].MEMBERS,
STRTOMEMBER("[Employee].[login].[login]. & ["+username+"]")),
POST).ITEM(0).ITEM(0).PARENT,
HIERARCHIZE(EXISTS([Employee].[Employee].MEMBERS,
STRTOMEMBER("[Employee].[login].[login]. & ["+USERNAME+"]")),
POST).ITEM(0).ITEM(0))
).ITEM(0) , Employee.Employee.CURRENTMEMBER)) > 0
```

This example is an MDX cell security expression used to allow employees to see Sales information made by them or by the employees reporting to them and not other employees. This MDX expression uses several MDX functions (you learn some of these in the next section). You can see that this is not a simple MDX expression. The preceding MDX expression returns a value " True " or " False " based on the employee that is logged in. Analysis Services allows appropriate cells to be accessed by the employee based on the evaluation. This example is analyzed in more detail in Chapter 22 .

MDX has progressed extensively since its birth and you can pretty quickly end up with a complex MDX query or MDX expression like the one shown in Example 3. There can be multiple people working on implementing a solution and hence it is good to have some kind of documentation for your queries or expressions. Similar to other programming languages, MDX supports commenting within queries and

MDX expressions. At this time there are three different ways to comment your MDX. They are:
// (two forward slashes) comment goes here
-- (two hyphens) comment goes here
/* comment goes here */ (slash-asterisk pairs)
We highly recommend that you add comments to your MDX expressions and queries so that you can look back at a later point in time and interpret or understand what you were implementing with a specific MDX expression or query.

Operators

The MDX language, similar to other query languages such as SQL or other general - purpose programming languages, has several operators. An operator is a function that is used to perform a specific action, takes arguments, and returns a result. MDX has several types of operators including arithmetic operators, logical operators, and special MDX operators.

Arithmetic Operators

Regular arithmetic operators such as +, – , *, and / are available in MDX. Just as with other programming languages, these operators can be applied on two numbers. The + and – operators can also be used as unary operators on numbers. Unary operators, as the name indicates, are used with a single operand (single number) in MDX expressions such as + 100 or – 100.

Set Operators

The +, – , and * operators, in addition to being arithmetic operators, are also used to perform operations on the MDX sets. The + operator returns the union of two sets, the – operator returns the difference of two sets, and the * operator returns the cross product of two sets. The cross product of two sets results in all possible combinations of the tuples in each set and helps in retrieving data in a matrix format. For example, if you have two sets, {Male, Female} and {2003, 2004, 2005}, the cross product, represented as {Male, Female} * {2003, 2004, 2005}, is {(Male,2003), (Male,2004), (Male,2005),(Female,2003),(Female,2004), (Female,2005)}. The following examples show MDX expressions that use the set operators:
Example 1: The result of the MDX expression
{[Customer].[Country].[Australia]} + {[Customer].[Country].[Canada]}
is the union of the two sets as shown here:
{[Customer].[Country].[Australia], [Customer].[Country].[Canada]}
Example 2: The result of the MDX expression
{[Customer].[Country].[Australia],[Customer].[Country].[Canada]}*
{[Product].[Product Line].[Mountain],[Product].[Product Line].[Road]}
is the cross product of the sets as shown here:
{([Customer].[Country].[Australia],[Product].[Product Line].[Mountain])
([Customer].[Country].[Australia],[Product].[Product Line].[Road])
([Customer].[Country].[Canada],[Product].[Product Line].[Mountain])
([Customer].[Country].[Canada],[Product].[Product Line].[Road])}

Comparison Operators

MDX supports the comparison operators < , < =, > , > =, =, and < > . These operators take two MDX expressions as arguments and return TRUE or FALSE based on the result of comparing the values of each expression.
Example:
The following MDX expression uses the greater than comparison operator, > :
Count (Customer.[Country].members) > 3

In this example Count is an MDX function that is used to count the number of members in Country hierarchy of the Customer dimension. Because there are more than three members, the result of the MDX expression is TRUE.

Logical Operators

The logical operators that are part of MDX are AND, OR, XOR, NOT, and IS, which are used for logical conjunction, logical disjunction, logical exclusion, logical negation, and comparison, respectively. These operators take two MDX expressions as arguments and return TRUE or FALSE based on the logical operation. Logical operators are typically used in MDX expressions for cell and dimension security, which you learn about in Chapter 22 .

Special MDX Operators — Curly Braces, Commas, and Colons

The curly braces, represented by the characters { and }, are used to enclose a tuple or a set of tuples to form an MDX set. Whenever you have a set with a single tuple, the curly brace is optional because Analysis Services implicitly converts a single tuple to a set when needed. When there is more than one tuple to be represented as a set or when there is an empty set, you need to use the curly braces.
You have already seen the comma character used in several earlier examples. The comma character is used to form a tuple that contains more than one member. By doing this you are creating a slice of data on the cube. In addition, the comma character is used to separate multiple tuples specified to define a set. In the set {(Male,2003), (Male,2004), (Male,2005),(Female,2003),(Female,2004),(Female,2005)} the comma character is not only used to form tuples but also to form the set of tuples.
The colon character is used to define a range of members within a set. It is used between two non - consecutive members in a set to indicate inclusion of all the members between them, based on the set ordering (key - based or name - based). For example, if you have the following set:
{[Customer].[Country].[Australia], [Customer].[Country].[Canada],
[Customer].[Country].[France], [Customer].[Country].[Germany],
[Customer].[Country].[United Kingdom], [Customer].[Country].[United States]}
the following MDX expression
{[Customer].[Country].[Canada] : [Customer].[Country].[United Kingdom]}
results in the following set:
{[Customer].[Country].[Canada], [Customer].[Country].[France],
[Customer].[Country].[Germany], [Customer].[Country].[United Kingdom]}

MDX Functions

MDX functions can be used in MDX expressions or in MDX queries. MDX forms the bedrock of Analysis Services 2008. BIDS builds MDX expressions that typically include MDX functions to retrieve data from the Analysis Services database based upon your actions like browsing dimensions or cubes. MDX functions help address some of the common operations that are needed in your MDX expressions or queries including ordering tuples in a set, counting the number of members in a dimension, and string manipulation required to transform user input into corresponding MDX objects.
This section splits the MDX functions into various categories and provides some basic examples. The best way to learn MDX functions is to understand their use in business scenarios so that you can apply the right MDX function in appropriate situations. In this book, you will often see the MDX that the

product generates. Paying attention to and experimenting with such MDX is critical to your transition from basic understanding of Analysis Services 2008 to complete mastery — and, though it is a profound challenge, mastery is attainable. You can do it. Again, when you slice a dimension in any cube - viewing software, like Office Web Components, it is MDX that is generated and executed to retrieve the results. Also, when you create a report based on a cube (UDM) using Excel (as you see in Chapter 17) or using Reporting Services (Chapter 20), it is MDX that is created behind the scenes to capture the contents with which to populate the report. Almost all these MDX queries or expressions generated by BIDS or by client tools use MDX functions; some of which you learn about in detail as you work through this book.
In Chapter 11 you learn about the stored procedure support in Analysis Services 2008 and how you can write your custom functions that can be called within your MDX expressions or queries. For example, the following MDX query contains a custom function MyStoredProc that takes two arguments and returns an MDX object:
SELECT MyStoredProc (arg1, arg2) ON COLUMNS FROM CorporateCube
What we expect will get you even more excited about Chapter 11 is that the .NET assemblies that implement stored procedures can themselves contain MDX expressions within them due to an object model that exposes MDX objects! It should be obvious if you are experienced with Analysis Services that the new version opens up whole new approaches to problem solving in the Business Intelligence space.
Because MDX functions are so central to successful use of Analysis Services 2008, it is best if you jump right in and learn some of them now. Putting those functions together to accomplish more meaningful tasks will come later in the book. For now, please snap on your seatbelt; it ’ s time to learn about MDX functions.

MDX Function Categories

MDX functions are used to programmatically operate on multidimensional databases. From traversing dimension hierarchies to calculating numeric functions over fact data, there is plenty of surface area to explore. In this section, the MDX functions have been categorized in a specific way to help you understand them efficiently. You also see some details on select functions of interest, where interest level is defined by the probability you will use a given function in your future BI development work. You can see all of the MDX functions in detail in Appendix A (available online at www.wrox.com) . We have categorized the MDX functions into several categories very similar to the product documentations of MDX functions. MDX functions can be called in several ways:

.Function (read *dot* function)
Example: Dimension.Name returns the name of the object being referenced (could be a hierarchy or level/member expression). Perhaps this reminds you of the dot operator in VB.NET or C# programming — that ' s fine. It ' s roughly the same idea.

```
WITH MEMBER measures.LocationName AS [Customer].[Country].CurrentMember.Name
SELECT measures.LocationName ON COLUMNS,
Customer.Country.members on ROWS
FROM [Adventure Works]
```

Function
Example: Username is used to acquire the username of the logged - in user. It returns a string in the following format: domain - name\user - name. Most often this is used in dimension or cell security related MDX expressions. The following is an example of how username can be used in an MDX expression:

```
WITH MEMBER Measures.User AS USERNAME
SELECT Measures.User ON 0 FROM [Adventure Works]
```

Function ( )
Example: The function CalculationCurrentPass ( ) requires parentheses, but takes no arguments. You can find more on CalculationCurrentPass ( ) in Appendix A (available online at www.wrox.com).
Function (arguments)
Example: OpeningPeriod ([Level_Expression [, Member_Expression]]) is an MDX function that takes an argument that can specify both level_expression with member_expression or just the member_expression itself. This function is most often used with Time dimensions, but will work with other dimension types. It returns the first member at the level of the member_ expression. For example, the following returns the first member of the Day level of the April member of the default time dimension:

```
OpeningPeriod (Day, [April])
```

Set Functions

Set functions, as the category name suggests, operate on sets. They take sets as arguments and often return a set. Some of the widely used set functions are Crossjoin and Filter , which we are quite sure you will be using in your MDX queries. Hence these two functions are discussed here with examples. Crossjoin returns all possible combinations of sets as specified by the arguments to the Crossjoin function. If there are N sets specified in the Crossjoin function, this will result in a combination of all the possible members within that set on a single axis. You see this in the following example:

```
Crossjoin ( Set_Expression [ ,Set_Expression ...] )
SELECT Measures.[Internet Sales Amount] ON COLUMNS,
CROSSJOIN( {Product.[Product Line].[Product Line].MEMBERS},
{[Customer].[Country].MEMBERS}) on ROWS
FROM [Adventure Works]
```

This query produces the cross product of each member in the Product dimension with each member of the Customer dimension along the sales amount measure. The following are the first few rows of results from executing this query:

Sales Amount
Accessory All Customers $604,053.30
Accessory Australia $127,128.61
Accessory Canada $82,736.07
Accessory France $55,001.21
Accessory Germany $54,382.29
Accessory United Kingdom $67,636.33
Accessory United States $217,168.79
Components All Customers (null)

Sometimes the result of the combination of the members of the set results in values being null. For example, assume that there is one product that is sold only in Australia. The sales amount for this product in other countries is going to be Null. Obviously you are not interested in the empty results. It does not help in any business decision. Instead of retrieving all the results and then checking for null values, there is a way to restrict these on the server side of Analysis Services. In addition to this, Analysis Services optimizes the query so that only the appropriate result is retrieved and sent. For this, you use the NonEmptyCrossjoin function or the NonEmpty function. The syntax for these two functions are :

```
NonEmptyCrossjoin(
Set_Expression [ , Set_Expression ...][ , Crossjoin_Set_Count ] )
```

NonEmpty(*Set_Expression* [, *FilterSet_Expression*])
To remove empty cells in these query results using Crossjoin you can use one of the following queries, which use the NonEmptyCrossjoin and NonEmpty functions. When using the NonEmptyCrossjoin function, you need to apply the filter condition on [Internet Sales Amount] and then retrieve the crossjoin of members from the first two sets. This is due to the fact that the default measure for the Adventure Works cube is not [Internet Sales Amount] and hence, if the measure is not included as a parameter in the function, NonEmptyCrossjoin will use the default measure. When using the NonEmpty function, you first do the crossjoin and then filter out the tuples that have null values for the Internet Sales amount as shown in the second query in the following code. The NonEmpty MDX function was first introduced in Analysis Services 2005.

```
SELECT Measures.[Internet Sales Amount] ON COLUMNS,
NONEMPTYCROSSJOIN( {Product.[Product Line].[Product Line].MEMBERS},
{[Customer].[Country].MEMBERS},Measures.[Internet Sales Amount],2 ) ON ROWS
FROM [Adventure Works]
SELECT Measures.[Internet Sales Amount] ON COLUMNS,
NONEMPTY(CROSSJOIN ( {Product.[Product Line].[Product Line].MEMBERS},
{[Customer].[Country].MEMBERS}),Measures.[Internet Sales Amount]) ON ROWS
FROM [Adventure Works]
```

Most users and client tools interacting with Analysis Services use the NonEmptyCrossjoin function extensively. You see more examples of this function in later chapters of this book.
Another MDX function that is quite useful is the Filter function. The Filter function helps restrict the query results based on one or more conditions. The Filter function takes two arguments: a set expression and a logical expression. The logical expression is applied on each item of the set and returns a set of items that satisfy the logical condition. The function arguments for the Filter function are:

```
Filter( Set_Expression , { Logical_Expression | [ CAPTION | KEY | NAME ]
=String_Expression } )
```

The result of the example query shown for the Crossjoin function results in 35 cells. If you are only interested in the products for which the sales amount is greater than a specific value and are still interested in finding out amounts by countries, you can use the Filter function as shown here:

```
SELECT Measures.[Internet Sales Amount] ON COLUMNS,
FILTER(CROSSJOIN( {Product.[Product Line].[Product Line].MEMBERS},
{[Customer].[Country].MEMBERS}),[Internet Sales Amount] > 2000000) on ROWS
FROM [Adventure Works]
```

This query filters out all the products for which the sales amount is less than 2,000,000 and returns only the products that have the sales amount greater than 2,000,000. The result of execution of this query is as follows:

Sales Amount
Mountain All Customers $10,251,183.52
Mountain Australia $2,906,994.45
Mountain United States $3,547,956.78
Road All Customers $14,624,108.58
Road Australia $5,029,120.41
Road United States $4,322,438.41
Touring All Customers $3,879,331.82

Member Functions

Member functions are used for operations on the members such as retrieving the current member, ancestor, parent, children, sibling, next member, and so on. All the member functions return a member. One of the most widely used member functions is called ParallelPeriod . The ParallelPeriod function helps you to retrieve a member in the Time dimension based on a given member and certain conditions. The function definition for ParallelPeriod is :

```
ParallelPeriod( [ Level_Expression [ ,Numeric_Expression [ , Member_Expression ] ] ] )
```

Figure 3 - 5 shows an illustration of ParallelPeriod function. ParallelPeriod is a function that returns a member from a Time dimension (you learn about time dimensions in Chapter 5) relative to a given member for a specific time period. For example, ParallelPeriod([Quarter], 1, [April]) is [January]. You might be wondering how this result came about. The following steps describe the execution of the ParallelPeriod function and how Analysis Services arrives at the result:

1. The ParallelPeriod function can only be used in conjunction with time dimensions. For the illustration shown in Figure 3 - 5 , assume you have a time dimension with a Calendar hierarchy that contains the levels Year, Semester, Quarter, and Month.
2. The ParallelPeriod function first finds the ancestor member of last argument, April, in the specified level, Quarter, which is the first argument. It identifies that the ancestor of April at the

specified level is Quarter2.

3. The sibling of [Quarter2] is then evaluated based on the numeric expression. A positive number indicates that the sibling of interest exists as a predecessor to the current member in the collection of members at that level. A negative number indicates that the sibling of interest is a successor of the current member. In this example, the sibling of interest is [Quarter1] because the numeric expression is 1.

4. Next, the member at the same position as that of member [April] is identified in [Quarter1], which is January.

The ParallelPeriod function is used to compare measure values relative to various time periods. Typically a customer would be interested in comparing Sales between Quarters or over Years, and this function really comes in handy when you want to make relative comparisons. Most of the client tools interacting with Analysis Services use this function.

Numeric Functions

Numeric functions come in very handy when you are defining the parameters for an MDX query or creating any calculated measure. Note that there are plenty of statistical functions in this group, including standard deviation, sample variance, and correlation. The most common of the numeric functions is a simple one called Count along with its close cousin, DistinctCount . The Count function is used to count the number of items in the collection of a specific object like a Dimension, a Tuple, a Set, or a Level. The DistinctCount function, on the other hand, takes a Set_Expression as an argument and returns a number that indicates the number of distinct items in the Set_Expression, not the total count of all items. Here are the function definitions for each:

```
Count ( Dimension | Tuples | Set| Level)
DistinctCount ( Set_Expression )
```

Please take a look at the following query:

```
WITH MEMBER Measures.CustomerCount AS DistinctCount(
Exists([Customer].[Customer].MEMBERS,[Product].[Product Line].Mountain,
“Internet Sales”))
SELECT Measures.CustomerCount ON COLUMNS
FROM [Adventure Works]
```

The DistinctCount function counts the number of distinct members in the Customer dimension who have purchased products in the Mountain product line. If a customer has purchased multiple products from the specified product line, the DistinctCount function will count the customer just once. The MDX function Exists is used to filter customers who have only purchased product line Mountain through the Internet. You learn more about the Exists function in Chapter 10 . The result of the Exists function is the set of Internet customers who have purchased products from the Mountain product line. The result of the preceding query is 9590.

Dimension Functions, Level Functions, and Hierarchy Functions

Functions in these groups are typically used for navigation and manipulation. Here is an example of just such a function, the “ Level ” function from the Level group:

```
SELECT [Date].[Calendar].[Calendar Quarter].[Q1 CY 2004].LEVEL ON COLUMNS
FROM [Adventure Works]
```

This query results in a list of all the quarters displayed in the results. The reason is because [Date] .[Calendar].[Calendar Quarter].[Q1 CY 2004].LEVEL evaluates to [Date].[Calendar Year].[Calendar Semster].[Calender Quarter]. From this, you get the list of all quarters for all calendar years.

String Manipulation Functions

To extract the names of sets, tuples, and members in the form of a string, you can use functions like MemberToStr (< Member_Expression >) and to do the inverse, take a string and create a member expression, you can use StrToMember (< String >). Consider the following case, in which there is a client application that displays sales information for all countries. When a user selects a specific country, you need to extract the sales information for the specific country from Analysis Services. Because the countries are represented as strings in the client application, you need to translate this string to a corresponding member, and then you can retrieve the data. String manipulation functions are useful when accepting parameters from users and transforming them to corresponding MDX objects. However there is a significant performance cost involved when using string manipulation functions. Hence we recommend you use these functions only if necessary.

```
SELECT STRTOMEMBER (‘[Customer].[Country].[Australia]’ ) ON COLUMNS
```

```
FROM [Adventure Works]
```

Other Functions

Four other function categories exist: Subcube and Array both have one function each. The final two categories are logical functions, which allow you to do Boolean evaluations on multidimensional objects, and tuple functions that you can use to access tuples. In addition, Analysis Services 2005 and 2008 have introduced a few new MDX functions. You have seen some of them in this chapter such as NonEmpty and Exists . You learn more about these in Chapter 10 and Appendix A (available online at www.wrox.com).

Summary

Congratulations, you have made it through the first three chapters! Ostensibly, you should now feel free to take on the rest of the chapters in no particular order. But you got this far, so why not go immediately to Chapter 4 and jump right in? Now you know the fundamental elements of MDX — cells, members, tuples, and sets. Further, you learned that MDX has two forms: queries and expressions.

You saw that MDX queries, which are used to retrieve data from Analysis Services databases, retain a superficial resemblance to SQL, but that the resemblance breaks down the more you drill down on the details. MDX expressions, on the other hand, are simple yet powerful constructs that are partial statements — by themselves they do not return results like queries. The expressions are what enable you to define and manipulate multidimensional objects and data through calculations, like specifying a default member ' s contents, for example.

To solidify your basic understanding of MDX, you learned the common query statements, WITH , SELECT , FROM , and WHERE , as well as the MDX operators like addition, subtraction, multiplication, division, and rollup, and the logical operators AND and OR. These details are crucial to effective use of the language. You got a good look at the eleven MDX function categories, saw the four forms MDX functions can take, and even saw detailed examples of some commonly used functions like Filter , ParallelPeriod , MemberToStr , and StrToMember . You learn more advanced MDX concepts and functions in Chapters 8 , 9 , 10 , 11 , and 12 . All the MDX functions supported in Analysis Services 2008 are provided with examples in Appendix A, available online at www.wrox.com. Coming up next in Chapter 4 are the details of creating a data source, a Data Source View, and how to deal with multiple data source views in a single project.