# MDX

The Multidimensional Expressions (MDX) language is used to manipulate multidimensional information in Microsoft® SQL Server™ 2000 Analysis Services. MDX is defined in the OLAP extensions in OLE DB.

Similar to SQL in many respects, MDX provides a rich and powerful syntax for the retrieval and manipulation of multidimensional data, such as the data stored in cubes on the Analysis server. Analysis Services supports MDX functions in the definitions of calculated members, as well as a full language implementation for building local cubes and querying cube data using PivotTable® Service with OLE DB and Microsoft ActiveX® Data Objects (ADO).

Additionally, MDX supports the creation and registration of user-defined functions. You can create user-defined functions to operate on multidimensional data and accept arguments and return values in the MDX syntax.

The following topics provide more information about MDX.

| Topic | Description |
|---|---|
| MDX Overview | Describes basic MDX concepts and provides a comparison between SQL syntax and MDX syntax. |
| Basic MDX | Gives a basic overview of the construction of a simple MDX query. |
| Advanced MDX | Details more advanced information, such as named sets and calculated members, for complex MDX queries. |
| Effective MDX | Provides a list of tips, workarounds, and feature discussions regarding MDX. |
| MDX Functions in Analysis Services | Details the statements and functions supported by MDX. |

# MDX Overview

This section introduces Multidimensional Expressions (MDX) and explains some of the concepts behind its structure and syntax. It contains the following topics.

# Introduction to MDX

MDX, an acronym for **M**ulti**d**imensional E**x**pressions, is a syntax that supports the definition and manipulation of multidimensional objects and data. MDX is similar in many ways to the Structured Query Language (SQL) syntax, but is not an extension of the SQL language; in fact, some of the functionality that is supplied by MDX can be supplied, although not as efficiently or intuitively, by SQL.

As with an SQL query, each MDX query requires a data request (the SELECT clause), a starting point (the FROM clause), and a filter (the WHERE clause). These and other keywords provide the tools used to extract specific portions of data from a cube for analysis. MDX also supplies a robust set of functions for the manipulation of retrieved data, as well as the ability to extend MDX with user-defined functions.

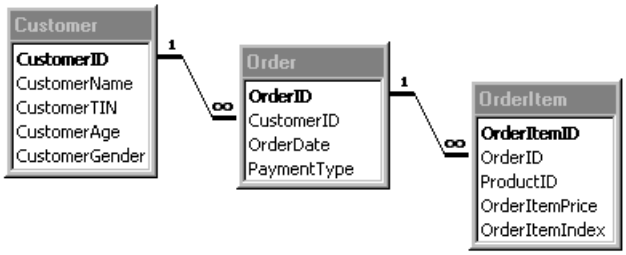
MDX, like SQL, provides data definition language (DDL) syntax for managing data structures. There are MDX commands for creating (and deleting) cubes, dimensions, measures, and their subordinate objects.

# Key Concepts in MDX

The purpose of Multidimensional Expressions (MDX) is to make accessing data from multiple dimensions easier and more intuitive.

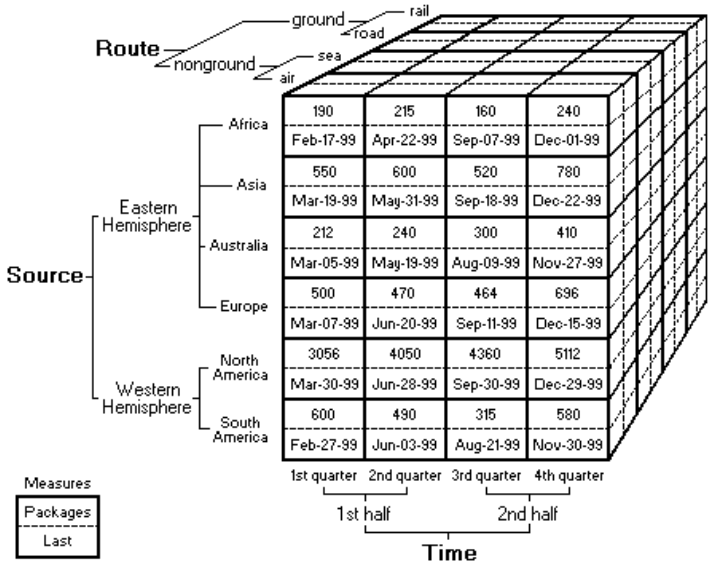
# Dimensions, Levels, Members, and Measures

Most languages used for data definition and manipulation, such as SQL, are designed to retrieve data in two dimensions: a column dimension and a row dimension. The following diagram illustrates a traditional relational database, used to store order information.




Each table represents two-dimensional data. At the intersection of each row and column is a single element of data, called a field. The specific columns to be viewed in an SQL query are specified with a SELECT statement, and the rows to be retrieved are limited by a WHERE clause.

Multidimensional data, on the other hand, can be represented by structures with more than two dimensions. These structures, called cubes, have multiple dimensions. At the intersection of dimensions in a cube, there may be more than one element of data, called a measure. The following diagram illustrates a cube that employs three dimensions, Route, Service and Time; and two measures, Packages and Last. Each dimension is broken down into different levels, each of which is broken down further into members. For example, the Source dimension supplies the Eastern Hemisphere level, which is broken down into four members, Africa, Asia, Australia, and Europe.




As you can see, the querying of even simple data out of a multidimensional data source can be a complex task. A cube can have more than three dimensions, for example, or it may only have one dimension.

The concepts of cubes, dimensions, levels, members, and measures are important to the understanding of MDX syntax. Further reading on these architectural topics is recommended if you are new to online analytical processing (OLAP) databases.

# Cells, Tuples, and Sets

As SQL returns a subset of two-dimensional data from tables, MDX returns a subset of multidimensional data from cubes.

The cube diagram illustrates that the intersection of multidimensional members creates cells from which you can obtain data. To identify and extract such data, whether it be a single cell or a block of cells, MDX uses a reference system called tuples. Tuples list dimensions and members to identify individual cells as well as larger sections of cells in the cube; because each cell is an intersection of all the dimensions of the cube, tuples can uniquely identify every cell in the cube. For the purposes of reference, measures in a cube are treated as a private dimension, named Measures, in the cube itself. For example, in the preceding diagram, the following tuple identifies a cell in which the value is 240:

```
(Source.[Eastern Hemisphere].Africa, Time.[2nd half].[4th quarter], Route.Air,
Measures.Packages)
```

The tuple uniquely identifies a section in the cube; it does not have to refer to a specific cell, nor does it have to encompass all of the dimensions in a cube. The following examples are all tuples of the cube diagram:

```
(Source.[Eastern Hemisphere])
(Time.[2nd half], Source.[Western Hemisphere])
```

These tuples provide sections of the cube, called slices, that encompass more than one cell.

An ordered collection of tuples is referred to as a set. In an MDX query, axis and slicer dimensions are composed of such sets of tuples. The following example is a description of a set of tuples in the cube in the diagram:

```
{ (Time.[1st half].[1st quarter]), Time.[2nd half].[3rd quarter]) }
```

In addition, it is possible to create a named set. A named set is a set with an alias, used to make your MDX query easier to understand and, if it is particularly complex, easier to process.

# Axis and Slicer Dimensions

In SQL, it is usually necessary to restrict the amount of data returned from a query on a table. For example, you may want to see only two fields of a table with forty fields, and you want to see them only if a third field meets a specific criteria. You can accomplish this by specifying columns in the SELECT statement, using a WHERE statement to restrict the rows that are returned based on specific criteria.

In MDX, those concepts also apply. A SELECT statement is used to select the dimensions and members to be returned, referred to as axis dimensions. The WHERE statement is used to restrict the returned data to specific dimension and member criteria, referred to as a slicer dimension. An axis dimension is expected to return data for multiple members, while a slicer dimension is expected to return data for a single member.

The terms "axis dimension" and "slicer dimension" are used to differentiate the dimensions of the cells in the source cube of the query, indicated in the FROM clause, from the dimensions of the cells in the result cube, which can be composed of multiple cube dimensions.

# Calculated Members

Calculated members are members that are based not on data, but on evaluated expressions in MDX. They are returned in the same fashion as a normal member. MDX supplies a robust set of functions that can be used to create calculated members, giving you extensive flexibility in the manipulation of multidimensional data.

## User-Defined Functions

MDX provides extensibility in the form of user-defined functions using any programming language that can support Component Object Model (COM) interfaces. You create and register your own functions that operate on multidimensional data as well as accept arguments and return values in the MDX syntax. You can call user-defined functions from within Calculated Member Builder, data definition language (DDL) statements that support MDX, and MDX queries.

## PivotTable Service

In Microsoft® SQL Server™ 2000 Analysis Services, MDX data definition and manipulation services are provided through PivotTable® Service. PivotTable Service also provides stand-alone OLE DB provider capabilities for multidimensional queries when not connected to an Analysis server. PivotTable Service is used for the definition and manipulation of local cubes, which can be used to locally store data in a multidimensional format.

# Comparison of SQL and MDX

The Multidimensional Expressions (MDX) syntax appears, at first glance, to be remarkably similar to the syntax of Structured Query Language (SQL). In many ways, the functionality supplied by MDX is also similar to that of SQL; with effort, you can even duplicate some of the functionality provided by MDX in SQL.

However, there are some striking differences between SQL and MDX, and you should be aware of these differences at a conceptual level. The following information is intended to provide a guide to these conceptual differences between SQL and MDX, from the point of view of an SQL developer.

The principal difference between SQL and MDX is the ability of MDX to reference multiple dimensions. Although it is possible to use SQL exclusively to query cubes in Microsoft® SQL Server™ 2000 Analysis Services, MDX provides commands that are designed specifically to retrieve data as multidimensional data structures with almost any number of dimensions.

SQL refers to only two dimensions, columns and rows, when processing queries. Because SQL was designed to handle only two-dimensional tabular data, the terms "column" and "row" have meaning in SQL syntax.

MDX, in comparison, can process one, two, three, or more dimensions in queries. Because multiple dimensions can be used in MDX, each dimension is referred to as an axis. The terms "column" and "row" in MDX are simply used as aliases for the first two axis dimensions in an MDX query; there are other dimensions that are also aliased, but the alias itself holds no real meaning to MDX. MDX supports such aliases for display purposes; many OLAP tools are incapable of displaying a result set with more than two dimensions.

In SQL, the SELECT clause is used to define the column layout for a query, while the WHERE clause is used to define the row layout. However, in MDX the SELECT clause can be used to define several axis dimensions, while the WHERE clause is used to restrict multidimensional data to a specific dimension or member.

In SQL, the WHERE clause is used to filter the data returned by a query. In MDX, the WHERE clause is used to provide a slice of the data returned by a query. While the two concepts are similar, they are not equivalent.

The SQL query uses the WHERE clause to contain an arbitrary list of items that should (or should not) be returned in the result set. While a long list of conditions in the filter can narrow the scope of the data that is retrieved, there is no requirement that the elements in the clause will produce a clear and concise subset of data.

In MDX, however, the concept of a slice means that each member in the WHERE clause identifies a distinct portion of data from a different dimension. Because of the organizational

structure of multidimensional data, it is not possible to request a slice for multiple members of the same dimension. Because of this, the WHERE clause in MDX can provide a clear and concise subset of data.

The process of creating an SQL query is also different than that of creating an MDX query. The creator of an SQL query visualizes and defines the structure of a two-dimensional rowset and writes a query on one or more tables to populate it. In contrast, the creator of an MDX query usually visualizes and defines the structure of a multidimensional dataset and writes a query on a single cube to populate it. This could result in a multidimensional dataset with any number of dimensions; a one-dimensional dataset is possible, for example.

The visualization of an SQL result set is intuitive; the set is a two-dimensional grid of columns and rows. The visualization of an MDX result set is not as intuitive, however. Because a multidimensional result set can have more than three dimensions, it can be challenging to visualize the structure. To refer to such two-dimensional data in SQL, the name of a column and the unique identification of a row, in whatever method is appropriate for the data, are used to refer to a single cell of data, called a field. However, MDX uses a very specific and uniform syntax to refer to cells of data, whether the data forms a single cell or a group of cells.

Although SQL and MDX share similar syntax, the MDX syntax is remarkably robust, and it can be complex. However, because MDX was designed to provide a simple, effective way of querying multidimensional data, it addresses the conceptual differences between two-dimensional and multidimensional querying in a consistent and easily understood fashion.

# Basic MDX

Multidimensional Expressions (MDX) commands allow you to query multidimensional objects, such as cubes, and return multidimensional datasets. This topic and its subtopics provide an overview of MDX queries.

As is the case with SQL, the author of an MDX query must determine the structure of the requested dataset before writing the query. The following topics describe MDX queries and the datasets they produce, and provide more detailed information about basic MDX syntax.

# The Basic MDX Query

A basic Multidimensional Expressions (MDX) query is structured in a fashion similar to the following example:

```
SELECT [<axis_specification>
       [, <axis_specification>...]]
  FROM [<cube_specification>]
[WHERE [<slicer_specification>]]
```

## Basic MDX Syntax - SELECT Statement

In MDX, the SELECT statement is used to specify a dataset containing a subset of multidimensional data. To discuss the various syntax elements of the MDX SELECT statement, this topic presents a basic MDX query example and breaks it down into its syntax elements, discussing the purpose and structure of each element.

To specify a dataset, an MDX query must contain information about:

- The number of axes. You can specify up to 128 axes in an MDX query.
- The members from each dimension to include on each axis of the MDX query.
- The name of the cube that sets the context of the MDX query.
- The members from a slicer dimension on which data is sliced for members from the axis dimensions.

This information can be complex. As you will see in this topic, MDX syntax can provide such information in a simple and straightforward manner, using the MDX SELECT statement.

# Basic MDX Query Example

The following MDX query example is used to discuss the various parts of basic MDX SELECT statement syntax:

```
SELECT
   { [Measures].[Unit Sales], [Measures].[Store Sales] } ON COLUMNS,
   { [Time].[1997], [Time].[1998] } ON ROWS
FROM Sales
WHERE ( [Store].[USA].[CA] )
```

The basic MDX SELECT statement contains a SELECT clause and a FROM clause, with an optional WHERE clause.

The SELECT clause determines the axis dimensions of an MDX SELECT statement. Two axis dimensions are defined in the MDX query example. For more information about the construction of axis dimensions in a SELECT clause, see Specifying the Contents of an Axis Dimension.

The FROM clause determines which multidimensional data source is to be used when extracting data to populate the result set of the MDX SELECT statement. For more information about the FROM clause, see SELECT Statement.

The WHERE clause optionally determines which dimension or member to use as a slicer dimension; this restricts the extracting of data to a specific dimension or member. The MDX query example uses a WHERE clause to restrict the data extract for the axis dimensions to a specific member of the Store dimension. For more information about the construction of a slicer dimension in a WHERE clause, see Specifying the Contents of a Slicer Dimension.

The MDX SELECT statement supports other optional syntax, such as the WITH keyword, and the use of MDX functions to construct members by calculation for inclusion in an axis or slicer dimension. For more information about the MDX SELECT statement, see SELECT Statement.

The syntax format of the MDX SELECT statement is similar to that of SQL syntax; however, you will note several obvious differences:

- MDX syntax distinguishes sets by surrounding tuples or members with braces (the { and } characters.) For more information about member, tuple, and set syntax, see Members, Tuples, and Sets.
- MDX queries can have up to 128 axis dimensions in the SELECT statement, but only the first 5 axes have aliases. An axis can be referred to by its ordinal position within an MDX query or by its alias, if it has an alias assigned to it. In the MDX query example, the COLUMNS and ROWS axis aliases are used. The MDX query could also have been written in the following fashion, using the ordinal position of each axis:

```
SELECT
   { [Measures].[Unit Sales], [Measures].[Store Sales] } ON AXIS(0),
   { [Time].[1997], [Time].[1998] } ON AXIS(1)
FROM Sales
WHERE ( [Store].[USA].[CA] )
```

- As with an SQL query, the FROM clause names the source of the data for the MDX query. However, unlike an SQL query, the FROM clause in an MDX query is restricted to a single cube. Information from other cubes can be retrieved, however, on a value-by-value basis using the **LookupCube** function.
- The WHERE clause is used to describe the slicer dimensions. If a dimension is not mentioned as part of the WHERE clause, Microsoft® SQL Server™ 2000 Analysis Services assumes that any dimension not assigned to an axis dimension is a slicer dimension, and the dimension is filtered on its default members. The WHERE clause

can change the filtering process for specified dimensions, allowing fine control of included data.

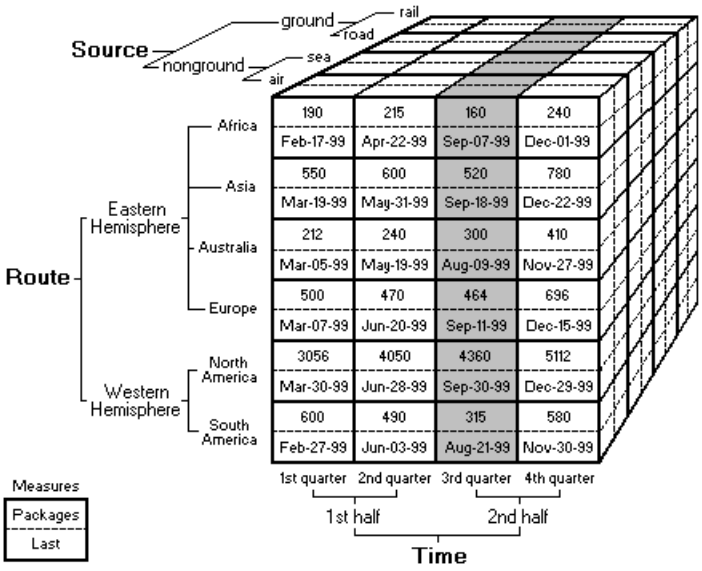
# Members, Tuples, and Sets

Before proceeding on the creation of a Multidimensional Expressions (MDX) query, you should understand the definitions of members, tuples and sets, as well as the MDX syntax used to construct and refer to these elements.

## Members

A member is an item in a dimension representing one or more occurrences of data. Think of a member in a dimension as one or more records in the underlying database whose value in this column falls under this category. A member is the lowest level of reference when describing cell data in a cube.

For example, the following diagram is shaded to represent the `Time.[2nd half].[3rd quarter]` member.




The bracket characters, [ and ], are used if the name of a member has a space or a number in it. Although the Time dimension is one word, bracket characters can also be used around it as well; the member shown in the previous diagram could also be represented as:

```
[Time].[2nd half].[4th quarter]
```

The right bracket (**]**) can be used as an escape character in MDX if the member name or member key contains a right bracket, as shown in the following example:

```
[Premier [150]] 98]
```

### Member Names and Member Keys

A member can be referenced by either its member name or by its member key. The previous example referenced the member by its member name, 4th quarter, in the Time dimension. However, the member name can be duplicated in the case of dimensions with nonunique member names, or it can be changed in the case of changing dimensions.

An alternate method to reference members is by referencing the member key. The member key is used by the dimension to specifically identify a given member. The ampersand (**&**) character is used in MDX to differentiate a member key from a member name, as shown in the following example:

```
[Time].[2nd half].&[Q4]
```

In this case, the member key of the 4th quarter member, Q4, is used. Referencing the member key ensures proper member identification in changing dimensions and in dimensions with nonunique member names.

The ampersand character can be used to indicate a member key reference in any MDX expression.

### Calculated Members

Members can also be created, as part of an MDX query, to return data based on evaluated expressions instead of stored data in a cube to be queried. These members are called calculated members, and they provide a great deal of the power and flexibility of MDX. The WITH keyword is used in an MDX query to define a calculated member. For example, if you want to provide a forecast estimate all of the packages by adding 10% of the existing value of the Packages measure, you can simply create a calculated member that provides the information and use it just like any other member in the cube, as demonstrated in the following example.

```
WITH MEMBER [Measures].[PackagesForecast] AS
'[Measures].[Packages] * 1.1'
```

For more information, see Calculated Members.

### Member Functions

MDX supplies a number of functions for retrieving members from other MDX entities, such as dimensions and levels, so that explicit references to a member are not always necessary. For example, the **FirstChild** function allows the retrieval of all the members from a given dimension or level; to get the first child member of the Time dimension, you can explicitly state it, as demonstrated in the following example:

```
Time.[1st half]
```

You can also use the **FirstChild** function to return the same member, demonstrated in the next example.

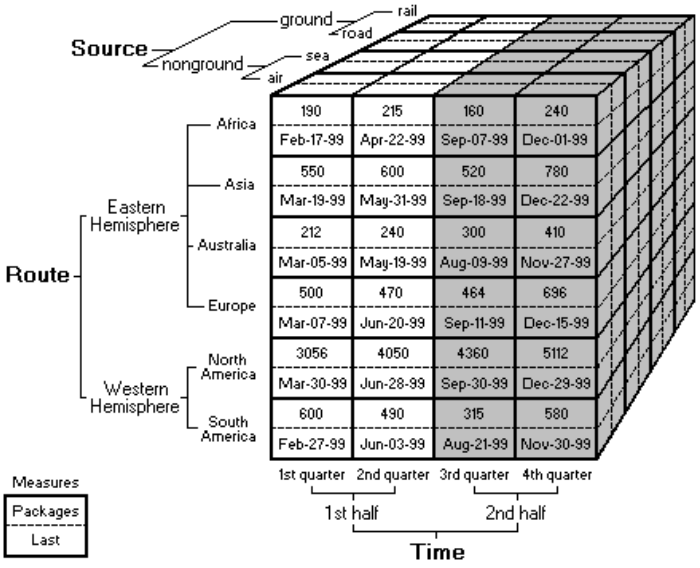
```
Time.FirstChild
```

For more information about MDX member functions, see MDX Function List.

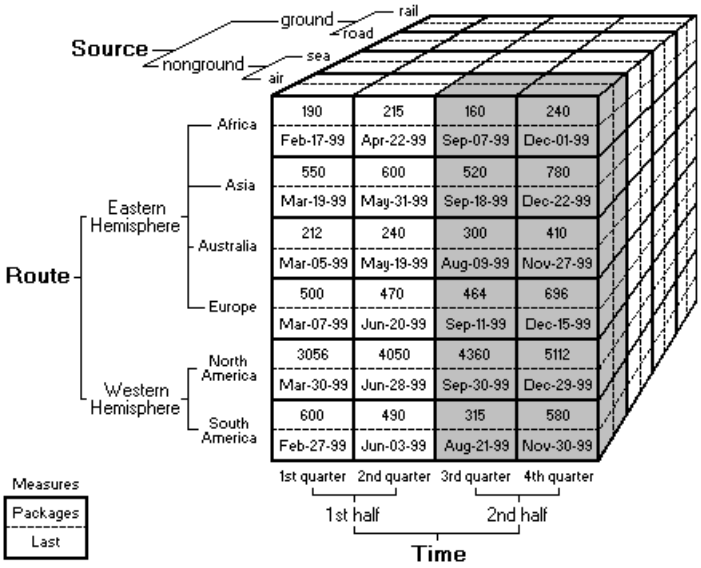
## Tuples

A tuple is used to define a slice of data from a cube; it is composed of an ordered collection of one member from one or more dimensions. A tuple is used to identify specific sections of multidimensional data from a cube; a tuple composed of one member from each dimension in a cube completely describes a cell value. Put another way, a tuple is a vector of members; think of a tuple as one or more records in the underlying database whose value in these columns falls under these categories. A series of diagrams presents different types of tuples.

The shaded area of the cube represents the `(Time.[2nd half])` tuple. Note that this tuple encompasses half of the cube, because it does not rule out any information in the Source or Route dimensions.




The following diagram is shaded to represent the `(Time.[2nd half], Route.nonground.air)` tuple.




This tuple represents the cells at the intersection of these members.

In MDX, tuples are syntactically constructed depending upon their complexity. If a tuple is composed of only one member from a single dimension, often referred to as a simple tuple, the following syntax is acceptable.

```
Time.[2nd half]
```

If a tuple is composed of members from more than one dimension, the members represented by the tuple must be enclosed in parentheses, as demonstrated in the following example.

```
(Time.[2nd half], Route.nonground.air)
```

A tuple composed of a single member can also be enclosed in parentheses, but this is not required. Tuples are often grouped together in sets for use in MDX queries.

### Tuple Functions

There are a few MDX functions that return tuples, and they can be used anywhere that a tuple is accepted.

For more information about tuple functions, see MDX Function List.

### Tuples and Dimensionality

A tuple can encompass members in multiple dimensions, as well as multiple members from the same dimension. The term *dimensionality* is used to indicate the dimensions described by the members in a tuple. Order plays a factor in the dimensionality of a tuple, and can affect the use of a tuple within a set.

## Sets

A set is an ordered collection of zero, one or more tuples. A set is most commonly used to define axis and slicer dimensions in an MDX query, and as such may have only a single tuple or may be, in certain cases, empty. The following example shows a set of two tuples:

```
{ (Time.[1st half], Route.nonground.air), (Time.[2nd half], Route.nonground.sea) }
```

A set can contain more than one occurrence of the same tuple. The following set is acceptable:

```
{ Time.[2nd half], Time.[2nd half] }
```

A set refers to either a set of member combinations, represented as tuples, or to the values in the cells that the tuples in the set represent, depending on the context of usage for the set.

In MDX syntax, tuples are enclosed in braces to construct a set.

> **IMPORTANT** Sets composed of a single tuple are not tuples; they are interpreted as sets by MDX. Certain MDX functions accept tuples as parameters, and will raise an error if a single tuple set is passed. Tuples and single-tuple sets are not interchangeable.

### Set Functions

Explicitly typing tuples and enclosing them in braces is not the only way to retrieve a set. MDX supports a wide variety of functions that return sets.

The colon operator allows you to use the natural order of members to create a set. For example, the following set:

```
{[1st quarter]:[4th quarter]}
```

retrieves the same set of members as the following set:

```
{[1st quarter], [2nd quarter], [3rd quarter], [4th quarter]}
```

The colon operator is an inclusive function; the members on both sides of the colon operator are included in the resulting set.

Other MDX functions that return sets can be used either by themselves or as part of a comma-delimited list of members. For example, all of the following MDX expressions are valid:

```
{Time.Children}
{Time.Children, Route.nonground.air}
{Time.Children, Route.nonground.air, Source.Children}
```

For more information about set functions, see MDX Function List.

### Sets and Dimensionality

Like tuples, sets also have dimensionality. As a set is composed of tuples, so the dimensionality of a set is expressed by the dimensionality of each tuple within it. Because of this, tuples within a set must have the same dimensionality. In other words, this example would not work as a set:

```
{ (Time.[2nd half], Route.nonground.air), (Route.nonground.air, Time.[2nd half]) }
```

The order of tuples in a set is important; it affects, for example, the nesting order in an axis dimension. The first tuple represents the first, or outermost, dimension, the second tuple represents the next outermost dimension, and so on.

### Named Sets

A named set is a set for which an alias has been created. A named set is most commonly used in complex MDX queries to make these queries easier to read and to increase the ease of maintenance.

For more information about named sets, see Building Named Sets in MDX.

# Axis and Slicer Dimensions

When formulating a Multidimensional Expressions (MDX) query, an application typically looks at the cubes and divides the set of dimensions into two subsets:

- Axis dimensions, for which data is retrieved for multiple members.
- Slicer dimensions, for which data is retrieved for a single member.

Because axis and slicer dimensions can be constructed from multiple dimensions of the cube to be queried, these terms are used to differentiate the dimensions employed by the cube to be queried from the dimensions created in the cube returned by an MDX query.

For example, assume that a cube exists, named TestCube, with two simple dimensions named Route and Time. Because the measures of the cube are part of the Measures dimension, this cube has three dimensions in all. The query is to provide a matrix in which the Packages measure can be compared across routes and times.

In the following MDX query example, the Route and Time dimensions are used as axis dimensions and the Measures dimension is used as the slicer dimension. The Members function indicates that the members of the dimension or level are to be used to construct a set, instead of having to explicitly state each member of a given dimension or level in an MDX query.

```
SELECT
   { Route.nonground.Members } ON COLUMNS,
   { Time.[1st half].Members } ON ROWS
FROM TestCube
WHERE ( [Measures].[Packages] )
```

The resulting grid of values would resemble the following table, showing the value of the Packages measure at each intersection of the COLUMNS and ROWS axis dimensions.

| | air | sea |
|---|---|---|
| 1st quarter | 60 | 50 |
| 2nd quarter | 45 | 45 |

MDX evaluates the axis and slicer dimensions first, building the structure of the result cube before retrieving the information from the cube to be queried.

The slicer dimension is similar to an axis dimension in its purpose, but has limitations that axis dimensions do not share.

> **Note** Microsoft® SQL Server™ 2000 Analysis Services supports a maximum of 128 shared or private dimensions in a cube, in addition to the Measures dimension. Therefore, MDX queries on Analysis Services cubes are limited to 129 axes maximum.

# Specifying the Contents of an Axis Dimension

Axis dimensions determine the edges of a multidimensional result set. Multidimensional Expressions (MDX) uses the SELECT clause to specify axis dimensions by assigning a set to a particular axis. The following information describes how this assignment is handled in MDX.

In the following syntax example, each `<axis_specification>` value defines one axis dimension. The number of axes in the dataset is equal to the number of `<axis_specification>` values in the Multidimensional Expressions (MDX) query. An MDX query can support up to 128 specified axes, but very few MDX queries will use more than 5 axes.

The breakdown of the <axis_specification> syntax is:

```
<axis_specification> ::= <set> ON <axis_name>
<axis_name> ::= COLUMNS | ROWS | PAGES | SECTIONS | CHAPTERS | AXIS(<index>)
```

Each axis dimension is associated with a number: 0 for the x-axis, 1 for the y-axis, 2 for the z-axis, and so on. The `<index>` value is the axis number. For the first 5 axes, the aliases COLUMNS, ROWS, PAGES, SECTIONS, and CHAPTERS can be used in place of AXIS(0), AXIS(1), AXIS(2), AXIS(3), and AXIS(4), respectively.

An MDX query cannot skip axes. That is, a query that includes one or more `<axis_name>` values must not exclude lower-numbered or intermediate axes. For example, a query cannot have a ROWS axis without a COLUMNS axis, or have COLUMNS and PAGES axes without a ROWS axis.

However, you can specify a SELECT clause with no axes (that is, an empty SELECT clause). In this case, all dimensions are slicer dimensions, and the MDX query selects one cell.

Each `<set>` value defines the contents of the axis. For more information about sets, see Members, Tuples, and Sets.

# Specifying the Contents of a Slicer Dimension

Slicer dimensions filter multidimensional data. You can use them to limit the data returned by including them in the WHERE clause of a Multidimensional Expressions (MDX) query.

Dimensions that are not explicitly assigned to an axis are assumed to be slicer dimensions and filter with their default members. The default member of a dimension can be explicitly specified in its **Default Member** property in Analysis Manager. This property is equivalent to the **DefaultMember** property in Decision Support Objects (DSO). If no default member is explicitly specified, the default member is the All member if an (All) level exists, or else an arbitrary member of the highest level. (The name of the All member is not necessarily All.)

Slicer dimensions can also be specified explicitly by using the WHERE clause of the MDX syntax. The breakdown of the WHERE clause syntax is:

```
[WHERE [<slicer_specification>]]
```

The member name [All] will probably not be unique within the cube, because many dimensions possess an [All] level. It is recommended that you qualify it with the dimension name to make it unambiguous. The following example demonstrates the use of the WHERE clause and the All member:

```
WHERE ( [Route].[All], [Time].[1st half] )
```

A slicer dimension can accept only expressions that evaluate into a single tuple. This does not mean that only a single tuple can be explicitly stated in the slicer dimension, as the following example shows:

```
WHERE ( [Time].[1st half], [Route].[nonground] )
```

If a set of tuples is supplied as the slicer expression, MDX will attempt to evaluate the set, aggregating the result cells in every tuple along the set. In other words, MDX will attempt to

use the **Aggregate** function on the set, aggregating each measure by its associated aggregation function. The following examples show a valid WHERE clause using a set of tuples:

```
WHERE { ([Time].[1st half], [Route].[nonground]), ([Time].[1st half], [Route].[ground]) }
```

If the «slicer_specification» cannot be resolved into a single tuple, an error will occur.

For more information about the **Aggregate** function, see Aggregate.

# Establishing Cube Context

To establish cube context, indicate the cube on which you want the Multidimensional Expressions (MDX) query to run. The FROM clause in an MDX query determines the cube context. The following syntax indicates which cube supplies the context for the MDX query:

```
FROM «cube_specification»
```

The «cube_specification» is completed with the name of a single cube.

For example, if an MDX query is to be run against the SalesCube cube, the FROM clause would be:

```
FROM SalesCube
```

This does not limit you from working with more than one cube at a time; you can use the **LookupCube** function to retrieve data from cubes outside the cube context. The following syntax will cause an error, because unlike SQL, the FROM clause in an MDX query does not usually permit joins:

```
FROM SalesCube, OtherCube
```

However, some OLAP providers may permit the joining of cubes along congruent dimensions; if two cubes share a dimension, the cubes can be joined using this syntax, in a fashion similar to that of linked cubes. For more information about joining cubes, see your OLAP provider documentation.

For more information about the FROM clause in the MDX SELECT statement, see SELECT Statement.

# Advanced MDX

The Multidimensional Expressions (MDX) syntax is designed not only to extract simple data from multidimensional data sources, but also to provide additional functionality to create named sets, calculated members, and write information back to dimensions and cells.

The following topics cover the more advanced aspects of MDX syntax.

# Creating and Using Property Values

Multidimensional Expressions (MDX) supports intrinsic and custom properties for dimensions, levels, members, and cells. The intrinsic properties are used to provide unique names, captions, and even formatting and font sizes for individual cells. Custom properties, on the other hand, can be used to provide almost any kind of additional attribute to members.

## Using Member Properties

In the axis specification for a given axis, the set expression selects tuples to populate the axis. The dataset returns some basic information about each member in each tuple, such as the member name, parent level, the number of children, and so on. These are referred to as member properties. Members often have additional properties associated with them, and member properties are available for all members at a given level. In terms of organization, member properties are treated as dimensionally organized data, stored on a single dimension.

For example, the Products level may offer the SKU, SRP, Weight, and Volume properties for each product. These properties are not members, but contain additional information about members at the Products level. All members support intrinsic member properties, such as the formatted value of a member, while dimensions and levels supply additional intrinsic dimension and level member properties, such as the ID of a member. Additional member properties can be created in Analysis Manager using Dimension Editor or Cube Editor, or with Multidimensional Expressions (MDX) statements. Member properties can be retrieved through the use of the DIMENSION PROPERTIES keyword or the **Properties** function.

### DIMENSION PROPERTIES Keyword

An application might want to extend member information by adding member properties on the axis. Therefore, each level of each dimension may contain a set of available properties for the members.

The DIMENSION PROPERTIES keyword is used to specify member properties to be used for a given axis dimension. The following syntax defines the MDX SELECT syntax, adding the syntax for the DIMENSION PROPERTIES keyword:

```
SELECT [<axis_specification>
       [, <axis_specification>...]]
  FROM [<cube_specification>]
[WHERE [<slicer_specification>]]
```

The `<axis_specification>` value includes an optional `<dim_props>` value, which enables querying of dimension, level, and member properties using the DIMENSION PROPERTIES keyword. The breakdown of the `<axis_specification>` syntax with the `<dim_props>` value is:

```
<axis_specification> ::= <set> [<dim_props>] ON <axis_name>
```

The `<set>` and `<axis_name>` values are described in Specifying the Contents of an Axis Dimension. The breakdown of the `<dim_props>` syntax is:

```
<dim_props> ::= [DIMENSION] PROPERTIES <property> [,<property>...]
```

The breakdown of the `<property>` syntax varies depending on the property you are querying. Intrinsic member properties for dimensions and levels must be preceded with the name of the dimension and/or level. Intrinsic member properties for members cannot be qualified by the dimension or level name. Custom member properties should be preceded by the name of the level in which they reside.

Additional member properties can be selected by using the DIMENSION PROPERTIES keyword after the set expression of the axis specification. For example, the following MDX query:

```
SELECT
   CROSSJOIN(Years, (Sales, BudgetedSales)) ON COLUMNS,
   NON EMPTY Products.MEMBERS
   DIMENSION PROPERTIES Products.SKU, Products.SRP ON ROWS
FROM SalesCube
WHERE (January, SalesRep.[All], Geography.USA)
```

returns the following dataset:

| Products | SKU | SRP | 1996 | | 1997 | |
|---|---|---|---|---|---|---|
| | | | Sales | Budgeted Sales | Sales | Budgeted Sales |
| Ceiling Fan | CF675 | $15,000 | 3 | 5000 | 2 | 4500 |
| 75W Light Bulb | LB150 | $254.99 | 1 | 25000 | 0 | 19000 |
| AC Adapter, 12V | AA120 | $1.99 | 52000 | 10000 | 99000 | 50000 |
| AC Adapter, 6V | AA60 | $735.60 | 4 | 9000 | 1 | 5000 |

You can specify only those dimension properties projected on the axis for that particular axis. You can mix requests for intrinsic dimension and level member properties in the same query

with intrinsic member properties. The difference between intrinsic dimension and level member properties and intrinsic member properties is explained in greater detail later in this topic.

### Properties Function

Member properties can also be retrieved by the use of the **Properties** function in MDX. For example, the following MDX query uses the WITH keyword to create a calculated member consisting of the [Store Sqft] member property:

```
WITH
   MEMBER [Measures].[Store Size] AS
   'Val(Store.CurrentMember.Properties("Store Sqft"))'

SELECT
   {[Measures].[Unit Sales], [Measures].[Store Size]} ON COLUMNS,
   {[Store].[Store Name].Members} ON ROWS
From Sales
```

to generate a result set similar to the one in the following table:

| Store Name | Unit Sales | Store Size |
|---|---|---|
| Store 14 | 2,117.00 | 22478 |
| Store 11 | 26,079.00 | 20319 |
| Store 13 | 41,580.00 | 27694 |
| Store 2 | 2,237.00 | 28206 |

For more information about building calculated members, see Building Calculated Members in MDX.

Note the use of the **Val**() function in the MDX query example. The **Properties** function is a string function; all member properties retrieved with the **Properties** function will be coerced into strings.

### Intrinsic Dimension and Level Member Properties

All dimensions and levels support a list of intrinsic member properties, displayed in the following table. These member properties are used in the context of a specific dimension or level, and supply values for each member of the specified dimension or level. For example, specifying the following statement in a Multidimensional Expressions (MDX) query:

```
[Sales].Name
```

returns the names of each referenced member of the [Sales] dimension.

| Property | Description |
|---|---|
| ID | The internally maintained ID for the member |
| Key | The value stored in the MEMBER_KEY column of the MEMBERS schema rowset for the member |
| Name | The name of the member |

Dimension member properties are preceded by the name of the dimension to which the property applies. The following example demonstrates the appropriate syntax:

```
DIMENSION PROPERTIES «Dimension».ID
```

Level member properties can be preceded with the level name or, for additional specification, the dimension and level name, as shown here:

```
DIMENSION PROPERTIES [«Dimension».]«Level».ID
```

### Intrinsic Member Properties

All members support a list of intrinsic member properties as well, displayed in the following table. Intrinsic member properties cannot be requested for a specific dimension or level; they apply to all members of an axis dimension in a Multidimensional Expressions (MDX) query. Specifying, for example, the following statement in an MDX query:

```
PROPERTIES DESCRIPTION
```

returns the description of each member in the axis dimension.

The following table lists the intrinsic member properties supported by Microsoft® SQL Server™ 2000 Analysis Services.

| Property | Description |
|---|---|
| CALCULATION_PASS_DEPTH | For calculated cells only. The pass depth for the calculation formula, this property determines how many passes are needed to resolve the calculation formula. For more information about pass order, see Understanding Pass Order and Solve Order. |
| CALCULATION_PASS_NUMBER | For calculated cells only. The pass number for the calculation formula, this property determines on which pass the calculation formula will begin evaluation and end calculation. The default for this property is 1; its maximum value is 65,535. For more information about pass order, see Understanding Pass Order and Solve Order. |
| CATALOG_NAME | The name of the catalog to which this member belongs. |
| CHILDREN_CARDINALITY | The number of children that the member has. This can be an estimate, so you should not rely on this to be the exact count. Providers should return the best estimate possible. |
| CONDITION | For calculated cells only. The calculation condition of the calculated cells. This property receives an MDX logical expression, which is evaluated on each cell in the calculation subcube. If it returns True, the calculation formula is applied and the cell returns the resulting value. If it returns False, the cell returns the original cell value. If not specified, CONDITION defaults to True (in other words, the calculation formula applies to all cells in the calculation subcube.) |
| CUBE_NAME | The name of the cube to which this member belongs. |
| DESCRIPTION | A human-readable description of the member or calculated cells definition. |
| DIMENSION_UNIQUE_NAME | The unique name of the dimension to which this member belongs. For providers that generate unique names by qualification, each component of this name is delimited. |
| DISABLED | For calculated cells only. A Boolean property that indicates whether or not the calculated cells are disabled. DISABLED defaults to False. |
| HIERARCHY_UNIQUE_NAME | The unique name of the hierarchy. If the member belongs to more than one hierarchy, there is one row for each hierarchy to which it belongs. For providers that generate unique names by qualification, each component of this name is delimited. |
| LEVEL_NUMBER | The distance of the member from the root of the hierarchy. The root level is zero. |
| LEVEL_UNIQUE_NAME | Unique name of the level to which the member belongs. For providers that generate unique names by qualification, each component of this name is delimited. |
| MEMBER_CAPTION | A label or caption associated with the member. It is used primarily for display purposes. If a caption does not exist, MEMBER_NAME is returned. |
| MEMBER_GUID | The member GUID. |
| MEMBER_NAME | The name of the member. |
| MEMBER_ORDINAL | The ordinal number of the member. This is the sort rank of the member when members of this dimension are sorted in their natural sort order. If providers do not have the concept of natural ordering, this should be the |

| | |
|---|---|
| | rank when sorted by MEMBER_NAME. |
| MEMBER_TYPE | The type of the member. It can be one of the following values:<br>MDMEMBER_TYPE_REGULAR<br>MDMEMBER_TYPE_ALL<br>MDMEMBER_TYPE_FORMULA<br>MDMEMBER_TYPE_MEASURE<br>MDMEMBER_TYPE_UNKNOWN<br>MDMEMBER_TYPE_FORMULA takes precedence over MDMEMBER_TYPE_MEASURE. Therefore, if there is a formula (calculated) member on the Measures dimension, it is listed as MDMEMBER_TYPE_FORMULA. |
| MEMBER_UNIQUE_NAME | The unique name of the member. For providers that generate unique names by qualification, each component of this name is delimited. |
| PARENT_COUNT | The number of parents that this member has. |
| PARENT_LEVEL | The distance of the member's parent from the root level of the hierarchy. The root level is zero. |
| PARENT_UNIQUE_NAME | The unique name of the member's parent. NULL is returned for any members at the root level. For providers that generate unique names by qualification, each component of this name is delimited. |
| SCHEMA_NAME | The name of the schema to which this member belongs. |

Columns in the MEMBERS schema rowset support the intrinsic member properties. For more information about the MEMBERS schema rowset, see MDSCHEMA_MEMBERS. Other intrinsic member properties can be supported, depending upon the provider. However, all providers must support the intrinsic member properties listed here to be compliant with the OLAP section of the OLE DB specification dated March 1999 (2.6).

Intrinsic member properties are used without additional specification of any sort, as intrinsic member properties apply to all members. The following syntax example demonstrates usage:

```
PROPERTIES «Property»
```

> **IMPORTANT** Because intrinsic member properties cannot be qualified by the dimension or level name, a consumer cannot choose different intrinsic member properties for different dimensions (or levels) on an axis. For example, if the ROWS axis has Geography and SalesRep dimensions, the consumer cannot choose the MEMBER_CAPTION intrinsic member property for the Geography dimension or the MEMBER_UNIQUE_NAME intrinsic member property for the SalesRep dimension. The consumer must choose the same intrinsic member property (or properties) for all dimensions on an axis.

### Custom Member Properties

Custom member properties can be added to a specific named level in a dimension. Custom member properties cannot be added to the (All) level of a dimension, or to the dimension itself. Custom member properties can be added to server based dimensions or cubes using Dimension Editor or Cube Editor in Analysis Manager, or by an application using the Decision Support Objects (DSO) library. Additionally, custom member properties can be defined as part of the CREATE CUBE statement when creating local cubes in PivotTable® Service.

The syntax used to refer to custom member properties is similar to that used to refer to intrinsic level member properties, as demonstrated in the following example:

```
PROPERTIES [«Dimension».]«Level».«Custom Member Property»
```

## Using Cell Properties

Cell properties in Multidimensional Expressions (MDX) contain information about the content and format of cells in a multidimensional data source, such as a cube. MDX supports the CELL PROPERTIES keyword in an MDX SELECT statement to retrieve intrinsic cell properties.

Intrinsic cell properties are most commonly used to assist in the visual presentation of cell data.

The following example displays the syntax of the MDX SELECT statement, with the CELL PROPERTIES keyword syntax included.

```
SELECT [<axis_specification>
       [, <axis_specification>...]]
  FROM [<cube_specification>]
[WHERE [<slicer_specification>]]
[<cell_props>]
```

The syntax of the `<cell_props>` value is displayed here, and it employs the CELL PROPERTIES keyword along with one or more intrinsic cell properties:

```
<cell_props> ::= CELL PROPERTIES <property> [, <property>...]
```

The supported intrinsic cell properties used in the `<property>` value are listed in the following table, with brief descriptions on the content of the cell property.

| Property | Description |
|---|---|
| BACK_COLOR | The background color for displaying the VALUE or FORMATTED_VALUE property. For more information, see FORE_COLOR and BACK_COLOR Contents. |
| CELL_EVALUATION_LIST | The semicolon-delimited list of evaluated formulas applicable to the cell, in order from lowest to highest solve order. For more information about solve order, see Understanding Pass Order and Solve Order |
| CELL_ORDINAL | The ordinal number of the cell in the dataset. |
| FORE_COLOR | The foreground color for displaying the VALUE or FORMATTED_VALUE property. For more information, see FORE_COLOR and BACK_COLOR Contents. |
| FONT_NAME | The font to be used to display the VALUE or FORMATTED_VALUE property. |
| FONT_SIZE | Font size to be used to display the VALUE or FORMATTED_VALUE property. |
| FONT_FLAGS | The bitmask detailing effects on the font. The value is the result of a bitwise OR operation of one or more of the following constants:<br>MDFF_BOLD = 1<br>MDFF_ITALIC = 2<br>MDFF_UNDERLINE = 4<br>MDFF_STRIKEOUT = 8<br>For example, the value 5 represents the combination of bold (MDFF_BOLD) and underline (MDFF_UNDERLINE) font effects. |
| FORMAT_STRING | The format string used to create the FORMATTED_VALUE property value.<br>For more information, see FORMAT_STRING Contents. |
| FORMATTED_VALUE | The character string that represents a formatted display of the VALUE property. |
| NON_EMPTY_BEHAVIOR | The measure used to determine the behavior of calculated members when resolving empty cells. |
| SOLVE_ORDER | The solve order of the cell. |
| VALUE | The unformatted value of the cell. |

Providers are not required to support all intrinsic cell properties; only the **CELL_ORDINAL**, **FORMATTED_VALUE**, and **VALUE** cell properties must be supported. All cell properties, intrinsic or provider-specific, are defined in the PROPERTIES schema rowset, including their data types and provider support. For more information about the PROPERTIES schema rowset, see MDSCHEMA_PROPERTIES.

By default, if the CELL PROPERTIES keyword is not used, the cell properties returned are **VALUE**, **FORMATTED_VALUE**, and **CELL_ORDINAL** (in that order). If the CELL PROPERTIES keyword is used, only those cell properties explicitly stated with the keyword are returned.

The following example demonstrates the use of the CELL PROPERTIES keyword in an MDX query:

```
SELECT
   {[Measures].[Unit Sales], [Measures].[Store Size]} ON COLUMNS,
   {[Store].[Store Name].Members} ON ROWS
FROM Sales
CELL PROPERTIES VALUE, FORMATTED_VALUE, FORMAT_STRING, FORE_COLOR, BACK_COLOR
```

Cell properties are not returned for MDX queries that return flattened rowsets; in this case, each cell is represented as if only the **FORMATTED_VALUE** cell property were returned.

### Custom Member Options

Cell properties can be set through Analysis Manager by using the **Custom Member Options** property of Dimension Editor or Cube Editor. The **Custom Member Options** property accepts a column reference containing, for each member, a comma-delimited list of cell properties. The cell properties are represented as string expressions, shown in the following example.

```
FORE_COLOR='255',BACK_COLOR='65535'
```

The example will provide, for the specified member, a yellow background with a red foreground. Cell properties usually roll up to parent members, unless the parent is a custom member with cell properties. In this case, the parent cell properties override the cell properties derived from its children.

### FORMAT_STRING Contents

The cell property **FORMAT_STRING** is used to format the **VALUE** cell property, creating the value for the **FORMATTED_VALUE** cell property. The **FORMAT_STRING** cell property handles both string and numeric raw values, applying a format expression against the value to return a formatted value for the **FORMATTED_VALUE** cell property. The following tables detail the syntax and formatting characters used to handle string and numeric values.

### String Values

A format expression for strings can have one section or two sections separated by a semicolon (;).

| Usage | Result |
|---|---|
| One section | The format applies to all string values. |
| Two sections | The first section applies to string data, whereas the second section applies to null values and zero-length strings (""). |

The characters described in the following table can appear in the format string for character strings.

| Character | Description |
|---|---|
| @ | Character placeholder. It displays a character or a space. If the string has a character in the position where the at sign (@) appears in the format string, it displays the character. Otherwise, it displays a space in that position. Placeholders are filled from right to left unless there is an exclamation point (!) in the format string. |
| & | Character placeholder. It displays a character or nothing. If the string has a character in the position where the ampersand (&) appears, it displays the character. Otherwise, it displays nothing. Placeholders are filled from right to left unless there is an exclamation point (!) in the format string. |
| < | Forces lowercase. It displays all characters in lowercase format. |
| > | Forces uppercase. It displays all characters in uppercase format. |
| ! | Forces left-to-right fill of placeholders. (The default is to fill placeholders from right to left.) |

### Numeric Values

A user-defined format expression for numbers can have anywhere from one to four sections separated by semicolons. If the format argument contains one of the named numeric formats, only one section is allowed.

| Usage | Result |
|---|---|
| One section | The format expression applies to all values. |
| Two sections | The first section applies to positive values and zeros, the second to negative values. |
| Three sections | The first section applies to positive values, the second to negative values, and the third to zeros. |
| Four sections | The first section applies to positive values, the second to negative values, the third to zeros, and the fourth to null values. |

The following example has two sections: The first section defines the format for positive values and zeros, and the second section defines the format for negative values.

"$#,##0;($#,##0)"

If you include semicolons with nothing between them, the missing section is printed using the format of the positive value. For example, the following format displays positive and negative values using the format in the first section and displays "Zero" if the value is zero:

"$#,##0;;\Z\e\r\o"

The following table identifies the characters that can appear in the format string for number formats.

| Character | Description |
|---|---|
| None | Displays the number with no formatting. |
| 0 | Digit placeholder. Displays a digit or a zero.<br>If the expression has a digit in the position where the 0 appears in the format string, it displays the digit. Otherwise, it displays a zero in that position.<br>If the number has fewer digits than there are zeros (on either side of the decimal) in the format expression, it displays leading or trailing zeros.<br>If the number has more digits to the right of the decimal separator than there are zeros to the right of the decimal separator in the format expression, it rounds the number to as many decimal places as there are zeros.<br>If the number has more digits to the left of the decimal separator than there are zeros to the left of the decimal separator in the format expression, it displays the extra digits without modification. |
| # | Digit placeholder. Displays a digit or nothing. If the expression has a digit in the position where the # appears in the format string, it displays the digit. Otherwise, it displays nothing in that position. This symbol works like the 0 digit placeholder except that leading and trailing zeros are not displayed if the number has the same or fewer digits than there are # characters on either side of the decimal separator in the format expression. |
| . | Decimal placeholder. (In some locales, a comma is used as the decimal separator.) The decimal placeholder determines how many digits are displayed to the left and right of the decimal separator. If the format expression contains only number signs (#) to the left of this symbol, numbers smaller than 1 begin with a decimal separator. To display a leading zero displayed with fractional numbers, use 0 as the first digit placeholder to the left of the decimal separator. The actual character used as a decimal placeholder in the formatted output depends on the number format recognized by your system. |
| % | Percentage placeholder. The expression is multiplied by 100. The percent character (%) is inserted in the position where it appears in the format string. |
| , | Thousand separator. (In some locales, a period is used as a thousand separator.) The thousand separator separates thousands from hundreds within a number that has four or more places to the left of the decimal separator. Standard use of the thousand separator is specified if the format contains a thousand separator surrounded by digit placeholders (0 or #). Two adjacent thousand separators, or a thousand separator immediately to the left of the decimal separator (whether or not a decimal is specified), means "scale the number by dividing it by 1000, rounding as needed." For example, you can use the format string "##0,," to represent 100 million as 100. Numbers smaller than 1 million are displayed as |

| | |
|---|---|
| | 0. Two adjacent thousand separators in any position other than immediately to the left of the decimal separator are treated simply as specifying the use of a thousand separator. The actual character used as the thousand separator in the formatted output depends on the number format recognized by your system. |
| : | Time separator. (In some locales, other characters may be used to represent the time separator.) The time separator separates hours, minutes, and seconds when time values are formatted. The actual character used as the time separator in formatted output is determined by your system settings. |
| / | Date separator. (In some locales, other characters may be used to represent the date separator.) The date separator separates the day, month, and year when date values are formatted. The actual character used as the date separator in formatted output is determined by your system settings. |
| E- E+ e- e+ | Scientific format. If the format expression contains at least one digit placeholder (0 or #) to the right of E-, E+, e-, or e+, the number is displayed in scientific format and E or e is inserted between the number and its exponent. The number of digit placeholders to the right determines the number of digits in the exponent. Use E- or e- to place a minus sign next to negative exponents. Use E+ or e+ to place a minus sign next to negative exponents and a plus sign next to positive exponents. |
| - + $ ( ) | Displays a literal character. To display a character other than one of those listed, precede it with a backslash (\) or enclose it in double quotation marks (" "). |
| \ | Displays the next character in the format string. To display a character that has special meaning as a literal character, precede it with a backslash (\). The backslash itself is not displayed. Using a backslash is the same as enclosing the next character in double quotation marks. To display a backslash, use two backslashes (\\). Examples of characters that cannot be displayed as literal characters are the date-formatting and time-formatting characters (a, c, d, h, m, n, p, q, s, t, w, y, /, and :), the numeric-formatting characters (#, 0, %, E, e, comma, and period), and the string-formatting characters (@, &, <, >, and !). |
| "ABC" | Displays the string inside the double quotation marks (" "). To include a string in format from within code, use Chr(34) to enclose the text. (The character code for a double quotation mark is 34.) |

### Date Values

The following table identifies characters that can appear in the format string for date/time formats.

| Character | Description |
|---|---|
| : | Time separator. (In some locales, other characters may be used to represent the time separator.) The time separator separates hours, minutes, and seconds when time values are formatted. The actual character used as the time separator in formatted output is determined by your system settings. |
| / | Date separator. (In some locales, other characters may be used to represent the date separator.) The date separator separates the day, month, and year when date values are formatted. The actual character used as the date separator in formatted output is determined by your system settings. |
| C | Displays the date as ddddd and displays the time as ttttt, in that order. Displays only date information if there is no fractional part to the date serial number. Displays only time information if there is no integer portion. |
| d | Displays the day as a number without a leading zero (1–31). |
| dd | Displays the day as a number with a leading zero (01–31). |
| ddd | Displays the day as an abbreviation (Sun–Sat). |
| dddd | Displays the day as a full name (Sunday–Saturday). |
| ddddd | Displays the date as a complete date (including day, month, and year), formatted according to your system's short date format setting. For Microsoft® Windows®, the default short date format is m/d/yy. |

| | |
|---|---|
| dddddd | Displays a date serial number as a complete date (including day, month, and year), formatted according to the long date setting recognized by your system. For Windows, the default long date format is mmmm dd, yyyy. |
| w | Displays the day of the week as a number (1 for Sunday through 7 for Saturday). |
| ww | Displays the week of the year as a number (1–54). |
| m | Displays the month as a number without a leading zero (1–12). If m immediately follows h or hh, the minute rather than the month is displayed. |
| mm | Displays the month as a number with a leading zero (01–12). If m immediately follows h or hh, the minute rather than the month is displayed. |
| mmm | Displays the month as an abbreviation (Jan–Dec). |
| mmmm | Displays the month as a full month name (January–December). |
| q | Displays the quarter of the year as a number (1–4). |
| y | Displays the day of the year as a number (1–366). |
| yy | Displays the year as a two-digit number (00–99). |
| yyyy | Displays the year as a four-digit number (100–9999). |
| h | Displays the hour as a number without leading zeros (0–23). |
| hh | Displays the hour as a number with leading zeros (00–23). |
| n | Displays the minute as a number without leading zeros (0–59). |
| nn | Displays the minute as a number with leading zeros (00–59). |
| s | Displays the second as a number without leading zeros (0–59). |
| ss | Displays the second as a number with leading zeros (00–59). |
| t t t t t | Displays a time as a complete time (including hour, minute, and second), formatted using the time separator defined by the time format recognized by your system. A leading zero is displayed if the leading zero option is selected and the time is earlier than 10:00 (for example 09:59), in either the A.M. or the P.M. cycle. For Windows, the default time format is h:mm:ss. |
| AM/PM | Uses the 12-hour clock. Displays an uppercase with any hour from midnight until noon; displays an uppercase AMPM with any hour from noon until midnight. |
| am/pm | Uses the 12-hour clock. Displays a lowercase am with any hour from midnight until noon; displays a lowercase pm with any hour from noon until midnight. |
| A/P | Uses the 12-hour clock. Displays an uppercase A with any hour from midnight until noon; displays an uppercase P with any hour from noon until midnight. |
| a/p | Uses the 12-hour clock. Displays a lowercase a with any hour from midnight until noon; displays a lowercase p with any hour from noon until midnight. |
| AMPM | Uses the 12-hour clock. Displays the AM string literal as defined by your system with any hour from midnight until noon; displays the PM string literal as defined by your system with any hour from noon until midnight. AMPM can be either uppercase or lowercase, but the case of the string displayed matches the string as defined by your system settings. For Windows, the default format is AM/PM. |

**FORE_COLOR and BACK_COLOR Contents**

The **FORE_COLOR** and **BACK_COLOR** cell properties are used to store color information for the text and the background of a cell, respectively, in the Microsoft® Windows® operating system red-green-blue (RGB) format.

The valid range for a normal RGB color is 0 to 16,777,215 (&H00FFFFFF). The high byte of a number in this range always equals 0; the lower 3 bytes, from least to most significant byte, determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number between 0 and 255 (&HFF).

For example, the value 255 (&H000000FF) represents red, the value (65280 (&H0000FF00) represents green, and the value 16711680 (&H00FF0000) represents blue.

# Building Named Sets in MDX

A set in Multidimensional Expressions (MDX) can be a lengthy and complex declaration, and difficult to follow or understand. For example, the following MDX query examines the unit sales of the various Chardonnay and Chablis wines in **FoodMart 2000**:

```
SELECT
  {[Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Good].[Good Chardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Pearl].[Pearl Chardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Portsmouth].[PortsmouthChardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Top Measure].[TopMeasureChardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Walrus].[Walrus Chardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Good].[Good Chablis Wine],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Pearl].[Pearl Chablis Wine],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Portmouth].[Portmouth Chablis Wine],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[TopMeasure].[TopMeasureChablisWine],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Walrus].[Walrus Chablis Wine]}
  ON COLUMNS,
  {Measures.[Unit Sales]} ON ROWS
FROM Sales
```

The MDX query, although fairly simple in terms of the result set, is lengthy and unwieldy when it comes to maintenance.

One method of easing maintenance and increasing understandability of an MDX query such as the previous example is to create a named set. A named set is simply a set expression associated with an alias. A named set can incorporate member or function that can normally be incorporated into a set. The named set alias is treated as a set expression, and can be used anywhere a set expression is accepted.

To illustrate, the previous MDX query example is rewritten to employ a named set, as shown in the following example:

```
WITH SET [ChardonnayChablis] AS
   '{[Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Good].[Good Chardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Pearl].[Pearl Chardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Portsmouth].[PortsmouthChardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Top Measure].[TopMeasureChardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Walrus].[Walrus Chardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Good].[Good Chablis Wine],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Pearl].[Pearl Chablis Wine],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Portmouth].[Portmouth Chablis Wine],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[TopMeasure].[TopMeasureChablisWine],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Walrus].[Walrus Chablis Wine]}'

SELECT
   [ChardonnayChablis] ON COLUMNS,
   {Measures.[Unit Sales]} ON ROWS
FROM Sales
```

The WITH keyword is used to create the `[ChardonnayChablis]` named set, which is then reused in the MDX SELECT statement. In this fashion, the set created with the WITH keyword can be changed without disturbing the MDX SELECT statement. For more information about using the WITH keyword to create named sets, see Using WITH to Create Named Sets.

The named set makes the MDX query example a bit easier to follow, but still difficult to maintain because the named set is defined as part of the MDX query itself. The scope of the named set is limited to this MDX query alone, and is not reusable.

MDX and PivotTable® Service, however, offer the capability of creating a named set with a wider scope. The CREATE SET statement allows the client application to create a named set that exists for the lifetime of the MDX session, making the named set available to all MDX queries in that session. The CREATE SET statement makes sense, for example, in a client

application that consistently reuses a set in a variety of queries. For more information about using the CREATE SET to create named sets in a session, see CREATE SET Statement.

Even this scope, however, may be limiting in terms of maintenance. Microsoft® SQL Server™ 2000 Analysis Services offers the capability of creating global named sets, stored as part of a cube. For more information about creating global named sets, see Creating Named Sets.

## Using WITH to Create Named Sets

The WITH keyword is included as part of the MDX SELECT statement, to allow construction of named sets as part of an MDX query.

The following syntax is used to add the WITH keyword to the MDX SELECT statement:

```
[WITH <formula_specification>
      [ <formula_specification>...]]
SELECT [<axis_specification>
       [, <axis_specification>...]]
  FROM [<cube_specification>]
[WHERE [<slicer_specification>]]
```

The `<formula_specification>` value for named sets is further broken out in the following syntax definition:

```
<formula_specification> ::= SET <set_name> AS '<set>'
```

The <set_name> parameter contains the alias for the named set. The <set> parameter contains the set expression to which the named set alias will refer.

For example, the [ChardonnayChablis] named set is used to refer specifically to all of the Chardonnay and Chablis wine members in the Product dimension of the **FoodMart 2000** database. The syntax for the named set is depicted in the following example:

```
WITH SET [ChardonnayChablis] AS
 '{[Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Good].[Good Chardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Pearl].[Pearl Chardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Portsmouth].[PortsmouthChardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Top Measure].[TopMeasureChardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Walrus].[Walrus Chardonnay],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Good].[Good Chablis Wine],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Pearl].[Pearl Chablis Wine],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Portmouth].[Portmouth Chablis Wine],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[TopMeasure].[TopMeasureChablisWine],
   [Prod].[All Prods].[Drink].[AlcBev].[B&W].[Wine].[Walrus].[Walrus Chablis Wine]}'
```

You can also use MDX functions in the set expression used to create a named set. The following MDX query example uses the **Filter**, **CurrentMember**, **Name**, and **InStr** functions to create the [ChardonnayChablis] named set, as used in earlier MDX query examples in this topic.

```
WITH SET [ChardonnayChablis] AS
   'Filter([Prod].Members, (InStr(1, [Prod].CurrentMember.Name, "chardonnay") <> 0)
                           OR
                           (InStr(1, [Prod].CurrentMember.Name, "chablis") <> 0))'

SELECT
   [ChardonnayChablis] ON COLUMNS,
   {Measures.[Unit Sales]} ON ROWS
FROM Sales
```

# Building Calculated Members in MDX

In Multidimensional Expressions (MDX), a calculated member is defined as a member that is resolved not by retrieving data, but by calculating an MDX expression to return a value. This innocuous definition covers an incredible amount of ground; the ability to construct and use calculated members in an MDX query provides a great deal of manipulation capability for

multidimensional data. This topic discusses some of the simpler aspects of creating calculated members, as covered in the following table.

# Using WITH to Create Calculated Members

Similar to the way it is used in named sets, the WITH keyword in Multidimensional Expressions (MDX) is used to describe calculated members.

The following syntax is used to add the WITH keyword to the MDX SELECT statement:

```
[WITH <formula_specification>
      [ <formula_specification>...]]
SELECT [<axis_specification>
       [, <axis_specification>...]]
FROM [<cube_specification>]
[WHERE [<slicer_specification>]]
```

The `<formula_specification>` value for calculated members is further broken out in the following syntax definition:

```
<formula_specification> ::= MEMBER <member_name>
                            AS '<value_expression>'
                            [,SOLVE_ORDER = <unsigned integer>]
                            [,<cell_property>=<value_expression>...]
```

The `<member_name>` value is the fully qualified name of the calculated member, including the dimension or level to which the calculated member is associated, and the `<value_expression>` value, after it has been evaluated, returns the value of the calculated member. Optionally, the SOLVE_ORDER keyword can be used to specify the solve order of the calculated member; if not used, the solve order of the calculated member is set by default to 0.

The values of intrinsic cell properties for a calculated member can be optionally specified by supplying the name of the cell property in the `<cell_property>` value and the value of the cell property in the `<value_expression>` value.

For example, the following MDX query example defines two calculated members. The first calculated member, [Measures].[StoreType], is used to represent the Store Type member property. The second calculated member, [Measures].[ProfitPct], is used to calculate the total profit margin for a given store, and represent it as a formatted percentile value.

```
WITH
   MEMBER [Measures].[StoreType] AS
                                  '[Store].CurrentMember.Properties("Store Type")',
          SOLVE_ORDER = 2
   MEMBER [Measures].[ProfitPct] AS
        'Val((Measures.[Store Sales] - Measures.[Store Cost]) / Measures.[Store Sales])',
          SOLVE_ORDER = 1, FORMAT_STRING = 'Percent'
SELECT
   {[Store].[Store Name].Members} ON COLUMNS,
   {[Measures].[Store Sales],
    [Measures].[Store Cost],
    [Measures].[StoreType],
    [Measures].[ProfitPct] } ON ROWS
FROM Sales
```

Calculated members can be created at any point within a hierarchy. For example, the following MDX query example defines a calculated member, created as a child member of the [Beer and Wine] member, to determine whether a given store has at least 100.00 in unit sales for beer and wine:

```
WITH
   MEMBER [Prod].[B&W].[BigSeller] AS 'IIf([Prod].[B&W] > 100, "Yes","No")'
SELECT
   {[Prod].[BigSeller]} ON COLUMNS,
   {[Store].[Store Name].Members} ON ROWS
FROM Sales
```

You can also create calculated members that depend not only on existing members in a cube, but also on other calculated members defined in the same MDX expression. The following example illustrates such an MDX expression:

```
WITH
   MEMBER [Measures].[ProfitPct] AS
        'Val((Measures.[Store Sales] - Measures.[Store Cost]) / Measures.[Store Sales])',
          SOLVE_ORDER = 1, FORMAT_STRING = 'Percent'
   MEMBER [Measures].[ProfitValue] AS
        '[Measures].[Store Sales] * [Measures].[ProfitPct]',
          SOLVE_ORDER = 2, FORMAT_STRING = 'Currency'
SELECT
   {[Store].[Store Name].Members} ON COLUMNS,
   {[Measures].[Store Sales],
    [Measures].[Store Cost],
    [Measures].[ProfitValue],
    [Measures].[ProfitPct] } ON ROWS
FROM Sales
```

The second calculated member, [Measures].[ProfitValue], uses the value created in the first calculated member, [Measures].[ProfitPct], to generate its value.

# Using Functions in Calculated Members

Calculated members in Multidimensional Expressions (MDX) are extremely flexible. One of the ways in which calculated members provide such flexibility is in the wide variety of functions available for use in MDX. Besides the intrinsic MDX functions provided by the Microsoft® SQL Server™ 2000 Analysis Services function library, calculated members can also take advantage of external function libraries to supply additional capability.

A discussion of all of the myriad ways to use calculated members is beyond the scope of this topic. Instead, this topic focuses on the most commonly employed operators and functions in calculated members, and how to use them.

### Operators

MDX supports a variety of arithmetic, logical, and comparison operators for use in MDX expressions.

### Arithmetic Operators

Arithmetic operators support a basic set of arithmetic operations. Arithmetic precedence is followed when resolving arithmetic operations; multiplication and division operators are processed first, followed by addition and subtraction operators. If all of the arithmetic operators used in an expression have the same order of precedence; for example, as in the statement `a + b + c + d`, the arithmetic operators are handled in a left to right order. The basic arithmetic operators supported are specified in the following table.

| Operator | Description |
|---|---|
| + | Addition |
| - | Subtraction and unary negation |
| * | Multiplication |
| / | Division |

### Comparison Operators

Comparison operators compare two string, numeric or date expressions and return TRUE or FALSE based on the outcome of the tested comparison. For the purposes of comparison, null values are treated as zero when a null value is compared with a nonnull value. To check for null values in a cell, use the **IsEmpty** or **Is** functions to return TRUE if the cell contains a null

value, FALSE otherwise. The TRUE and FALSE constants are supported; the TRUE constant evaluates to 1, while the FALSE constant evaluates to 0.

| Operator | Description |
|---|---|
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |
| <> | Not equal to |
| = | Equal to |

### Bitwise Operators

Bitwise operators return a TRUE or FALSE value based on the review of logical expressions. As the TRUE and FALSE constants are supported, either of the following MDX expressions is now valid:

```
([Measures].[IsTrue] AND [Measures].[IsFalse]) = 0
([Measures].[IsTrue] AND [Measures].[IsFalse]) = FALSE
```

Logical operators require expressions that can be evaluated to a logical value. Numeric expressions are implicitly converted to logical values before a logical comparison is performed. Any numeric expression that evaluates to 0 or NULL is considered FALSE, while any numeric expression that evaluates to something other than 0 is considered TRUE. String expressions are not implicitly converted; attempting to use a bitwise operator with string expressions will result in an error.

| Operator | Description |
|---|---|
| «Expression1» AND «Expression2» | Returns TRUE if both expressions are true, FALSE otherwise. |
| «Expression1» OR «Expression2» | Returns TRUE if either expression is true, FALSE otherwise. |
| NOT «Expression1» | Returns TRUE if the expression is not true, FALSE otherwise. |
| «Expression1» XOR «Expression2» | Returns TRUE if either expression, but not both, is true, FALSE otherwise. |

### Set Operators

Set operators are provided to deal with the creation, separation, and joining of sets, as described in the following table.

| Operator | Description |
|---|---|
| «Set1» + «Set1» | Performs the Union function on two sets. |
| «Set1» * «Set2» | Performs the Crossjoin function on two sets. |
| «Set1» - «Set2» | Performs the Except function on two sets. |
| «Member1»:«Member2» | Creates a naturally ordered set, with the two members as endpoints and all members between the two specified members included as members of the set. |

### Functions

MDX supplies a wide variety of functions for use in MDX expressions. This topic briefly touches on each category of functions, broken out by the type of data returned by the MDX functions in a specific category.

For more information about the categories of MDX functions, see MDX Function List.

### Numeric Functions

MDX supplies a rich set of numeric functions, which can be used to perform a variety of aggregation and statistical calculations.

Aggregate functions in MDX are used to quickly perform a calculation across a number of members, usually specified as a set. For example, the **Aggregate** function aggregates the cells formed by all the members in a set, and can do so much easier than attempting to perform a manual aggregation. The **Aggregate** function is extremely powerful when combined with a measure that produces a sum, as the following MDX query example demonstrates:

```
WITH
   MEMBER [Time].[1st Half Sales] AS 'Aggregate({Time.[Q1], Time.[Q2]})'
   MEMBER [Time].[2nd Half Sales] AS 'Aggregate({Time.[Q3], Time.[Q4]})',
   MEMBER [Time].[Difference] AS 'Time.[2nd Half Sales] - Time.[1st Half Sales]',
SELECT
   {[Store].[Store State].Members} ON COLUMNS,
   {[Time].[1st Half Sales],
    [Time].[2nd Half Sales],
    [Time].Difference} ON ROWS
FROM Sales
WHERE [Measures].[Store Sales]
```

The query produces the sum of the store sales for each state, with aggregations for the first and second halves of the year supplied by the first two calculated members using the **Aggregate** function, with a difference between the two supplied by a third calculated member.

MDX also supplies a list of statistical functions as well, for handling routine statistical calculations such as statistical covariance and standard deviation. For example, the **Median** function computes the median value across a set, as demonstrated in the following MDX query.

```
WITH
   MEMBER [Time].[1st Half Sales] AS 'Sum({[Time].[Q1], [Time].[Q2]})'
   MEMBER [Time].[2nd Half Sales] AS 'Sum({[Time].[Q3], [Time].[Q4]})'
   MEMBER [Time].[Median] AS 'Median(Time.Members)'
SELECT
   NON EMPTY {[Store].[Store Name].Members} ON COLUMNS,
   {[Time].[1st Half Sales],
    [Time].[2nd Half Sales],
    [Time].[Median]} ON ROWS
FROM Sales
WHERE [Measures].[Store Sales]
```

In this case, the [Time].[Median] calculated member provides the median value of store sales for each store, in addition to the aggregation of store sales for each half of the year for each store provided by the [Time].[1st Half Sales] and [Time].[2nd Half Sales] calculated members.

### String Functions

MDX supplies a number of string functions not just for string processing within MDX expressions, but to support user-defined functions in MDX as well. For example, the **MemberToStr** function converts a member reference to a string in the MDX format for use with a user-defined function, as user-defined functions cannot accept object references from MDX.

### Set Functions

Set functions are used to return sets in MDX, giving you the capability to easily build dynamically defined sets and quickly create reusable named sets. One of the most commonly used set functions, the **Members** function, returns all members, excluding calculated members, of a level or dimension as a set. The following MDX query example shows the **Members** function in action.

```
SELECT
   NON EMPTY {[Store].[Store Name].Members} ON COLUMNS,
   {Measures.[Store Sales]} ON ROWS
FROM Sales
```

The MDX query example returns the total store sales figures for each store in the Sales cube. Without the Members function, you would have to explicitly enter each and every store name for it to function as it does in the MDX query example.

### Tuple Functions

As with set functions, tuple functions are used to return tuples in MDX. Tuple functions are also supplied, such as the **StrToTuple** function, to aid user-defined functions in MDX. As user-defined functions cannot handle MDX object references, a user-defined function can pass back a string return value in MDX format, representing a tuple, and use the **StrToTuple** function to convert it to a valid tuple reference.

### Member Functions

Members are often referred to in calculated members; member functions allow calculated members to perform complex member retrieval, negotiating hierarchies and sets with equal ease.

The resolution of calculated members in MDX can be iterative in nature, as calculated members can be constructed based upon iteration over the members of a set. Functions in MDX such as **CurrentMember** allow you to take advantage of this iterative capability.

### Other Functions

MDX supplies other functions as well, including functions that deal with dimensions, hierarchies, levels, and arrays. For example, the **SetToArray** function allows user-defined functions to receive set references as a variant array of individual members represented as strings, allowing you to create user-defined functions that can supply set related functionality.

## Conditional Expressions

Another capability in Multidimensional Expressions (MDX) is the ability to create conditional expressions, expressions that return different information depending upon a decision made in the calculated member based on the existence of a condition.

### IIf Function

The **IIf** function in Multidimensional Expressions (MDX) can be used to perform simple, yes-or-no decisions. For example, consider the following MDX query example.

```
WITH MEMBER [Measures].[BigSeller] AS 'IIf(Measures.[Store Sales] > 20000, "Yes", "No")'

SELECT
   {[Store].[Store Name].Members} ON COLUMNS,
   {[Measures].[Store Sales],
    [Measures].[BigSeller]} ON ROWS
FROM Sales
```

The MDX query example returns two rows for each store in the Sales cube. One row, the [Measures].[Store Sales] member, supplies the total store sales for each store. The second row is a calculated member that, based on the store sales for each store, determines if the store is a "big seller". That is, the **IIf** function is used to check a simple yes-or-no condition. In this case, the condition is whether or not the store sales figure for each store is greater than $20,000.00. If it is, the value of the member for that store is Yes. If the store sales figure is equal to or less than $20,000.00, it returns the value No.

This is a simple but graphic example of the use of the **IIf** function to return different values based upon a single Boolean condition; other MDX functions and operators can be used to supply the returned values in the **IIf** function.

For more information about the syntax of the **IIf** function, see IIf.

# Building Caches in MDX

Another feature Multidimensional Expressions (MDX) provides to improve performance is the ability to load a commonly used slice of a cube into memory, caching it for faster retrieval.

Microsoft® SQL Server™ 2000 Analysis Services and PivotTable® Service automatically cache query definitions, data, and meta data on the server and client sides, respectively. This caching increases performance in those cases where queries are repeatedly requesting the same data or meta data, reducing network traffic or execution time.

The ability to create caches for specific data in MDX gives you complete control over the caching of data to be used repeatedly, allowing fine-tuning of query performance.

In terms of creation scope, caches are similar to named sets in that a cache may be created for the lifetime of a single query or a session.

To create a cache to be used at the session level, the CREATE CACHE statement can be used. The CREATE CACHE statement can be used to create caches at the query level, but the WITH statement can perform this task just as easily.

For example, the following MDX query uses the WITH statement to cache:

```
WITH CACHE AS '(Store.[Store Name].Members)'

SELECT
   {[Store].[Store Name].Members} ON COLUMNS,
   {[Measures].[Unit Sales]} ON ROWS
FROM Sales
```

While the WITH statement can be used to create a cache for a single query, the CREATE CACHE statement can be used to create caches at the session level, as well. The CREATE CACHE statement requires PivotTable Service in order to employ a session level cache.

For more information about the CREATE CACHE statement, see CREATE CACHE Statement.

## Using WITH to Create Caches

As with named sets and calculated members, the WITH keyword is also used to create query level caches, usable for the lifetime of a single query. The following syntax is used to add the WITH keyword to the MDX SELECT statement:

```
[WITH <formula_specification>
      [ <formula_specification>...]]
SELECT [<axis_specification>
       [, <axis_specification>...]]
  FROM [<cube_specification>]
[WHERE [<slicer_specification>]]
```

The `<formula_specification>` value for caches is further broken out in the following syntax definition:

```
<formula_specification> ::= CACHE AS '(<set>[, <set>...])'
```

The `<set>` value is the set expression used to create the cache. The `<set>` value can support the use of MDX set functions.

When using the `<set>` set expression for constructing a cache, the following rules apply:

- Each `<set>` must contain members from only one dimension. Each member must be distinct.
- Each `<set>` must be from a different dimension.
- The `<set>` cannot contain measures.

# Building Calculated Cells in MDX

Multidimensional Expressions (MDX) provides you with a number of tools for generating calculated values, such as calculated members, custom rollups, and custom members. Although powerful and versatile features, they provide limited functionality because they affect members, not cells. It is difficult to affect a specific set of cells, or a single cell for that matter, using these features.

The calculated cells feature provides this functionality by allowing you to define a specific slice of cells, called a calculation subcube, and apply a formula to each and every cell within the calculation subcube, subject to an optional condition that can be applied to each cell.

Calculated cells take advantage of the pass order feature in Microsoft® SQL Server™ 2000 Analysis Services to provide such complex functionality as goal-seeking formulas, by allowing recursive passes to be made with calculated cells, with calculation formulas applied at specific passes in the pass order.

For more information on pass order, see Understanding Pass Order and Solve Order.

In terms of creation scope, calculated cells are similar to calculated members in that calculated cells can be made globally available as part of a cube, or temporarily created for the lifetime of either a session or a single query.

To create calculated cells as part of a cube, use the CREATE CELL CALCULATION statement. For existing cubes, the ALTER CUBE statement can also be used to add calculated cells.

To create calculated cells for the lifetime of a session, use the CREATE CELL CALCULATION statement.

To create calculated cells for the lifetime of a query, use the WITH statement.

## Using WITH to Create Calculated Cells

Similar to the way it is used in calculated members, the WITH keyword in Multidimensional Expressions (MDX) is used to describe calculated cells.

The following syntax is used to add the WITH keyword to the MDX SELECT statement:

```
[WITH <formula_specification>
      [ <formula_specification>...]]
SELECT [<axis_specification>
       [, <axis_specification>...]]
  FROM [<cube_specification>]
[WHERE [<slicer_specification>]]
```

The `<formula_specification>` value for calculated cells is further broken out in the following syntax definition:

```
<formula_specification> ::= CELL CALCULATION <formula_name>
                            FOR '(<calculation_subcube>)'
                            AS '<calculation_formula>'
                            [,<calculation_property_list>]
```

The `<cell_property_list>` is further defined by the following syntax:

```
<cell_property_list> ::= <property_name = '<value>'
                         [, <property_name> = '<value>'...]
```

The `<formula_name>` value is the name of the calculated cells. The `<calculation_subcube>` contains a list of orthogonal, single-dimensional MDX set expressions, each of which must resolve to one of the following categories of sets.

| Category | Description |
|---|---|
| Empty set | An MDX set expression that resolves into an empty set. In this case, the set is ignored. |
| Single member set | An MDX set expression that resolves into a single member. |

| | |
|---|---|
| Set of level members | An MDX set expression that resolves into the members of a single level. An example of this is the «Level».Members MDX function. To include calculated members, use the «Level».AllMembers MDX function. |
| Set of descendants | An MDX set expression that resolves into the descendants of a specified member. An example of this is the Descendants(«Member», «Level», «Desc_flags») MDX function. |

If a dimension is not described in the `<calculation_subcube>` argument, it is assumed that all members are included for the purposes of constructing the calculation subcube. Therefore, if the `<calculation_subcube>` argument is NULL, the calculated cells definition applies to the entire cube.

The `<calculation_formula>` argument contains an MDX expression that evaluates to a cell value for all of the cells defined in the `<calculation_subcube>` argument.

The `<calculation property list>` argument contains a list of member properties to be applied to the cells specified in the `<calculation_subcube>` argument.

The following properties apply specifically to calculated cells.

| Property | Description |
|---|---|
| CALCULATION_PASS_DEPTH | The pass depth for the calculation formula, this property determines how many passes are needed to resolve the calculation formula. For more information about pass order, see Understanding Pass Order and Solve Order. |
| CALCULATION_PASS_NUMBER | The pass number for the calculation formula, this property determines on which pass the calculation formula will begin calculation. The default for this property is 1. For more information about pass order, see Understanding Pass Order and Solve Order. |
| CELL_EVALUATION_LIST | The semicolon-delimited list of evaluated formulas applicable to the cell, in order from lowest to highest solve order. For more information about solve order, see Understanding Pass Order and Solve Order |
| CONDITION | The calculation condition of the calculated cells, this property receives an MDX logical expression, which is evaluated on each cell in the calculation subcube. If it returns True, the calculation formula is applied and the cell returns the resulting value. If it returns False, the cell returns the original cell value. If not specified, CONDITION defaults to True (in other words, the calculation formula applies to all cells in the calculation subcube). |
| DESCRIPTION | A human-readable text description of the calculated cells definition. |
| DISABLED | A Boolean property which indicates whether or not the calculated cells are disabled. DISABLED defaults to False. |

Other standard cell properties, such as **FORE_COLOR** and **BACK_COLOR**, can be used as well.

For more information about using cell properties and using member properties, see Using Cell Properties and Using Member Properties.

### Additional Considerations

The calculation condition, specified by the **CONDITION** property, is processed only once, depending on the creation scope of the calculated cells definition. This provides increased performance for the evaluation of multiple calculated cells definitions, especially with overlapping calculated cells across cube passes.

If created at global scope, as part of a cube, the calculation condition is processed when the cube is processed. If cells are modified in the cube in any way, and the cells are included in the calculation subcube of a calculated cells definition, the calculation condition may not be

accurate until the cube is reprocessed. This can occur through the use of writebacks, for example. The calculation condition is reprocessed when the cube is reprocessed.

If created at session scope, the calculation condition is processed when the statement is issued during the session. As with calculated cells definitions created globally, if the cells are modified, the calculation condition may not be accurate for the calculated cells definition.

If created at query scope, the calculation condition is processed when the query is executed. The cell modification issue applies here, as well, although data latency issues are minimal at best due to the low processing time of MDX query execution.

The calculation formula, on the other hand, is processed whenever an MDX query is issued against the cube involving cells included in the calculated cells definition, no matter the scope.

# Creating and Using User-defined Functions in MDX

Multidimensional Expressions (MDX) supplies a great deal of intrinsic functions, designed to accomplish everything from standard statistical calculation to member traversal in a hierarchy. But, as with any other complex and robust product, there is always the need to extend the functionality of such a product further.

To this end, MDX provides the ability to add user-defined function references to MDX statements. This ability is already in common use in MDX; the functionality supplied by external libraries, such as the Microsoft® Excel and Microsoft Visual Basic® for Applications libraries, takes advantage of this capability.

## Using a User-Defined Function in MDX

Calling a user-defined function in MDX is done in the same manner as calling an intrinsic MDX function. For a function that takes no parameters, the name of the function and an empty pair of parentheses are used, as shown here:

```
MyNewFunction()
```

If the user-defined function takes one or more parameters, then the parameters are supplied, in order, separated by commas. The following example demonstrates a sample user-defined function with three parameters:

```
MyNewFunctionWithParms("Parameter1", 2, 800)
```

## USE LIBRARY Statement

Before employing a user-defined function in an MDX statement, however, the external library that contains the user-defined function must first be loaded into memory. Loading an external library is performed with the USE LIBRARY statement.

All user-defined functions must be associated with a Component Object Model (COM) class in order to be used, usually supplied in the form of a Microsoft ActiveX® dynamic link library (DLL).

If, for example, the user-defined function is part of an ActiveX DLL named MyFunc.dll, located in the C:\Winnt\System path, you can use the USE LIBRARY statement to load it by the library name, as demonstrated here:

```
USE LIBRARY "C:\WINNT\SYSTEM\MyFunc.dll"
```

The USE LIBRARY statement can also load user-defined functions by class name, as each class must be registered in order to work correctly. So, if your example function is located in the example ActiveX DLL and associated with the class "MyFuncClass", the library can be loaded using the following example:

```
USE LIBRARY "MyFunc.MyFuncClass"
```

This method is recommended when referring to libraries that may be in different locations on different servers. As ActiveX DLL components must be registered on server and client machines, referring to the class name ensures that the library is loaded from the correct location, regardless of that location.

Multiple libraries can be loaded at the same time with a single USE LIBRARY statement, by separating the library names or class names with commas, as demonstrated here:

```
USE LIBRARY "C:\WINNT\SYSTEM\MyFunc.dll", "C:\WINNT\SYSTEM\NewFuncs.dll"
```

A USE LIBRARY statement with no parameters unregisters all function libraries except the Microsoft SQL Server™ 2000 Analysis Services function library.

PivotTable® Service supports the USE LIBRARY statement. For more information about the USE LIBRARY statement, see USE LIBRARY Statement.

# DROP LIBRARY Statement

The DROP LIBRARY statement can be used to unload a specific library or to unload all libraries. As with the USE LIBRARY statement, the DROP LIBRARY syntax can accept either the file name or the class name, as demonstrated in the following statement:

```
DROP LIBRARY "MyFunc.MyFuncClass"
```

PivotTable Service supports the DROP LIBRARY statement. For more information about the DROP LIBRARY statement, see DROP LIBRARY Statement.

# Creating User-Defined Functions

User-defined functions can be created in any programming language that supports COM interfaces.

### Parameter and Return Values

A user-defined function can accept any parameter that can be coerced into strings, numbers, or arrays of strings or numbers. User-defined types or object references cannot be used as a parameter. If the parameter data type is explicitly declared as part of the function prototype, such as a double or an integer, Microsoft SQL Server™ 2000 Analysis Services will first coerce values passed into the parameter to the explicitly declared data type. For example, long integer values may be coerced into double precision floating point values if the parameter accepts a Double data type. Date data types are coerced into string representations of a date. PivotTable Service also attempts to coerce strings passed directly into a numeric parameter into numeric values. If coercion fails for any reason, PivotTable Service returns an error condition.

Arrays can also be used as parameters; Analysis Services supports the use of arrays through such functions as **SetToArray**. As with other parameters, if the data type of the array parameter is explicitly declared as part of the function prototype, PivotTable Service will coerce array values into the explicitly declared data type.

If the data type of the array parameter is not explicitly declared or is declared as a variant array, PivotTable Service will also attempt to coerce the elements of the array. However, PivotTable Service handles the coercion of variables in an array a bit differently; the coerced data type is dependent upon the first element of the array, and all other array elements are expected to conform to the same data type. If, for example, the first element in an array is a string, then it is expected that all of the elements in the array are strings, and PivotTable Service will attempt to coerce the other elements into a string data type. If the first element in an array, such as an empty cell, evaluates as empty, then an empty variant is passed to the array parameter instead of a variant array whose first element is empty. If other elements in the array evaluate as empty, the array element is coerced into a zero. You are recommended to explicitly declare the data types of arrays to be used as parameters in user-defined functions.

Similarly, a user-defined function can return any data type that can be coerced into a number, a string, or a variant. The return values are more restrictive; arrays are not allowed. Additionally, PivotTable Service assumes that if a variant is returned, it contains numeric data. If a string, array, or other non-numeric data is returned through a variant, PivotTable Service returns an error condition for the calculation.

Optional parameters in a function are not supported; PivotTable Service requires all parameters in a user-defined function to be populated.

Functions that return void values (for example, subroutines in Visual Basic) can also be used, but are employed with the CALL keyword. If, for example, you wanted to use the function MyVoidFunction() in an MDX statement, the following syntax would be employed:

```
CALL(MyVoidFunction)
```

**Other Considerations**

As with any other MDX function, an external function must be resolved before an MDX session can continue; external functions lock MDX sessions while executing. Unless a specific reason exists to halt an MDX session pending user interaction, it is strongly recommended that any user interaction, such as dialog boxes, be discouraged.

External function libraries can duplicate the function names of the Analysis Services function library or other external function libraries. Normally, if an external function library contains a function with the same name as a function in the Analysis Services function library, the Analysis Services function library takes precedence. If two external function libraries contain a function with the same name, the registration order of the external function libraries determines precedence.

However, if you want to override precedence or call a function from a specific external function library, the external function can be preceded by the program ID, delimited with an exclamation point character, as demonstrated here:

```
«ProgramID»!«FunctionName»(«Argument1», «Argument2», ...)
```

If an external function library supports multiple interfaces, the interface ID can also be used to additionally specify the function, as demonstrated here:

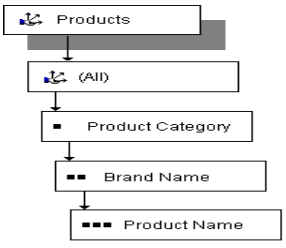
```
«ProgramID»!«InterfaceID»!«FunctionName»(«Argument1», «Argument2», ...)
```

# Using Writebacks

The ability to write information to a write-enabled cube in Multidimensional Expressions (MDX) is called a writeback. Writebacks are supported by two different methods, depending upon the level depth of the member to be changed. Writebacks are supported on server cubes through PivotTable® Service, as described later in this topic. Writebacks to local cubes are not supported.

## Lowest-Level Member Writebacks

A lowest-level member is a member in a dimension associated with the lowest defined level of that dimension. For example, in the following diagram, the Products dimension is defined with three levels (not counting the (All) level).



Any writeback to a member at the [Product Name] level is considered a lowest-level writeback, because there are no defined levels below the [Product Name] level.

A separate table is maintained by Microsoft® SQL Server™ 2000 Analysis Services to store data changed by writebacks, and PivotTable Service propagates the data through the affected aggregate members.

For more information about lowest-level writebacks, see Writing a Value Back to a Cell.

Lowest-level writebacks are most commonly used to modify individual lowest-level member data for speculative analysis. If all of the members of a given aggregate are to be modified, it is often easier to use an aggregate-level member writeback.

## Aggregate-Level Member Writebacks

An aggregate-level member is any member in a dimension whose value depends upon the value of members related to levels below the aggregate level. For example, in the previous diagram, the [Brand Name] level is an aggregate level because the values for its members depend upon aggregations performed on the [Product Name] level. The [Product Category], too, is an aggregate level, because the values for its members depend upon aggregations created from the [Brand Name] level members.

Aggregate-level writebacks are more difficult to process, because in order to modify an aggregate level, all of the members that are used to construct the values for that aggregate level must be modified. You could individually modify each lowest-level member so that the aggregate level represents the desired value, but for cubes representing thousands, tens of thousands, or more values, this is not a recommended option.

Instead, the UPDATE CUBE statement can be employed, using an allocation. Using one of four different allocation formulas, MDX can distribute the desired aggregate value across all of the lowest level members, in effect handling all of the individual lowest-level writebacks for you. Aggregate-level writebacks can be used only when the values are aggregated using the **Sum** aggregate function.

Aggregate-level writebacks are best used when a correction to an aggregate figure is required affecting all lowest-level members of a particular aggregation. Although lowest-level writebacks can also be used to accomplish this task, the aggregate-level writeback is faster and, because it is treated as a single atomic transaction, ensures that security or formula validation issues will not leave a cube in an inconsistent state.

> **Note** Aggregate-level writebacks may produce imprecise results when integer values are allocated, due to incremental rounding variations.

For more information about aggregate-level member writebacks, see UPDATE CUBE Statement.

# Using DRILLTHROUGH to Retrieve Source Data

The DRILLTHROUGH statement is used in Multidimensional Expressions (MDX) to retrieve a rowset from the source data for a cube cell.

In order to execute a DRILLTHROUGH statement on a cube, drillthrough must be enabled for that cube in the **Drillthrough Options** dialog box. The columns that are returned by a DRILLTHROUGH statement are also specified in this dialog box. (If you are programming with Decision Support Objects (DSO), instead of using the dialog box, you can use the **AllowDrillThrough** and **DrillThroughColumns** properties.) For more information, see Specifying Drillthrough Options.

The following syntax construct describes the DRILLTHROUGH statement:

```
<drillthrough> := DRILLTHROUGH [<Max_Rows>] [<First_Rowset>] <MDX select>
   < Max_Rows> := MAXROWS <positive number>
   <First_Rowset> := FIRSTROWSET <positive number>
```

The DRILLTHROUGH statement contains a SELECT clause to identify the cube cell for which source data is retrieved. The SELECT clause is identical to an ordinary MDX SELECT statement except that in the SELECT clause only one member can be specified on each axis. If more than one member is specified on an axis, an error occurs.

The <max_rows> syntax specifies the maximum number of the rows in each returned rowset. If the OLE DB provider that is used to connect to the data source does not support DBPROP_MAXROWS, the <max_rows> setting is ignored.

The <first_rowset> syntax identifies the partition whose rowset is returned first.

The following example demonstrates the use of the DRILLTHROUGH statement:

```
DRILLTHROUGH
    SELECT [Warehouse].[All Warehouses].[Canada].[BC] ON ROWS,
           [Time].[1998].[Q1] ON COLUMNS,
           [Prod].[All Prods].[Drink] ON PAGES,
           [Measures].[Units Shipped] ON SECTIONS
    FROM [My Cube]
```

# Understanding Pass Order and Solve Order

Two of the most powerful and, correspondingly, most difficult concepts in Microsoft® SQL Server™ 2000 Analysis Services, solve order and pass order together determine the manner in which a cube is resolved when queries are processed. This topic assumes that you have a basic understanding of cubes, custom members, calculated members, and custom rollups.

## Pass Order

When a cube is calculated as the result of a Multidimensional Expressions (MDX) query, it goes through at least one stage of computation, and potentially more stages depending on the use of various calculation-related features, such as custom rollup formulas, custom rollup operators, and calculated cells.

Each stage is referred to as a calculation pass, because the Analysis server makes a complete pass of the calculations applicable for that stage. A calculation pass can be referred to by an ordinal position, called the calculation pass number. The count of calculation passes required to fully compute all the cells of a cube is referred to as the calculation pass depth of the cube.

A cube always has one calculation pass, which retrieves data stored for the cube. Because the ordinal position of the pass number begins at zero, this is always referred to as calculation pass 0. All calculated members and custom members are also calculated on pass 0, and every calculation pass thereafter, with formula precedence within this calculation pass established by the solve order of each calculated member. No other actions are allowed for this calculation pass; calculated cells cannot have calculation pass 0 assigned to their calculation pass number.

If a cube has custom rollup formulas or custom rollup operators, a second calculation pass is performed to handle the computations needed to calculate these features. These features are calculated starting at calculation pass 1, and for every calculation pass thereafter as determined by calculated cells definitions. The calculation pass number cannot be changed for custom rollup formulas or custom rollup operators, because they are calculated on each calculation pass, with formula precedence handled by solve order. However, calculated cells can have calculation pass 1 assigned to their calculation pass number, described in more detail later in this topic.

A cube without calculated cells will have at most two calculation passes. Calculated cells, however, can specify the last calculation pass number on which the calculated cells definition is calculated, and how many passes with which the calculated cells definition is used, providing the ability to create cubes that use two or more calculation passes.

Calculated cells can specify the calculation pass number by using the **Calculation Pass Number** property on the **Advanced** tab of Cube Editor, or by using the

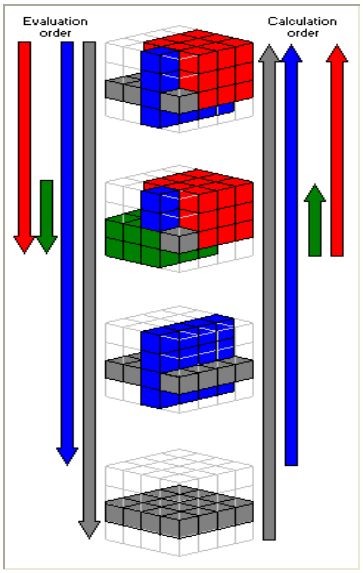
CALCULATION_PASS_NUMBER property in MDX statements. Additionally, recursive calculation is allowed by specifying the number of calculation passes to which the calculation formula is recursively applied to the calculation subcube. This feature, accessed through the **Calculation Pass Depth** property on the **Advanced** tab of Cube Editor or by using the CALCULATION_PASS_DEPTH property in MDX statements, can allow highly complex calculations, such as goal-seeking equations, to be employed in a cube. The calculation pass number determines the calculation pass at which evaluation starts and calculation finishes for a calculated cells definition. The calculation pass depth determines how many calculation passes are required to fully compute a calculated cells definition. Only calculated cells will have a calculation pass number higher than 1.

The number of the inclusive range of calculation passes required to fully compute calculated cells can be defined by the formula CALCULATION_PASS_NUMBER to (CALCULATION_PASS_NUMBER - CALCULATION_PASS_DEPTH) + 1, using the cell properties CALCULATION_PASS_NUMBER and CALCULATION_PASS_DEPTH of the calculated cells definition. In other words, if a calculated cells definition has a CALCULATION_PASS_NUMBER of 4 and a CALCULATION_PASS_DEPTH of 3, the calculated cells definition is evaluated in calculation passes 4, 3, and 2, then calculated in calculation passes 2, 3, and 4.

All calculation passes are retained in memory, to facilitate references to previous pass values in calculation formulas. This ability to refer to previous pass values for a given cell allows for complex calculations, such as speculative analysis and goal-seeking formulas, with an increase in performance.

The number of calculation passes required to fully compute all of the cells of a cube is determined by first evaluating all of the custom members, custom rollups, calculated members, and calculated cells. Evaluation is done from highest calculation pass to lowest calculation pass, determined by the CALCULATION_PASS_NUMBER property, in order to accurately determine formula precedence across calculation passes. The order is then reversed when calculating the calculation passes, by calculating from lowest to highest. Essentially, each calculation pass is treated as a nested calculation, with the lowest calculation pass being the most nested.

The following table illustrates the effects of calculation pass number and calculation pass depth on a sample cube. The sample cube contains four calculations:

| Pass diagram | Pass description |
|---|---|
|  | Calculation Pass 3<br>Because the CALCULATION_PASS_DEPTH of the calculated cells definition shaded with red is 2, the cells are recursively calculated again, using the values derived from the previous calculation pass. The calculated member and custom rollup formula are also calculated again on this pass.<br><br>Calculation Pass 2<br>The calculations for both calculated cells definitions start here, based on the evaluation of CALCULATION_PASS_NUMBER and CALCULATION_PASS_DEPTH. The calculated member and custom rollup formula are also calculated again on this pass.<br><br>Calculation Pass 1<br>All custom rollup formulas and custom rollup operators start calculation on pass 1.<br><br>Calculation Pass 0<br>Data is retrieved from sources. Calculated and custom members are calculated. No other calculations can be performed at this point. |

- A calculated member, shaded in dark gray, with a SOLVE_ORDER of 1.
- A custom rollup formula, shaded in dark blue, with a SOLVE_ORDER of 2.

- A calculated cells definition, shaded in green, with a CALCULATION_PASS_NUMBER of 2, a CALCULATION_PASS_DEPTH of 1, and a SOLVE_ORDER of 1.
- A calculated cells definition, shaded in red, with a CALCULATION_PASS_NUMBER of 3, a CALCULATION_PASS_DEPTH of 2, and a SOLVE_ORDER of 2.

Color-coded arrows show evaluation and calculation order for the various calculations in the diagram shown in the previous table. In cubes with multiple calculations, some of the calculations can overlap. When this occurs, the solve order of the overlapping calculations is used to resolve formula precedence, but only within a given pass. If a solve order is specified for a calculated cells definition that encompasses multiple passes (that is, the CALCULATION_PASS_DEPTH is greater than 1), the solve order is applied to each pass to resolve formula precedence. Because solve order is applied on each calculation pass, overlapping calculated cells definitions can generate different values for cells that may be involved in custom rollup or calculated member resolution. A more detailed discussion of solve order is given later in this topic.

Recursive calculations and goal-seeking calculations can make use of values obtained in previous passes through the use of the **CalculationPassValue** and **CalculationCurrentPass** functions in MDX. The **CalculationCurrentPass** function provides the current calculation pass number, and the **CalculationPassValue**, given an MDX expression and a calculation pass number, evaluates the MDX expression within the specified calculation pass number and returns the result.

### Solve Order

Within a single pass, solve order determines two things: the order in which dimensions, members, calculated members, custom rollups, and calculated cells are evaluated, and the order in which they are calculated. The member with the highest solve order is evaluated first, but calculated last. This is similar in behavior to any other nested operation: the outermost operation cannot complete until the innermost operation is completed, but the outermost operation is evaluated first in order to determine that the innermost operation must be completed before the outermost can be completed. The lower the solve order, the more nested the member in terms of evaluation and calculation, with the member having the highest solve order occupying the outermost position.

In cubes with dimensions that contain custom members, custom rollup formulas, calculated members, and calculated cells, the solve order determines the order in which various calculations are evaluated. The highest solve order is always evaluated first, then the next highest, and so on.

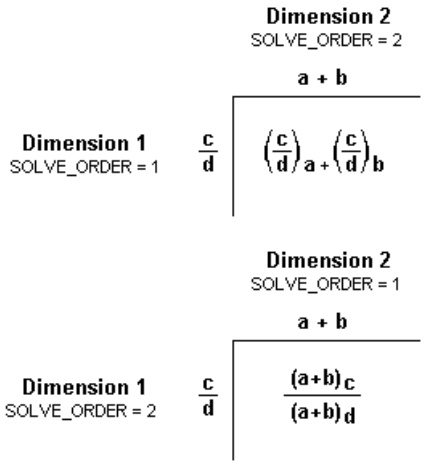
The order in which they are calculated, however, is reversed. The lowest solve order is calculated first, then the next lowest, and so on. Solve order essentially nests formulas, and as with any nested formula, the outermost formulas are evaluated first, but calculated last. The innermost formulas are evaluated last, but calculated first, because the outermost formulas may depend on the values produced by the innermost formulas for their calculation.

Although measures are usually treated as another dimension, they are always evaluated last and calculated first for solve order purposes. In other words, measures are always treated as having the lowest possible solve order.

In Cube Editor, the solve order for calculated members and calculated cells can be changed by altering the **Solve Order** property in the **Advanced** tab of the properties pane. The solve order for dimensions can be changed by reordering the positions of the dimensions within the tree pane. For more information about using Cube Editor, see Cube Editor - Schema View.

In MDX the **SOLVE_ORDER** member property can be used when creating or changing calculated members and calculated cells

Solve order directly affects the results generated by the calculation of dimensions and members in this fashion. The following diagram describes the behavior of two dimensions, each with a calculated member, that intersect at a cell. Two examples are presented in the following diagram, with different solve orders.




In the first example, Dimension 2 has a higher solve order. So, the intersection is evaluated using the formula for Dimension 2. However, in order to provide data for this formula, the calculated member in Dimension 1 must be evaluated and calculated. So, the formula for Dimension 1 is calculated to provide the values needed to calculate the formula for Dimension 2. Then, the formula for Dimension 2 is calculated and the result is placed in the cell.

In the second example, Dimension 1 has a higher solve order. The cell at the intersection is evaluated using the formula for Dimension 1. As Dimension 2 has a lower solve order, it is evaluated next, then calculated first and the values provided to the formula for Dimension 1. Then, the formula for Dimension 1 is calculated and the result is placed in the cell.

To further demonstrate the potential complexities of solve order, a series of example MDX queries is presented here.

For the first example, you are interested in seeing the difference in income and expenses for each half of the year. You would then construct a simple MDX query similar to the following example:

```
WITH
MEMBER [Time].[Year Difference] AS '[Time].[2nd half] - [Time].[1st half]'
SELECT
   {[Money].[Income], [Money].[Expenses]} ON COLUMNS,
   {[Time].[1st half],
    [Time].[2nd half],
    [Time].[Year Difference]} ON ROWS
FROM TestCube
```

This MDX query would produce a result set similar to the following table, with the calculated member shaded.

| | Income | Expenses |
|---|---|---|
| **1st half** | 5000 | 4200 |
| **2nd half** | 8000 | 7000 |
| **Year Difference** | 3000 | 2800 |

For this query, solve order is not an issue, assuming the cube does not use any calculated members, because there is only one calculated member in the query.

Now, for the second example, you are interested in seeing the percentage of net income after expenses for each half of the year, using the following MDX query:

```
WITH
MEMBER [Money].[NetInc] AS '([Money].[Income] - [Money].[Expenses]) / [Money].[Income]'
SELECT
   {[Money].[Income], [Money].[Expenses], [Money].[NetInc]} ON COLUMNS,
   {[Time].[1st half], [Time].[2nd half] } ON ROWS
FROM TestCube
```

This MDX query would produce a slightly different result set, similar to the following table, with the calculated member shaded.

| | Income | Expenses | Net Income |
|---|---|---|---|
| **1st half** | 5000 | 4200 | 0.16 |
| **2nd half** | 8000 | 7000 | 0.125 |

This MDX query, like the previous one, does not have any solve order complications, because it also has only a single calculated member. Notice the placement of the calculated member in the result dataset of this example, as well as in the previous example. The first MDX query example uses a calculated member as part of the ROWS axis dimension, but this query example uses a calculated member as part of the COLUMNS axis dimension. This placement becomes important in the next example, which combines the two calculated members in a single MDX query.

Finally, you decide you want to combine both of the previous examples into a single MDX query. In this case, solve order becomes important. Take, for example, the first attempt at this combination in the following MDX query:

```
WITH
  MEMBER [Time].[Year Difference] AS '[Time].[2nd half] - [Time].[1st half]',
           SOLVE_ORDER = 1
  MEMBER [Money].[NetInc] AS '([Money].[Inc - [Money].[Expenses]) / [Money].[Inc]',
           SOLVE_ORDER = 2
SELECT
   {[Money].[Income], [Money].[Expenses], [Money].[Net Income]} ON COLUMNS,
   {[Time].[1st half], [Time].[2nd half], [Time].[Year Difference]} ON ROWS
FROM TestCube
```

The two calculated members, Year Difference and Net Income, intersect at a single cell in the result dataset of the MDX query example. The only way to determine how this cell will be evaluated is by the solve order. The formulas used to construct this cell will produce different results depending upon the solve order of the two calculated members.

The SOLVE_ORDER keyword is be used to specify the solve order of calculated members in an MDX query or the CREATE MEMBER command. If the solve order is not specified, it defaults to zero. In that case, the order of the dimensions in the cube whose context is specified in the MDX query is used to determine the solve order, following the rules listed earlier in this topic. This also applies to calculated members in different dimensions that are assigned the same SOLVE_ORDER value.

The integer values used with the SOLVE_ORDER keyword are relative; the value simply tells MDX to calculate a member based on values derived from calculating members with a higher value. If a calculated member is defined without the SOLVE_ORDER keyword, its default value is zero. The specified values do not need to start at zero, nor do they need to be consecutive.

In the MDX query example, Net Income has the highest solve order, so the cell in question is evaluated using the Net Income formula. But the Net Income formula is calculated last; in order to calculate Net Income, the next calculated member, Year Difference, must be calculated first so that Net Income can use the results of that calculated member to perform its own calculation.

The results of this nested calculation can be viewed in the following table.

| | Income | Expenses | Net Income |
|---|---|---|---|
| **1st half** | 5000 | 4200 | 0.16 |
| **2nd half** | 8000 | 7000 | 0.125 |
| **Year Difference** | 3000 | 2800 | 0.066 |

As you can see, the result in the shared cell is based on the formula for Net Income; in other words, it was calculated with the Year Difference data, producing the following formula (the result is rounded for clarity):

((8000 - 5000) - (7000 - 4200)) / (8000 - 5000) = 0.066

or

(3000 - 2800) / 3000 = 0.066

The result in the shared cell, however, is calculated differently if the solve orders for the calculated members in the MDX query are switched, as demonstrated here:

```
WITH
MEMBER [Time].[Year Difference] AS '[Time].[2nd half] - [Time].[1st half]'
   SOLVE_ORDER = 2
MEMBER [Money].[NetInc] AS '([Money].[Income] - [Money].[Expenses]) / [Money].[Income]',
   SOLVE_ORDER = 1
SELECT
   {[Money].[Income], [Money].[Expenses], [Money].[Net Income]} ON COLUMNS,
   {[Time].[1st half], [Time].[2nd half], [Time].[Year Difference]} ON ROWS
FROM TestCube
```

As the order of the calculated members has been switched, the Year Difference formula is used to evaluate the cell. The Net Income calculated member is resolved first, and then the Year Difference calculated member is resolved, producing a strikingly different result as shown in the following table.

| | Income | Expenses | Net Income |
|---|---|---|---|
| **1st half** | 5000 | 4200 | 0.16 |
| **2nd half** | 8000 | 7000 | 0.125 |
| **Year Difference** | 3000 | 2800 | -0.035 |

Because it uses the Year Difference formula with the Net Income data, the formula for the shared cell resembles the following calculation:

((8000 - 7000) / 8000) - ((5000 - 4200) / 5000) = -0.035

Or

0.125 - 0.16 = -0.035

### Changing Solve Order Values

Solve order values can range from -8181 to 65535. It is highly recommended that you use only positive integers when setting solve order values. Certain calculations reside at specific solve orders, as listed in the following table. The solve order for a pass can become unpredictable if these values are used by other calculations.

| Calculation | Solve order |
|---|---|
| Calculated cell formula "dirtiness" | -6143 |
| Custom rollup formulas (if not otherwise specified) | -5119 |
| Virtual dimensions created with earlier versions of Analysis Services | -4097 |
| Visual totals calculation | -4096 |
| All other calculations (if not otherwise specified) | 0 |

For example, changing the solve order for a calculated cells definition below the default custom rollup formula value of -5119 causes the calculated cells definition to be calculated before the custom rollup formulas; this can produce incorrect results.

In the case of multiple calculations having the same solve order, the following formula precedence is used:

1. Calculated cells
2. Custom rollup formulas
3. Custom and calculated members

4. All other calculations

Calculated cells take precedence over all other calculations in the case of solve order conflict. If multiple calculations occur within the same category, the declaration order of the calculation is used. For example, if two calculated cells definitions have the same solve order for the same calculation pass, the declaration order determines which is evaluated first.

### Additional Considerations

The combination of pass order and solve order can be a very complex issue to deal with, especially in cubes with a high number of dimensions involving calculated member, custom rollup formulas, or calculated cells. When MDX evaluates an MDX query, the solve order values for everything involved within a given pass, including the dimensions of the cube specified in the MDX query, are taken into account.

When a query with calculated members is executed against a cube with calculated members, for example, the solve orders for both the query and the cube are evaluated as if the query were part of the cube; it is executed within the context of a cube. Because it can be difficult to review the solve order of the dimensions on a cube, it can be challenging to ensure that the solve order for calculated members in a complex MDX query are correctly handled within the context of a cube.

Also, as the solve order for the dimensions on a cube can be changed from Cube Editor, MDX queries can be affected; a once-working MDX query can return unexpected results because the solve order of the cube on which context the MDX query executes is changed.

# Effective MDX

This topic provides information on more effective uses of Multidimensional Expressions (MDX) functions in various scenarios. The topics covered are listed in the following table.

# Comments in MDX

Statements in Multidimensional Expressions (MDX) can contain user-readable comments that are ignored when the commands are processed. The three different character sets that indicate comments are outlined in the following table.

| Characters | Description |
|---|---|
| // | C++-style forward slashes. All text between the forward slashes and the end of the same line is ignored. |
| -- | SQL-style hyphens. All text between the dashes and the end of the same line is ignored. |
| /*...*/ | C-style forward slash and asterisk pairs. All text between the opening forward slash and asterisk and the closing asterisk and backward slash is ignored. This type of comment can span multiple lines. |

The following example shows the use of comments in an MDX command:

```
/* Using this query to view
   info about units shipped
   and units ordered  */

WITH MEMBER [Measures].[ShippingPercent] AS
       '-- Returns [Units Shipped] over [Units Ordered] as a percent value
   Measures.[Units Shipped] / Measures.[Units Ordered]',
       FORMAT_STRING = 'Percent'

SELECT
   {[Measures].[Units Shipped],
    [Measures].[Units Ordered],
    [Measures].[ShippingPercent] } ON COLUMNS,
    // The next command specifies nonempty members only
   NON EMPTY [Store].[Store Name].Members ON ROWS
```

```
FROM Warehouse   -- Pulled from the Warehouse cube
```

Comments are recommended in complex or difficult to understand MDX queries, because they add information without incurring performance penalties.

# Working with Empty Cells

Empty cells occur in Multidimensional Expressions (MDX) statements when data for the intersection of two or more dimensions does not exist. For example, the following MDX query example produces many empty cells:

```
SELECT
   {[Store].[Store Name].Members} ON COLUMNS,
   {[Product].[Excellent Diet Cola]} ON ROWS
FROM Sales
WHERE [Measures].[Unit Sales]
```

The product, Excellent Diet Cola, is not sold in all stores. For the stores that sell the product, the Unit Sales measure will contain a numeric value. For the stores that do not sell the product, however, an empty cell will be displayed.

Empty cells affect the evaluation of value expressions and search conditions. To understand why this is so, note that a value expression is composed of value expression primaries. One of the value expression primaries is `<tuple>[.VALUE]`, which returns the value of a cell in the cube (some of whose coordinates are specified explicitly by `<tuple>`, and others that are available implicitly from the context of the MDX statement). This cell can be an empty cell. Empty cells affect expression evaluation in the following three cases:

- With numeric value expressions. In a numeric value expression, this value can be added, subtracted, multiplied, or divided by other values. It can also appear as the parameter of any function that has a <numeric_value_expression> argument.
- With string value expressions. In a string value expression, this value can be concatenated to another string.
- With search conditions composed of Boolean primaries. A Boolean primary is of the following form:

```
<boolean_primary> ::= <value_expression> <comp_op> <value_expression>
```

A value expression will be made up of the value expression primary, and this will lead to the first two cases described listed earlier.

## Empty Cell Evaluation

MDX specifically identifies an empty cell by defining a special empty cell value that is present in an empty cell. The empty cell value is evaluated as follows:

- The function **IsEmpty**(<value_expression>) returns TRUE if <value_expression> is the empty cell value. Otherwise it returns FALSE.
- When the empty cell value is an operand for any of the numeric operators (+, -, *, /), it behaves like the number zero.
- When the empty cell value is an operand for the string concatenation operator (||), it behaves like the empty string.
- When the empty cell value is an operand for any of the comparison operators (=. <>, >=, <=, >, <), it behaves like the number zero or the empty string, depending on whether the data type of the other operand is numeric or string, respectively.
- When collating numeric values, the empty cell value collates in the same place as zero. Between the empty cell value and zero, empty collates before zero.

- When collating string values, the empty cell value collates in the same place as the empty string. Between the empty cell value and the empty string, the empty cell value collates before an empty string.

Empty cells can be handled in a variety of ways; the easiest is to simply remove them from consideration. However, because this is not always practical in MDX, functions have been provided to deal with empty cells.

# NON EMPTY Keyword

The easiest way to remove empty cells from consideration is to use the NON EMPTY keyword in an MDX query. The following example is the same MDX query example discussed earlier in this topic, but using the NON EMPTY keyword.

```
SELECT
   NON EMPTY {[Store].[Store Name].Members} ON COLUMNS,
   {[Product].[Excellent Diet Cola]} ON ROWS
FROM Sales
WHERE [Measures].[Unit Sales]
```

All of the stores in the first axis dimension that do not have values for the unit sales of the product are excluded from the result dataset. The empty tuples are screened out of the result dataset of the MDX query.

It is important to note that this function screens out empty tuples, not individual empty cells. Because of this, empty cells can appear in a result dataset even when the NON EMPTY keyword is used. For example, suppose you want to examine the unit sales for two different products in 1997 for each store. The following MDX query example uses the NON EMPTY keyword to screen out empty tuples:

```
SELECT
   NON EMPTY CROSSJOIN ({[Product].[Excellent Diet Cola],
                         [Product].[Fabulous Diet Cola]},
                        {[Time].[1997]}) ON COLUMNS,
   NON EMPTY {[Store].[Store Name].Members} ON ROWS
FROM Sales
WHERE [Measures].[Unit Sales]
```

However, the result dataset resembles the following table.

| | Excellent Diet Soda | Fabulous Diet Soda |
|---|---|---|
| | 1997 | 1997 |
| Store 6 | 20.00 | 11.00 |
| Store 7 | 25.00 | 6.00 |
| Store 24 | 11.00 | 19.00 |
| Store 11 | 36.00 | 32.00 |
| Store 13 | 25.00 | 22.00 |
| Store 2 | 2.00 | |
| Store 3 | 23.00 | 16.00 |
| Store 15 | 14.00 | 17.00 |
| Store 16 | | 13.00 |
| Store 17 | 22.00 | 12.00 |
| Store 22 | 2.00 | |
| Store 23 | 4.00 | 5.00 |

The result dataset still shows three empty cells, despite the presence of the NON EMPTY keyword. The tuples created by the MDX query may contain empty cells, but the tuples themselves are not empty. For example, in the preceding result dataset, though Store 22 did

not sell any of the Fabulous Diet Soda product in 1997, it did sell some of the Excellent Diet Soda product in 1997. So, the tuple created by the CROSSJOIN command does contain a member that does not evaluate to an empty cell; therefore the tuple is not considered empty and is not screened out.

For more information about the use of NON EMPTY in MDX SELECT statements, see SELECT Statement.

# CoalesceEmpty Function

This MDX function returns the first nonempty value in a list of values. It is useful when you want to replace empty cell values with another numeric or string expression.

The **CoalesceEmpty** function allows you to evaluate a series of value expressions from left to right. The first value expression in the series that does not evaluate to the empty cell value is returned. For example, the following MDX query modifies the previous MDX query example to replace all of the empty cell values in the Unit Sales measure with zero:

```
WITH MEMBER [Measures].[NonEmptyUnitSales] AS
   'CoalesceEmpty(Measures.[Unit Sales], 0)'

SELECT
   NON EMPTY CROSSJOIN ({[Prod].[Excellent Diet Cola], [Prod].[Fabulous Diet Cola]},
                        {[Time].[1997]}) ON COLUMNS,
   NON EMPTY {[Store].[Store Name].Members} ON ROWS
FROM Sales
WHERE [Measures].[NonEmptyUnitSales]
```

The following table demonstrates the result dataset returned by the MDX query example.

| | Excellent Diet Soda | Fabulous Diet Soda |
|---|---|---|
| | 1997 | 1997 |
| **Store 19** | 0 | 0 |
| **Store 20** | 0 | 0 |
| **Store 9** | 0 | 0 |
| **Store 21** | 0 | 0 |
| **Store 1** | 0 | 0 |
| **Store 5** | 0 | 0 |
| **Store 10** | 0 | 0 |
| **Store 8** | 0 | 0 |
| **Store 4** | 0 | 0 |
| **Store 12** | 0 | 0 |
| **Store 18** | 0 | 0 |
| **HQ** | 0 | 0 |
| **Store 6** | 20.00 | 11.00 |
| **Store 7** | 25.00 | 6.00 |
| **Store 24** | 11.00 | 19.00 |
| **Store 11** | 36.00 | 32.00 |
| **Store 13** | 25.00 | 22.00 |
| **Store 2** | 2.00 | 0 |
| **Store 3** | 23.00 | 16.00 |
| **Store 15** | 14.00 | 17.00 |

| | | |
|---|---|---|
| **Store 16** | 0 | 13.00 |
| **Store 17** | 22.00 | 12.00 |
| **Store 22** | 2.00 | 0 |
| **Store 23** | 4.00 | 5.00 |

The values of the calculated member NonEmptyUnitSales were determined by the **CoalesceEmpty** function. If the Unit Sales value evaluated to a nonempty cell, the first value in the **CoalesceEmpty** statement was returned. If the [Unit Sales] value evaluated to an empty cell value, the second value in the **CoalesceEmpty** statement was returned. Because the **CoalesceEmpty** function replaced all of the empty cell values with zero, the NON EMPTY keyword has nothing to screen out, so all of the tuples in the query were valid and were presented in the result dataset.

## Other Functions

The way that other functions (especially calculation functions) deal with empty cells depends on the capabilities and options that are available to those functions. Functions such as **Count** and **Avg** evaluate a count of cells, but whether or not to evaluate an empty cell by this type of function should be given careful thought. In practice, it is sometimes preferable to count the number of empty cells. For example, when the number of sales representatives is counted as part of a performance evaluation query, all sales representatives should be included in the count whether or not they sold anything. In this case, each no-sale results in an empty cell. However, there are other situations in which empty cells should not be counted, such as when getting the average of sales over a certain domain. In this case, counting the no-sale cells would inaccurately decrease the average.

Some MDX functions in which empty cells may change the outcome allow for the inclusion or exclusion of empty cells as part of their calculation. **Count**, for example, supports the use of INCLUDEEMPTY and EXCLUDEEMPTY flags to handle the inclusion or exclusion of empty cells, respectively, while counting.

# Creating a Cell Within the Context of a Cube

For certain applications, you may want to return data for a single cell within a cube. For example, executives might have a decision support application written in Microsoft® Excel that uses data from a multidimensional data store. Suppose that when the application starts each day, the executives want to view, at the top of the application's main window, the quarter-to-date worldwide sales for the current year across all products and customers.

The solution is to create a dataset for which all dimensions are slicer dimensions. The Multidimensional Expressions (MDX) statement for doing this takes the following form:

```
SELECT FROM cube_name WHERE slicer_specification
```

This results in a dataset with one cell. Because no axis dimensions are specified, the slicer specification focuses on the desired point in the entire cube.

In this case, where there are no axes and hence only one cell, the following conditions apply:

- The IMDDataset::GetAxisInfo method returns 0 for **pcAxes* and a null pointer in *prgAxisInfo.
- The axis rowsets for all axes will be empty, except for the axis MDAXIS_SLICERS. The axis for the slicer dimension will contain information on the slicer conditions that created the single cell.
- The single cell can be addressed by using the cell ordinal 0.

# Working with the RollupChildren Function

The use of the **RollupChildren** function in Multidimensional Expressions (MDX) statements is simple to explain, but the impact of this function on MDX queries can be wide-ranging.

The **RollupChildren** function rolls up the children of a member, applying a different unary operator to each child, and returns the value of this rollup as a number. The unary operator used can be supplied by a member property associated with the child member, or it can be a string expression provided directly to the function.

The impact of the **RollupChildren** function occurs in MDX queries designed to perform selective analysis on existing cube data. For example, the following table contains a list of child members for the Net Sales parent member, with their unary operators (represented by the UNARY_OPERATOR member property) shown in parentheses.

| Parent member | Child member |
|---|---|
| Net Sales | Domestic Sales (+)<br>Domestic Returns (-)<br>Foreign Sales (+)<br>Foreign Returns (-) |

The Net Sales parent member currently provides a total of net sales minus the gross domestic and foreign sales values, with the domestic and foreign returns subtracted as part of the rollup.

Now, if you want to provide a quick and easy forecast of domestic and foreign gross sales plus 10%, ignoring the domestic and foreign returns, there are two ways to perform this action using the **RollupChildren** function.

## Custom Member Properties

If this is to be a commonly performed operation, one method is to create a member property that stores the operator to be used for each child for a given function. For example, a member property called SALES_OPERATOR is created, and the following unary operators are assigned to it, as shown in the following table.

| Parent member | Child member |
|---|---|
| Net Sales | Domestic Sales (+)<br>Domestic Returns (~)<br>Foreign Sales (+)<br>Foreign Returns (~) |

With this new member property, the following MDX statement performs the gross sales estimate operation quickly and efficiently:

```
RollupChildren([Net Sales], [Net Sales].CurrentMember.Properties("SALES_OPERATOR")) * 1.1
```

When the function is called, the value of each child is applied to a total using the operator stored in the member property. The following table displays valid unary operators and describes the expected result.

| Operator | Result |
|---|---|
| + | total = total + current child |
| - | total = total - current child |
| * | total = total * current child |
| / | total = total / current child |
| ~ | Child is not used in the rollup. Its value is ignored. |

The tilde (~) unary operator indicates that this member is to be ignored when generating rollups totals. The members for domestic and foreign returns are ignored and the rollup total returned by the **RollupChildren** function is multiplied by 1.1.

## IIf Function

However, if the example operation is not commonplace or if it applies only to one MDX query, then the **IIf** function can be used with the **RollupChildren** function to provide the same result. The following MDX query provides the same result as the earlier MDX example, but does so without resorting to the use of a custom member property:

```
RollupChildren([Net Sales],
               IIf([Net Sales].CurrentMember.Properties("UNARY_OPERATOR") = "-", "~",
                   [Net Sales].CurrentMember.Properties("UNARY_OPERATOR"))) * 1.1
```

The MDX statement checks the unary operator of the child member; if it is used for subtraction (as with the domestic and foreign returns members), the tilde (~) unary operator is substituted by the **IIf** function. Otherwise, the unary operator of the child member is used. Finally, the returned rollup total is then multiplied by 1.1 to provide the domestic and foreign gross sales forecast value.

# WHERE Clause Overrides

Each individual set, member, tuple, or numeric function in a Multidimensional Expressions (MDX) statement always executes in the larger context of the entire statement. For example, consider the FILTER function in the following expression:

```
SELECT FILTER(SalesRep.MEMBERS, [1996].VALUE > 500) ON COLUMNS,
       Quarters.MEMBERS ON ROWS
FROM SalesCube
WHERE ([Geography].[All], [Products].[All], [1996], Sales)
```

The second argument of FILTER, "[1996].VALUE", does not contain enough information by itself. Six coordinates are needed, one from each of the six dimensions, to determine VALUE. The argument contains only one coordinate, from the Years dimension. In such a case, the other coordinates are obtained by looking at the following, in order:

1. The rest of the axis specification. This yields (in the preceding example) the coordinate of the SalesRep dimension because the FILTER function iterates through each member of the SalesRep dimension.
2. The slicer condition (WHERE clause) and the coordinates for the slicer dimension. This yields the coordinates for the Geography, Products, and Measures dimensions as (respectively) Geography.[All], Products.[All], and Measures.Sales.
3. The default member for dimensions that appear neither on the axis nor on the slicer. Thus the default members are picked for the Quarters dimension.

A special case arises when a coordinate is specified both in the WHERE clause and within the expression. For example, suppose an application calls for a dataset that, on the COLUMNS axis, contains 1996 budgeted sales for all the states in the United States that had more than 500 units of ActualSales in 1995 and that, on the ROWS axis, contains the Quarters. The following statement can create this dataset:

```
SELECT FILTER({USA.CHILDREN}, ([1995], ActualSales) > 500) ON COLUMNS,
       Quarters.MEMBERS ON ROWS
FROM SalesCube
WHERE ([1996], BudgetedSales, [Products].[All], [SalesRep].[All])
```

As the FILTER function is evaluated for each state in the United States, it already has the coordinates ([1996], BudgetedSales) from the WHERE clause. However, it receives the coordinates ([1995], ActualSales) from the FILTER function. To avoid potential conflict, the argument of the FILTER function takes precedence. In general, any coordinates obtained from the WHERE clause are overridden by coordinates that are specified within an axis specification.

# MDX Functions in Analysis Services

Microsoft® SQL Server™ 2000 Analysis Services provides for the use of functions in Multidimensional Expressions (MDX) syntax. Functions can be used in any valid MDX statement, and are often used in queries, calculated members, and custom rollup definitions. There are three types of functions in MDX, and each is described in a separate topic.

# MDX Function Reference

This topic provides information about the Multidimensional Expressions (MDX) functions included with Microsoft® SQL Server™ 2000 Analysis Services. You can use the MDX Function List to find functions by their category of return value, or you can select a function by name from the alphabetical list in the table of contents.

## MDX Syntax Conventions

The diagrams for Multidimensional Expressions (MDX) syntax in the MDX Function Reference use these conventions.

| Convention | Usage |
|---|---|
| [ ] (brackets) | Optional syntax items. Do not type the brackets. |
| \| (vertical bar) | Separating syntax items within brackets or braces. You can choose only one of the items. |
| « » (guillemets) | User-supplied parameters of MDX syntax. Do not type the guillemets. |
| [,...] | Indicating that the preceding item can be repeated any number of times. The items are separated by commas. |

## MDX Function List

This topic contains lists of the Multidimensional Expressions (MDX) functions in Microsoft® SQL Server™ 2000 Analysis Services. You can use these lists to find functions by their category of return value, or you can select a function by name from the alphabetical list in the table of contents.

### Samples Used in Examples

For many expression examples in the following topics, SampleSet is defined as:

```
{USA, Buffalo, France, NYC, London, California, LA, Nice, UK, Paris}
```

The following table lists sales data for each member of the set.

| Location | 1995 sales | 1996 sales |
|---|---|---|
| UK | 1900 | 1700 |
| London | 250 | 300 |
| France | 2500 | 2500 |
| Paris | 365 | 250 |
| Nice | 27 | 100 |
| USA | 5000 | 6500 |
| Boston | 900 | 1100 |
| Buffalo | 300 | 200 |
| California | 2000 | 3500 |
| Los Angeles | 500 | 900 |

### MDX Function Groups

The following tables list the MDX functions grouped by their return value categories. You can use the links in the tables to jump to the function reference topics.

#### Array Functions

| Function | Description |
|---|---|
| SetToArray | Converts one or more sets to an array for use in a user-defined function. |

### Dimension, Hierarchy, and Level Functions

#### Dimension Functions

| Function | Description |
|---|---|
| Dimension | Returns the dimension that contains a specified hierarchy, level, or member. |
| Dimensions | Returns the dimension whose zero-based position within the cube is specified by a numeric expression or whose name is specified by a string. |

#### Hierarchy Functions

| Function | Description |
|---|---|
| Hierarchy | Returns the hierarchy of a level or member. |

#### Level Functions

| Function | Description |
|---|---|
| Level | Returns the level of a member. |
| Levels | Returns the level whose position in a dimension is specified by a numeric expression or whose name is specified by a string expression. |

#### Logical Functions

| Function | Description |
|---|---|
| Is | Returns True if two compared objects are equivalent, False otherwise. |
| IsAncestor | Determines whether a specified member is an ancestor of another specified member. |
| IsEmpty | Determines whether an expression evaluates to the empty cell value. |
| IsGeneration | Determines whether a specified member is in a specified generation. |
| IsLeaf | Determines whether a specified member is a leaf member. |
| IsSibling | Determines whether a specified member is a sibling of another specified member. |

#### Member Functions

| Function | Description |
|---|---|
| Ancestor | Returns the ancestor of a member at a specified level or at a specified distance from the member. |
| ClosingPeriod | Returns the last sibling among the descendants of a member at a level. |
| Cousin | Returns the member with the same relative position under a member as the member specified. |
| CurrentMember | Returns the current member along a dimension during an iteration. |
| DataMember | Returns the system-generated data member associated with a nonleaf member. |

| | |
|---|---|
| DefaultMember | Returns the default member of a dimension or hierarchy. |
| FirstChild | Returns the first child of a member. |
| FirstSibling | Returns the first child of the parent of a member. |
| Ignore | Reserved. |
| Item | Returns a member from a tuple. |
| Lag | Returns a member prior to the specified member along the member's dimension. |
| LastChild | Returns the last child of a member. |
| LastSibling | Returns the last child of the parent of a member. |
| Lead | Returns a member further along the specified member's dimension. |
| LinkMember | Returns a hierarchized member. |
| Members | Returns the member whose name is specified by a string expression. |
| NextMember | Returns the next member in the level that contains a specified member. |
| OpeningPeriod | Returns the first sibling among the descendants of a member at a level. |
| ParallelPeriod | Returns a member from a prior period in the same relative position as a specified member. |
| Parent | Returns the parent of a member. |
| PrevMember | Returns the previous member in the level that contains a specified member. |
| StrToMember | Returns a member based on a string expression. |
| ValidMeasure | Returns a valid measure in a virtual cube by forcing inapplicable dimensions to their top level. |

**Numeric Functions**

| Function | Description |
|---|---|
| Aggregate | Returns a calculated value using the appropriate aggregate function, based on the context of the query. |
| Avg | Returns the average value of a numeric expression evaluated over a set. |
| CalculationCurrentPass | Returns the current calculation pass of a cube for the current query context. |
| CalculationPassValue | Returns the value of an MDX expression evaluated over a specified calculation pass of the current cube. |
| CoalesceEmpty | Coalesces an empty cell value to a number or a string. |
| Correlation | Returns the correlation of two series evaluated over a set. |
| Count | Returns the number of dimensions in a cube, the number of levels in a dimension, the number of cells in a set, or the number of dimensions in a tuple. |
| Covariance | Returns the population covariance of two series evaluated over a set, using the biased population formula. |
| CovarianceN | Returns the sample covariance of two series evaluated over a set, using the unbiased population formula. |
| DistinctCount | Returns the count of tuples in a set, excluding duplicate tuples. |
| IIf | Returns one of two numeric or string values determined by a logical test. |
| LinRegIntercept | Calculates the linear regression of a set and returns the value of b in the regression line y = ax + b. |
| LinRegPoint | Calculates the linear regression of a set and returns the value of y in the regression line y = ax + b. |
| LinRegR2 | Calculates the linear regression of a set and returns $R^2$ (the coefficient of determination). |

| | |
|---|---|
| LinRegSlope | Calculates the linear regression of a set and returns the value of a in the regression line y = ax + b. |
| LinRegVariance | Calculates the linear regression of a set and returns the variance associated with the regression line y = ax + b. |
| LookupCube | Returns the value of an MDX expression evaluated over another specified cube in the same database. |
| Max | Returns the maximum value of a numeric expression evaluated over a set. |
| Median | Returns the median value of a numeric expression evaluated over a set. |
| Min | Returns the minimum value of a numeric expression evaluated over a set. |
| Ordinal | Returns the zero-based ordinal value associated with a level. |
| Predict | Evaluates the string expression within the data mining model specified within the current coordinates. |
| Rank | Returns the one-based rank of a tuple in a set. |
| RollupChildren | Scans the children of the member parameter and applies the string expression operator to their evaluated value. |
| Stddev | Alias for **Stdev**. |
| StddevP | Alias for **StdevP**. |
| Stdev | Returns the sample standard deviation of a numeric expression evaluated over a set, using the unbiased population formula. |
| StdevP | Returns the population standard deviation of a numeric expression evaluated over a set, using the biased population formula. |
| StrToValue | Returns a value based on a string expression. |
| Sum | Returns the sum of a numeric expression evaluated over a set. |
| Value | Returns the value of a measure. |
| Var | Returns the sample variance of a numeric expression evaluated over a set, using the unbiased population formula. |
| Variance | Alias for **Var**. |
| VarianceP | Alias for **VarP**. |
| VarP | Returns the population variance of a numeric expression evaluated over a set, using the biased population formula. |

### Other Functions

| Function | Description |
|---|---|
| Call | Executes the string expression containing a user-defined function. |

### Set Functions

| Function | Description |
|---|---|
| AddCalculatedMembers | Adds calculated members to a set. |
| AllMembers | Returns a set containing all members of a specified dimension or level, including calculated members. |
| Ancestors | Returns all the ancestors of a member at a specified distance. |
| Ascendants | Returns the set of the ascendants of the member, including the member itself. |
| Axis | Returns the set associated with the main axis. |
| BottomCount | Returns a specified number of items from the bottom of a set, optionally ordering the set first. |

| | |
|---|---|
| BottomPercent | Sorts a set and returns the bottom *n* elements whose cumulative total is at least a specified percentage. |
| BottomSum | Sorts a set and returns the bottom *n* elements whose cumulative total is at least a specified value. |
| Children | Returns the children of a member. |
| Crossjoin | Returns the cross product of two sets. |
| Descendants | Returns the set of descendants of either a member or a set at a specified level or at a specified distance from each member, optionally including or excluding descendants in other levels. |
| Distinct | Eliminates duplicate tuples from a set. |
| DrilldownLevel | Drills down the members of a set, at a specified level, to one level below. Alternatively, drills down on a specified dimension in the set. |
| DrilldownLevelBottom | Drills down the bottom *n* members of a set, at a specified level, to one level below. |
| DrilldownLevelTop | Drills down the top *n* members of a set, at a specified level, to one level below. |
| DrilldownMember | Drills down the members in a set that are present in a second specified set. |
| DrilldownMemberBottom | Similar to **DrilldownMember**, except that it includes only the bottom *n* children. |
| DrilldownMemberTop | Similar to **DrilldownMember**, except that it includes only the top *n* children. |
| DrillupLevel | Drills up the members of a set that are below a specified level. |
| DrillupMember | Drills up the members in a set that are present in a second specified set. |
| Except | Finds the difference between two sets, optionally retaining duplicates. |
| Extract | Returns a set of tuples from extracted dimension elements. The opposite of **Crossjoin**. |
| Filter | Returns the set resulting from filtering a set based on a search condition. |
| Generate | Applies a set to each member of another set and joins the resulting sets by union. |
| Head | Returns the first specified number of elements in a set. |
| Hierarchize | Orders the members of a set in a hierarchy. |
| Intersect | Returns the intersection of two input sets, optionally retaining duplicates. |
| LastPeriods | Returns a set of members prior to and including a specified member. |
| Members | Returns the set of all members in a dimension, hierarchy, or level. |
| Mtd | A shortcut function for the **PeriodsToDate** function that specifies the level to be Month. |
| NameToSet | Returns a set containing a single member based on a string expression containing a member name. |
| NonEmptyCrossjoin | Returns the cross product of two or more sets, excluding empty members. |
| Order | Arranges members of a set, optionally preserving or breaking the hierarchy. |
| PeriodsToDate | Returns a set of sibling members from the same level as a given member, starting with the first sibling and ending with the given member, as constrained by a specified level in the Time dimension. |
| Qtd | A shortcut function for the **PeriodsToDate** function that specifies the level to be Quarter. |
| Siblings | Returns the siblings of a member, including the member itself. |
| StripCalculatedMembers | Removes calculated members from a set. |
| StrToSet | Constructs a set from a string expression. |

| | |
|---|---|
| Subset | Returns a subset of elements from a set. |
| Tail | Returns a subset from the end of a set. |
| ToggleDrillState | Toggles the drill state of members. This function is a combination of **DrillupMember** and **DrilldownMember**. |
| TopCount | Returns a specified number of items from the top of a set, optionally ordering the set first. |
| TopPercent | Sorts a set and returns the top *n* elements whose cumulative total is at least a specified percentage. |
| TopSum | Sorts a set and returns the top *n* elements whose cumulative total is at least a specified value. |
| Union | Returns the union of two sets, optionally retaining duplicates. |
| VisualTotals | Dynamically totals child members specified in a set using a pattern for the total label in the result set. |
| Wtd | A shortcut function for the **PeriodsToDate** function that specifies the level to be Week. |
| Ytd | A shortcut function for the **PeriodsToDate** function that specifies the level to be Year. |

**String Functions**

| Function | Description |
|---|---|
| CalculationPassValue | Returns the value of an MDX expression evaluated over the specified calculation pass of a cube. |
| CoalesceEmpty | Coalesces an empty cell value to a string or number. |
| Generate | Returns a concatenated string created by evaluating a string expression over a set. |
| IIf | Returns one of two string or numeric values determined by a logical test. |
| LookupCube | Returns the value of an MDX expression evaluated over another specified cube in the same database. |
| MemberToStr | Constructs a string from a member. |
| Name | Returns the name of a dimension, hierarchy, level, or member. |
| Properties | Returns a string containing a member property value. |
| SetToStr | Constructs a string from a set. |
| TupleToStr | Constructs a string from a tuple. |
| UniqueName | Returns the unique name of a dimension, level, or member. |
| UserName | Returns the domain name and user name of the current connection. |

**Tuple Functions**

| Function | Description |
|---|---|
| Current | Returns the current tuple from a set during an iteration. |
| Item | Returns a tuple from a set. |
| StrToTuple | Constructs a tuple from a string. |

# Registered Function Libraries

Microsoft® SQL Server™ 2000 Analysis Services includes and automatically registers the Microsoft Visual Basic® for Applications Expression Services library of functions, and

automatically registers the Microsoft Excel worksheet library if it is installed on the computer with Analysis Services.

Analysis Services supports many but not all functions in these libraries. For information about supported functions, see Visual Basic for Applications Functions and Excel Functions.

# Visual Basic for Applications Functions

Microsoft® SQL Server™ 2000 Analysis Services supports many functions in the Microsoft Visual Basic® for Applications Expression Services library. This library is included with Analysis Services and automatically registered. Functions not supported in this release are marked by an asterisk in this table. For more information about syntax and examples of these functions, search on the function name in the MSDN® Library at the Microsoft Web site.

| | | | | | |
|---|---|---|---|---|---|
| Abs | *Add | *AppActivate | Array | Minute | *MkDir |
| Asc | AscB | AscW | Atn | *MonthName | Now |
| *Beep | *Calendar | *CallByName | CBool | *NPV | Oct |
| CByte | Ccur | CDate | CDbl | Pmt | PV |
| *CDec | *ChDir | *ChDrive | Choose | *Raise | Rate |
| Chr | *ChrB | ChrW | CInt | *Replace | RGB |
| *Clear | CLng | *Command | Cos | RightB | Rnd |
| *Count | *CreateObject | CSng | CStr | RTrim | Second |
| *CurDir | Cvar | CVDate | *CVErr | *SendKeys | Sgn |
| Date | DateAdd | DateDiff | DatePart | Sin | *Source |
| DateSerial | DateValue | Day | DDB | *Split | Str |
| *DeleteSetting | *Description | *Dir | *DoEvents | String | Switch |
| *Environ | *EOF | *Err | *Error | Tan | Timer |
| Exp | *FileAttr | *FileCopy | *FileDateTime | TimeValue | TypeName |
| FileLen | *Filter | Fix | Format | Val | Weekday |
| *FormatCurrency | *FormatDateTime | *FormatNumber | *FormatPercent | *Width | Month |
| *FreeFile | FV | *GetAllSettings | *GetAttr | *MIRR | NPer |
| *GetObject | *GetSetting | *HelpContext | *HelpFile | *MsgBox | Partition |
| Hex | Hour | IIf | *IMEStatus | *Number | QBColor |
| *Input | *InputB | *InputBox | InStr | PPmt | *Remove |
| InStrB | *InStrRev | Int | IPmt | *Randomize | Right |
| *IRR | *IsArray | IsDate | IsEmpty | *Reset | Round |
| IsError | *IsMissing | IsNull | IsNumeric | *RmDir | *Seek |
| IsObject | *Item | *Join | *Kill | *SaveSetting | *Shell |
| *LastDllError | LCase | Left | LeftB | *SetAttr | Space |
| Len | LenB | *Loc | *LOF | SLN | StrComp |
| Log | LTrim | Mid | MidB | Sqr | SYD |
| Trim | *WeekdayName | Time | UCase | *StrReverse | TimeSerial |
| *VarType | Year | | | | |

# Excel Functions

Microsoft® SQL Server™ 2000 Analysis Services supports many functions in the Microsoft Excel worksheet library, which is automatically registered if installed on the computer with Analysis Services. Functions not supported in this release are marked by an asterisk in this table.

| | | | | | | |
|---|---|---|---|---|---|---|
| Acos | Acosh | And | *Application | SearchB | Poisson | VarP |
| Asc | Asin | Asinh | Atan2 | Slope | Product | Weibull |
| Atanh | AveDev | Average | BetaDist | StDevP | Radians | Var |
| BetaInv | BinomDist | Ceiling | ChiDist | Sum | ReplaceB | Weekday |
| ChiInv | ChiTest | Choose | Clean | SumX2MY2 | RoundDown | Sln |
| Combin | Confidence | Correl | Cosh | Tanh | Power | NormSDist |
| Count | CountA | *CountBlank | *CountIf | Transpose | Proper | Odd |
| Covar | *Creator | CritBinom | *DAverage | TTest | *Rank | Percentile |
| Days360 | Db | Dbcs | *DCount | Vdb | Rept | Pmt |
| *DCountA | Ddb | Degrees | DevSq | ZTest | RoundUp | NormInv |
| *DGet | *DMax | *DMin | Dollar | Sinh | Ppmt | Npv |
| *DProduct | *DStDev | *DStDevP | *DSum | Small | Pv | Pearson |
| *DVar | *DVarP | Even | ExponDist | StEyx | Rate | Pi |
| Fact | FDist | Find | FindB | *SumIf | Roman | NormDist |
| FInv | Fisher | FisherInv | Fixed | SumX2PY2 | RSq | NPer |
| Floor | Forecast | *Frequency | FTest | TDist | Prob | *Parent |
| Fv | GammaDist | GammaInv | GammaLn | *Trend | Quartile | Permut |
| GeoMean | *Growth | HarMean | *HLookup | USDollar | Replace | NegBinomDist |
| HypGeomDist | *Index | Intercept | Ipmt | *VLookup | Round | NormSInv |
| Irr | IsErr | IsError | IsLogical | Skew | Search | Or |
| IsNA | IsNonText | IsNumber | Ispmt | Standardize | StDev | PercentRank |
| IsText | Kurt | Large | *LinEst | Substitute | *Subtotal | *MMult |
| Ln | Log | Log10 | *LogEst | SumProduct | SumSq | Mode |
| LogInv | LogNormDist | *Lookup | Match | SumXMY2 | Syd | Trim |
| Max | *MDeterm | Median | Min | Text | TInv | TrimMean |
| *MInverse | MIrr | | | | | |

# User-Defined Functions with MDX Syntax

You can create and register your own functions that operate on multidimensional data. These functions, called user-defined functions, can accept arguments and return values in the Multidimensional Expressions (MDX) syntax. You can create user-defined functions using Component Object Model (COM) automation languages such as Microsoft® Visual Basic® or Microsoft Visual C++®. A user-defined function can be developed using any tool capable of generating Microsoft ActiveX® libraries.

> **SECURITY NOTE** User-defined functions can be a source of security vulnerabilities; they can invoke system functions or other user-defined functions without user knowledge or intervention and may contain security credentials that are stored in plain text. Before implementing user-defined functions, review the functions for security issues. Always use absolute paths when loading libraries that contain user-defined functions.

Before you use a user-defined function, you must register the library (that is, file) in which it is compiled. You can register user-defined function libraries of the following types:

- Type libraries (*.olb, *.tlb, *.dll)
- Executable files (*.exe, *.dll)

- ActiveX controls (*.ocx)

To register a user-defined function library, issue a USE LIBRARY statement. Its syntax is:

```
USE LIBRARY "<library_path_and_file_name>" | <program_ID>
[,"<library_path_and_file_name>" | <program_ID>...]
```

Example:

```
USE LIBRARY "c:\functions\mylib.dll"
```

To register multiple libraries, issue a USE LIBRARY statement with multiple parameters in a comma-separated list. Example:

```
USE LIBRARY "c:\functions\mylib.dll","c:\functions\johnslib.dll"
```

A USE LIBRARY statement with no parameters unregisters all function libraries except the Microsoft SQL Server™ 2000 Analysis Services function library. Hidden and restricted user-defined functions are not supported.

> **Note** User-defined functions are supported only if they accept as arguments only string or numeric data types, or array or variant data types containing string or numeric values. In addition, user-defined functions are supported only if they return only string or numeric data types, or variant data types containing numeric values. Multiple user-defined functions can reside in the same ActiveX library.

# Calling a User-Defined Function within MDX

After a user-defined function is registered, it can be used anywhere in the MDX syntax that allows expressions. For example:

```
With
   Member Measures.[ForSales] As 'Sales*ForGrowthRate(SaleReps.CurrentMember.Name)'
Select TopCount(SalesReps, HowManyReps(), Sales) on Rows,
       {Sales, [Forecasted Sales]} on Columns
From Sales
```

The HowManyReps and ForecastedGrowthRate user-defined functions are defined as:

```
Public Function HowManyReps() as Integer
Public Function ForecastedGrowthRate(RepName as String) as Double
```

User-defined functions can also be used in Calculated Member Builder.

> **Note** When you call a user-defined function, you can omit an optional argument only if you also omit all arguments that follow it.

# Function Precedence and Qualification

If multiple function libraries contain a function with the same name, the Analysis Services function library takes precedence. Excluding the Analysis Services function library, precedence is resolved in order of registration by the USE LIBRARY statement. You can override precedence or call functions from specific libraries by using the following syntax when you invoke the function:

```
programid!functionname(argument1,argument2,...)
```

The function name is preceded by the function library's program ID and an exclamation point (!). This syntax ensures that the correct function is called in cases where a function name is not unique among libraries. If a library includes multiple interfaces, you can use the following syntax to specify the library and interface:

```
programid!interfaceid!functionname(argument1,argument2,...)
```