SAS® 9.4 OLAP Server: MDX Guide, Fifth Edition

SAS® Documentation
October 20, 2022

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® 9.4 OLAP Server: MDX Guide, Fifth Edition*. Cary, NC: SAS Institute Inc.

SAS® 9.4 OLAP Server: MDX Guide, Fifth Edition

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

January 2023

9.4-P7:mdxag

Contents

What's New in the SAS 9.4 OLAP Server

Overview

The SAS 9.4 OLAP Server provides the following changes and enhancements:

- a new OLAPCONTENTS procedure for generating reports of OLAP cube information
- a new OLAPCONFIG method for setting SAS OLAP Server options in batch mode
- updates to the OLAPOPERATE procedure
- a new MDX CASE statement
- a new VISUALTOTALS_BEHAVIOR option for calculated measures

OLAPCONTENTS

The OLAPCONTENTS procedure enables you to connect to a SAS OLAP Server and generate reports for SAS OLAP cubes. With this procedure, you can gather details about various cube details, including the tables that were used to load the cube, the dimension structures (hierarchy, level, property), and measures. The CONNECT command enables you to connect to a SAS OLAP Server. After you have connected, the DISPLAY CUBE command enables you to specify cube information to include in the output report. See "OLAPCONTENTS Procedure" in *SAS OLAP Server: User's Guide* for more information.

OLAPCONFIG

The OLAPCONFIG file enables you to set multiple server options in batch mode when starting a SAS OLAP Server. You can set options for cache size, available memory, tuple size, buffer size, and several other parameters. See "Setting SAS OLAP Server Options" in *SAS OLAP Server: User's Guide* for more information.

OLAPOPERATE

Several statements for cluster management were added for SAS 9.4. These statements enable you to pause, quiesce, stop, and resume a cluster. The NODEONLY option was also added to several LIST statements. Starting with SAS 9.4M1, a new LOGICALSERVERNAME option has been added to the OLAPOPERATE procedure. See "OLAPOPERATE Procedure" in *SAS OLAP Server: User's Guide* for more information.

MDX CASE Statement

The CASE statement is an MDX scripting statement that is used to compare an initial input expression to one or more conditional expression values. There are two types of CASE statements:

- simple CASE statement
- searched CASE statement

The simple CASE statement enables you to compare an initial input expression to one or more WHEN expression values. The searched CASE statement enables you to conduct more complex evaluations, as you can use a Boolean expression to assign a range of scalar values to the WHEN expression. See the "MDX CASE Statement" on page 11 for more information.

VISUALTOTALS_BEHAVIOR Option

The VISUALTOTALS_BEHAVIOR option enables you to clear total rows, subtotal rows, or both total and subtotal rows for calculated measures. When

VISUALTOTALS_BEHAVIOR is set on a calculated member, and that calculated member is then used in an MDX statement, subtotal and or total calculations are not computed. See the "OLAP VISUALTOTALS_BEHAVIOR Option" in *SAS OLAP Server: User's Guide* for more information.

Miscellaneous Updates

- Beginning in SAS 9.4, properties for multilingual cubes are automatically updated in multiple languages when reports are generated.
- When using the SQL pass-through facility for OLAP, the SQLRC macro can be used to determine the success or failure status of a PROC SQL connection and its contents. See "Verifying the SQL Connection with the SQLRC Macro" in *SAS OLAP Server: User's Guide* for more information.
- You can now remove a drill-through table from a cube with the following syntax: DT_TABLE=_NULL_UPDATE_DISPLAY_NAMES. See the option "UPDATE_DISPLAY_NAMES" in *SAS OLAP Server: User's Guide* for more information.
- Beginning in SAS 9.4, sets are now allowed on the slicer axis. See "Using Sets on the Slicer Axis" on page 13 for more information.
- Beginning in SAS 9.4, you can enable execution of user-defined (FORMAT procedure style) formats in SPDS. See the topic " Improving ROLAP Throughput Performance with SPDS" in *SAS OLAP Server: User's Guide* for more information.
- Starting with SAS 9.4M1, a new INSTR MDX function has been added. See "INSTR" on page 74 for more information.
- Starting with SAS 9.4M1, the LOCKDOWN option and statement are supported by the SAS OLAP Server.
- The EXISTS MDX function is now documented. See "Exists" on page 70.
- Starting with SAS 9.4M3, the chapter "SAS OLAP Variations" in *SAS OLAP Server: User's Guide* has been added.
- Starting with SAS 9.4M3, the topic "MDX Function Behavior and Ragged or Unbalanced Hierarchies" in *SAS OLAP Server: User's Guide* has been added.
- The option "THREADPOOLQRY" in *SAS OLAP Server: User's Guide* is now documented.

1

MDX Introduction and Overview

MDX Overview

Multidimensional Expressions (MDX) is a powerful syntax that enables you to query multidimensional objects and provide commands that retrieve and manipulate multidimensional data from those objects. MDX is designed to ease the process of accessing data from multiple dimensions. It addresses the conceptual differences between two-dimensional and multidimensional querying. MDX provides functionality for creating and querying multidimensional structures called *cubes* with a full and complete language of its own.

MDX is similar to the Structured Query Language (SQL), and MDX provides *Data Definition Language (DDL)* syntax for managing data structures. However, its features can be more complex and robust than SQL's features. The SAS 9.4 OLAP Server technology uses MDX to create OLAP cubes and data queries. MDX is part of the underlying foundation for the SAS 9.4 OLAP Server architecture, and it offers detailed and efficient searches of multidimensional data.

With MDX, specific portions of data from a cube can be extracted and then further manipulated for analysis. This allows for a thorough and flexible examination of SAS OLAP cube data. Users of MDX can take advantage of such features as calculated measures, numeric operations, and axis and slicer dimensions.

Basic MDX and Cube Concepts

Dimensions

Dimensions are the top or highest categories of a cube. They contain subcategories of data known as levels and measures. A dimension can have multiple hierarchies and can be used in multiple cubes. A cube can have up to 64 dimensions.

Hierarchies

A dimension might be categorized into different hierarchies. For example, a company might categorize its profit dimension along the verticals of geography, sales territory, or market.

Levels

Levels are categories of organization within a dimension. Levels are hierarchical, and each level descending in a dimension is a component of the previous level. For example, a time dimension could include the following levels: Year, Quarter, Month, Week, and Day.

Members and Measures

An additional component of a dimension and a level is a member. A member is a component of a level and is analogous to the value of a variable on an individual record in a data set. It is the smallest level of data in an OLAP cube. In addition to creating dimension members, a user can create calculated members and named sets that are based on underlying members or on other calculated members and named sets. These user-defined objects are based on evaluated query data from the cube.

Calculated members and named sets can be created in three different ways:

Query scope calculated member	is available only during the query that defines it. It is created by using the WITH MEMBER/SET keyword.

Session scope calculated member	is available for the user that defines the object for the duration of that session. It is created by using the CREATE SESSION MEMBER/SET keyword.
Global scope calculated member	is available for anyone to use and is stored with the cube. It is created by using the CREATE GLOBAL MEMBER/SET keyword. Named sets have the same three scopes.

Calculated members can be created in the Measures dimension and can include any combination of members. Calculated members can also be created in any other dimension and are known as *nonmeasure-based calculated members*. Examples of measures include sales counts, profit margins, and distribution costs.

Additional MDX Concepts and Expressions - Tuples and Sets

MDX extracts multidimensional views of data. A *tuple* is a slice of data from a cube. It is a selection of members (or cells) across dimensions in a cube. It can also be viewed as a cross-section or vector of member data in a cube. A tuple can be composed of member(s) from one or more dimensions. However, a tuple cannot be composed of more than one member from the same dimension.

Sets are collections of tuples. The order of tuples in a set is important when querying cube data and is known as *dimensionality*. It is important to note that the order of the dimension members in every tuple must be the same. For example, if your first tuple is (time_dimension_member, geography_dimension_member), then every other tuple in that set must also have two members in it, the first from the time dimension and the second from the geography dimension.

2

MDX Queries and Syntax

Basic MDX Queries and Syntax

MDX SELECT Statement

Basic MDX queries use the SELECT statement to identify a data set that contains a subset of multidimensional data. The basic MDX SELECT statement consists of the following clauses:

WITH clause (optional)
: This allows calculated members or named sets to be computed during the processing of the SELECT and WHERE clauses.

Note: You might encounter a syntax error when a member name containing a single quotation mark is used for a calculated member in an MDX query. To prevent this, include an additional single quotation mark in the member name that contains the quotation mark.

SELECT clause
: The SELECT clause defines the axes for the MDX query structure by identifying the dimension members to include on each axis. The number of axis dimensions of an MDX SELECT statement is also determined by the SELECT clause. The members from each dimension (to include on each axis of the MDX query) must be identified.

FROM clause
: The cube that is being queried is named in the FROM clause. It determines which multidimensional data source will be used when extracting data to populate the result set of the MDX SELECT statement. The FROM clause (in an MDX query) can list only a single cube. Queries are restricted to a single data source or cube.

WHERE clause (optional)
: The WHERE clause further restricts the result data. The axis that is formed by the WHERE clause is often referred to as the slicer. The WHERE clause

determines which dimension or member is used as a slicer dimension. This restricts the extracting of data to a specific dimension or member. Any dimension that does not appear on an axis in the SELECT clause can be named on the slicer.

Note: Beginning in SAS 9.4, sets are now allowed on the slicer axis. See"Using Sets on the Slicer Axis" on page 13 for more information.

Note: MDX queries, and specifically the SELECT statement, can have up to 128 axis dimensions. The first five axes have aliases. Furthermore, an axis can be referred to by its ordinal position within an MDX query or by its alias. In total you can have a maximum of 64 different axes.

The SELECT clause of the statement supports using MDX functions to construct different members in a set on axes. The WITH clause of the statement supports using MDX functions to construct calculated members to be used in an axis or slicer. The following example shows the syntax for the SELECT statement:

```
[WITH
  [MEMBER <member-name> AS '<value-expression>' |
   SET <set-name> AS '<set-expression>'] . . .]
SELECT [<axis_specification>
       [, <axis_specification>...]]
  FROM [<cube_specification>]
[WHERE [<slicer_specification>]]
```

MDX Syntax

When you create and edit MDX queries, be aware of the following syntax guidelines:

- MDX keywords are case insensitive. However, to easily locate keywords in your code, consider using uppercase text when documenting keywords in an MDX query.
- Do not use reserved words as names or identifiers. You can, however, quote reserved words.

  Note: For more information about reserved words see "SAS MDX Reserved Keywords" on page 19 .

- Brackets used in MDX queries should balance.—for example: [ ], ( ), and { }. If brackets do not balance, you should use the SAS option VALIDVARNAME.
- Single and double quotation marks should balance.

MDX Drillthrough

DRILLTHROUGH Statement

The DRILLTHROUGH statement is used in Multidimensional Expressions (MDX) to retrieve the source rowset or rowsets from the source data for a cube cell or specified tuple. This statement enables a client application to retrieve the rowsets that were used to create a specified cell in a cube. An MDX statement is used to specify the subject cell. All of the rowsets that make up the source data of that cell are returned. The total number of rowsets that are returned can also be affected by the MAXROWS and FIRSTROWSET modifiers. Not all cubes support drill–through. Only cubes that have a drill-through table that is specified at cube creation support drill-through.

Here is the syntax for the DRILLTHROUGH statement:

```
<drillthrough>    :=DRILLTHROUGH
[<maxrows>][<firstrowset>] <MDX select>
   <maxrows>     := MAXROWS <positive number>
   <firstrowset> := FIRSTROWSET <positive number>
```

The following modifiers can be used with the DRILLTHROUGH statement:

MAXROWS
: indicates the maximum number of rows that should be returned by the resulting rowset.

FIRSTROWSET
: specifies the first rowset to return.

Specifying the Maximum Number of Drill-Through Rows

You can limit the number of drill-through rows that users request in a query by selecting the OLAP server definition setting **Maximum number of flattened rows** from SAS Management Console. This setting controls the maximum number of flattened rows that are allowed for flattened (two-dimensional) data sets.

The following steps enable you to specify the maximum number of flattened rows:

1. In the tree view for the Server Manager plug-in of SAS Management Console, select the node for your OLAP server. This is the physical OLAP server and is located by drilling down from the top of the Server Manager tree view.
2. After selecting the physical OLAP server, right-click and select **Properties**.

3 At the SAS OLAP Server Properties dialog box, select the **Options** tab, and then the **Advanced Options** button.

4 At the Advanced Options dialog box, select the **Server** tab and enter the needed value for the **Maximum number of flattened rows** field. The default setting is 300,000 rows.

Ensuring That Tables Are Accessible at Query Time

Data that is external to a cube must be available to the SAS OLAP Server under the following conditions:

- If the cube does not include an NWAY, then the SAS OLAP Server must have access to the input data source table (also called the detail data) and any specified dimension tables.
- If the cube is associated with a drill-through table, then the SAS OLAP Server must have access to the drill-through table.
- If the cube uses pre-summarized aggregation tables, then the SAS OLAP Server must have access to those tables.

To ensure that the necessary tables are accessible at query time, the applicable library names need to be allocated. This is necessary when the OLAP server (that is associated with the OLAP schema that contains the cubes) is invoke.

Note: If any of the tables contain user-defined formats, then the SAS OLAP Server also needs information about how to find those formats. User-defined formats cannot be used with drill-through tables.

Note: For more information, see the SAS Intelligence Platform: Data Administration Guide and the SAS Intelligence Platform: System Administration Guide.

Working with User-Defined Formats

If you have existing SAS data sets, you might also have a catalog of user-defined formats and informats. You have two options for making these formats available to applications such as SAS Data Integration Studio:

- The preferred solution is to name the format catalog `formats.sas7bcat` and to place the catalog in the following directory: *path-to-configuration-directory* `\Lev1\SASMain\SASEnvironment\SASFormats`
- An alternative method of making user-defined formats “visible” is to follow this procedure:

  1 Add a line to the configuration file *path-to-configuration-directory* *\Lev1\SASMain\sasv9.cfg* that points to a configuration file for handling user-defined format catalogs. For example, you might add the following line:

```
-config
path-to-configuration-directory\Lev1\SASMain\userfmt.cfg
```

2 In the file *userfmt.cfg*, enter a SET statement and a FMTSEARCH statement.

```
-set fmtlib1
"path-to-configuration-directory\Lev1\Data\orformat"
-fmtsearch (work fmtlib1.orionfmt library)
```

This makes the format catalog *fmtlib1.orionfmt* available. For more information, see the SAS Intelligence Platform: Data Administration Guide.

Basic MDX DDL Syntax

The SAS OLAP Server provides support for the MDX Data Definition Language (DDL). DDL enables users and administrators to manage the definitions of calculated members and named sets at either a session or a global level. Management of calculated members and named sets is provided by the CREATE and DROP DDL statements.

By using the CREATE DDL statement, a user can create definitions of calculated members or named sets for use within a client session or for use within a cube on a global scale. Here is the format for the CREATE DDL statement:

```
CREATE [global | session]
  [MEMBER . AS '' |
   SET  AS ''] . . .]
```

If **GLOBAL** or **SESSION** is not specified, then the default scope is **SESSION**. When a calculated member or named set is defined within the **SESSION** scope, the definition is available only for the lifetime of the user's client session. When a calculated member or named set is defined within the **GLOBAL** scope, the definition is permanently attached to the cube definition and is visible to all current and future client sessions.

By using the DROP DDL statement, a user can remove definitions of calculated members or a named set from use within a client session or from use within a cube on a global scale. Here is the format for the DROP DDL statement:

```
DROP [MEMBER . . . .] |
     [SET ] . . .] .
```

When using the DROP statement, only calculated members or named sets can be dropped at the same time. However, a user cannot drop both calculated members and named sets in a single DROP statement.

Note: The name of the calculated member or named set must contain the cube name.

MDX CASE Statement

Overview

The CASE statement is an MDX scripting statement that is used to compare an initial input expression to one or more conditional expression values. The statement then identifies and returns scalar data values that are based on expressions in the script. If a conditional expression matches the initial input expression, a designated value is returned. You can compare an initial input expression against single conditional values or a range of conditional values.

The CASE statement is similar to the IIF function but can process more conditions. There are two types of CASE statements; the Simple CASE statement and the Searched CASE statement.

Simple CASE Statement

The simple CASE statement enables you to compare an initial input expression to one or more WHEN expression values. As the script evaluates the input expression against WHEN expressions, it searches for a match between these expressions. If the two values match, the value immediately after THEN for that WHEN expression is returned. The cycle is repeated for every WHEN expression in the script. WHEN expressions are evaluated in the order in which they are listed in the CASE statement. If no match is found, and you would like a default value returned, you can include an optional ELSE clause in the script to apply a default value if needed. Otherwise, a value of NULL is returned.

Here is the syntax for the simple CASE statement:

```
CASE [input_expression] WHEN when_expression THEN when_true_result_expression[...n]
 [ELSE else_result_expression]END
```

Here is an example of the simple CASE statement. It measures performance ratings for a company:

```
WITH MEMBER [measures].[Performance] AS
CASE [measures].[PerfRating]
   WHEN 5 THEN 'Excellent'
   WHEN 4 THEN 'Very Good'
   WHEN 3 THEN 'Good'
   WHEN 2 THEN 'Poor'
   WHEN 1 THEN 'Very Poor'
ELSE 'Unknown'
END
```

Searched CASE Statement

The searched CASE statement enables you to conduct more complex evaluations. Like the simple CASE statement, it enables you to compare an initial input expression to one or more WHEN expression values. However, with this statement, you can use a Boolean expression to assign a range of scalar values to the WHEN expression. The CASE statement analyzes that range of values, and if a match is made to the input expression, the value immediately after THEN for that WHEN expression is returned. If no match is found, and you would like a default value returned, you can include an optional ELSE clause in the script to apply a default value if needed. Otherwise, a value of NULL is returned.

Here is the syntax for the searched CASE statement:

```
CASE WHEN Boolean_expression THEN when_true_result_expression [...n]
 [ELSE else_result_expression] END
```

Here is an example of the searched CASE statement. It assigns color coding to specific age ranges:

```
WITH MEMBER [Measures].[Age_Category] AS
CASE
   WHEN [Measures].[Age] > 65 THEN 'Gold'
   WHEN [Measures].[Age] > 55 THEN 'Silver'
   WHEN [Measures].[Age] > 45 THEN 'Bronze'
   WHEN [Measures].[Age] > 35 THEN 'Blue'
   WHEN [Measures].[Age] > 25 THEN 'Green'
   ELSE "BLACK"
END
```

MDX CASE Statement Arguments

The following arguments are used with the CASE statement:

input_expression
is an MDX expression that resolves to a scalar value.

when_expression
specifies a scalar value against which the *input_expression* is evaluated. When evaluated to true, it returns the scalar value of the *else_result_expression*.

when_true_result_expression
is the scalar value returned when the WHEN clause evaluates to true.

else_result_expression
is the scalar value returned when none of the WHEN clauses evaluate to true.

Boolean_expression
is an MDX expression that evaluates to a scalar value.

Using Sets on the Slicer Axis

Beginning in SAS 9.4, sets are now allowed on the slicer axis. The set can either be expressed as a grouping of sets of members from different hierarchies or as a set expression that will be evaluated. Here are two MDX query examples for applying sets to the slicer axis:

```
select
  cars.[all cars].children on 0
from mddbcars
where ({dealers.[all dealers].finch, dealers.[all dealers].smith},
       {[DATE].[All DATE].[February], [DATE].[All DATE].[April],
[DATE].[All DATE].[May]})
```

```
select
  cars.[all cars].children on 0
from mddbcars
where crossjoin({dealers.[all dealers].finch, dealers.[all
dealers].smith},
                {[DATE].[All DATE].[February], [DATE].[All DATE].
[April],
[DATE].[All DATE].[May]})
```

Both of these MDX queries return the same results. The aggregation of these members is used to determine the value for each cell in the query. The values for the cars in both queries only reflect those cars that were sold by either dealer Finch or dealer Smith and only for the months of February, April, and May.

When sets are used on the slicer and the *CurrentMember* function is used either directly or indirectly for one of the hierarchies with sets on the slicer, then the first member of the set is used. Here is an example:

```
with
  member [measures].[dealercm] as '[dealers].currentmember.name'

select
  [measures].[dealercm] on 0
from mddbcars
where ({dealers.[all dealers].finch, dealers.[all dealers].smith})
This returns:

    dealercm
       Finch
```

Previously, beginning in SAS 9.2, implicit sets on the slicer were used. Currently, using implicit sets on the slicer is considered the optimal way to apply sets to the slicer axis.

SAS Functions

SAS Function in an MDX Query

SAS functions are functions that anyone can reference in MDX expressions. SAS functions are slightly limited in the arguments that they accept and return. Here is an MDX query that uses a SAS function called “MDY”:

```
 WITH MEMBER measures.mdy AS 'SAS!mdy(2,9,2003)'
SELECT {cars.members} ON 0 FROM mddbcars
WHERE (measures.mdy)
```

The resulting cells look like this:

```
NOTE:    0.3[0]: f=15742 (u=15742.00)
NOTE:    0.3[1]: f=15742 (u=15742.00)
NOTE:    0.3[2]: f=15742 (u=15742.00)
NOTE:    0.3[3]: f=15742 (u=15742.00)
NOTE:    0.3[4]: f=15742 (u=15742.00)
NOTE:    0.3[5]: f=15742 (u=15742.00)
NOTE:    0.3[6]: f=15742 (u=15742.00)
NOTE:    0.3[7]: f=15742 (u=15742.00)
NOTE:    0.3[8]: f=15742 (u=15742.00)
NOTE:    0.3[9]: f=15742 (u=15742.00)
NOTE:    0.3[10]: f=15742 (u=15742.00)
NOTE:    0.3[11]: f=15742 (u=15742.00)
NOTE:    0.3[12]: f=15742 (u=15742.00)
NOTE:    0.3[13]: f=15742 (u=15742.00)
NOTE:    0.3[14]: f=15742 (u=15742.00)
NOTE:    0.3[15]: f=15742 (u=15742.00)
```

In order to gain access to a SAS function library and before you can use a SAS function in a query, you must define or open the library for the current session. To do this, apply the USE statement at the beginning of your MDX query:

```
USE
LIBRARY "SAS"
```

SAS Functions Available for Use in MDX Expressions

Table 2.1 SAS Functions Available for Use in MDX Expressions

Function	Description	Argument
DATE	Returns the current date in SAS date format.	(none)
DATEJUL	Converts a Julian date to a SAS date value.	«julian-date»
DATEPART	Returns a SAS date value that corresponds to the date portion of a SAS datetime value .	«SAS datetime»
DATETIME	Returns the current data and time in SAS datetime format.	(none)
DAY	Returns an integer that represents the day of the month from a SAS date value.	«SAS date»
DHMS	Returns a SAS datetime value from a numeric expression that represents the date, hour, minute, and second.	«SAS date», «hour», «minute», «second»
HMS	Returns a SAS time value from a numeric expression that represents the hour, minute, and second.	«hour», «minute», «second»
HOUR	Returns a numeric value that represents the hour from a SAS time or datetime value.	«SAS time» | «SAS datetime»
IN	Returns TRUE if the first expression is contained in the list of expressions that start from the second parameter to the end of the parameters provided; otherwise, FALSE.	«expression», «expression1», . . ., «expressionN»
JULDATE	Converts a SAS date value to a numeric value that represents a Julian date.	«SAS date»

Function	Description	Argument
JULDATE7	Converts a SAS date value to a numeric value that represents a Julian date with the year represented in 4 digits.	«SAS date»
LEFT	Returns the argument with leading blanks moved to the end of the value; the argument's length does not change.	«argument»
MDY	Returns a SAS date value from numeric expressions that represent the month, day, and year.	«month», «day», «year»
MINUTE	Returns a numeric value that represents the minute from a SAS time or datetime value.	«SAS time» | «SAS datetime»
MONTH	Returns a numeric value that represents the month from a SAS time.	«SAS date»
QTR	Returns a value of 1, 2, 3, or 4 from a SAS date value to indicate the quarter of the year during which the SAS date value falls.	«SAS date»
RIGHT	Returns the argument with trailing blanks moved to the beginning of the value; the argument's length does not change.	«argument»
ROUND	Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted.	(argument <,rounding-unit>)
SECOND	Returns a numeric value that represents the second from a SAS time or datetime value.	«SAS time» | «SAS datetime»
SUBSTR	Returns a portion of the string expression argument, starting at the index position and returning up to “n” characters. If “n” is not specified, then the rest of the string is returned.	«argument», «position» <, «n»>
TIME	Returns the current time in SAS time format.	(none)

Function	Description	Argument
TIMEPART	Returns a SAS time value that corresponds to the time portion of a SAS datetime value.	«SAS datetime»
TODAY	Returns the current date in SAS date format.	(none)
TRIM	Returns the argument with the trailing blanks removed; if the argument contains all blanks, then the result is a string with a single blank .	«argument»
TRIMN	Returns the argument with the trailing blanks removed; if the argument contains all blanks, then the result is a null string.	«argument»
TRUNC	Truncates a numeric value to a specified length.	(number,length)
UPCASE	Returns the argument with all lowercase characters converted to uppercase characters.	«argument»
WEEKDAY	Returns an integer that represents the day of the week, where 1 = Sunday, 2 = Monday, . . ., 7 = Saturday, from a SAS date value.	«SAS date»
YEAR	Returns a numeric value that represents the month from a SAS time.	«SAS date»
YYQ	Returns a SAS date value that corresponds to the first day of the specified quarter.	«year», «quarter»

Function Arguments and Return Types

Currently only floating-point (double) arguments, character string arguments, and return values are supported. There is no limit to the number of arguments. The promotion of arguments from MDX types to SAS data types is automatically performed when there is a difference between the two types.

Numeric Precision

Floating-point Representation

To store numbers of large magnitude and to perform computations that require many digits of precision to the right of the decimal point, the SAS OLAP Server stores all numeric values as floating-point representation. Floating-point representation is an implementation of scientific notation, in which numbers are represented as numbers between 0 and 1 times a power of 10.

In most situations, the way the SAS OLAP Server stores numeric values does not affect you as a user. However, floating-point representation can account for anomalies that you might notice in MDX numeric expressions. This section identifies the types of problems that can occur and how you can anticipate and avoid them.

Magnitude versus Precision

Floating-point representation allows for numbers of very large magnitude (such as 2^{30}) and high degrees of precision (many digits to the right of the decimal place). However, operating systems differ on how much precision and how much magnitude to allow.

Whether magnitude or precision is more important depends on the characteristics of your data. For example, if you are working with engineering data, very large numbers might be needed and magnitude will probably be more important. However, if you are working with financial data where every digit is important, but the number of digits is not great, then precision is more important. Most often, applications that are created with the SAS OLAP Server need a moderate amount of both magnitude and precision, which is handled well by floating-point representation.

Computational Considerations of Fractions

Regardless of how much precision is available, there is still the problem that some numbers cannot be represented exactly. For example, the fraction 1/3 cannot be rendered exactly in floating-point representation. Likewise, .1 cannot be rendered exactly in a base 2 or base 16 representation, so it also cannot be accurately rendered in floating-point representation. This lack of precision is aggravated by arithmetic operations. Consider the following example:

```
((10 * .1) = 1)
```

This expression might not always return TRUE due to differences in numeric precision. However, the following expression uses the ROUND function to compensate for numeric precision and therefore will always return TRUE:

```
(round((10 * .1), .001) = 1)
```

Usually, if you are doing comparisons with fractional values, it is good practice to use the ROUND function.

Using the TRUNC Function

The TRUNC function truncates a number to a requested length and then expands the number back to full precision. The truncation and subsequent expansion duplicate the effect of storing numbers in less than full precision. So in the following example, the first expression would return FALSE and the second would return TRUE:

```
((1/3) = .333)
```

```
(TRUNC((1/3), 3)
= .333)
```

When you compare the result of a numeric expression to be equal to a specific value, such as 0, it is important that you use the TRUNC and ROUND functions to ensure that the comparison evaluates as intended.

Differences with Microsoft Analysis Services 2000

Microsoft Analysis Services 2000 (AS2K) labels external functions as user-defined functions (UDFs). Because AS2K runs only on Windows, it supports calling COM libraries (usually written in Visual Basic). Because MDX evaluation can occur on either the client or the server, Microsoft provides a means to install and use libraries on either location (due to a dual-mode OLE DB for OLAP provider, MSOLAP).

If you use a client-side function, then all the execution is on the client. The SAS OLAP Server is a thin-client system that is designed for high volume and scalability, with all evaluation done on the server. Therefore, external function libraries such as SAS functions can be installed only on the server. In addition, with the proper license, you can run a server on your own computer and install any libraries that you need.

SAS MDX Reserved Keywords

A reserved keyword should not be used to reference a dimension, hierarchy, level, or member name unless the reference is enclosed in square brackets [ ]. Otherwise, the keyword might be interpreted incorrectly.

Note: The SAS OLAP Server currently does not support the use of square brackets in cube, dimension, hierarchy, level, or member names or captions.

Table 2.2 *SAS MDX Reserved Keywords*

(	DRILLDOWNMEMBER	NONEMPTYCROSSJOIN

)	DRILLDOWNMEMBERBOTTOM	NOT
*	DRILLDOWNMEMBERTOP	NULL
+	DRILLTHROUGH	ON
'	DRILLUPLEVEL	OPENINGPERIOD
-	DRILLUPMEMBER	OR
.	DROP	ORDER
/	ELSE	ORDINAL
:	EMPTY	PAGES
<	END	PARALLELPERIOD
<=	EXCEPT	PARENT
<>	EXCLUDEEMPTY	PARENT_COUNT
=	EXTRACT	PARENT_LEVEL
>	FALSE	PARENT_UNIQUE_NAME
>=	FILTER	PERIODSTODATE
{	FIRSTCHILD	POST
}	FIRSTROWSET	PREDICT
||	FIRSTSIBLING	PREVMEMBER
ABSOLUTE	FONT_FLAGS	PROPERTIES
ADDCALCULATEDMEMBERS	FONT_NAME	PTD
AFTER	FONT_SIZE	PUT
AGGREGATE	FORMATTED_VALUE	QTD
ALL	FORMAT_STRING	RANGE
ALLMEMBERS	FORE_COLOR	RANK
ANCESTOR	FROM	RECURSIVE
ANCESTORS	GENERATE	RELATIVE

AND	GLOBAL	ROLLUPCHILDREN
AS	HEAD	ROOT
ASC	HIERARCHIZE	ROWS
ASCENDANTS	HIERARCHY	SCHEMA_NAME
AVG	HIERARCHY_UNIQUE_NAME	SECTIONS
AXIS	IGNORE	SELECT
BACK_COLOR	IIF	SELF
BASC	INCLUDEEMPTY	SELF_AND_AFTER
BDESC	INTERSECT	SELF_AND_BEFORE
BEFORE	IS	SELF_BEFORE_AFTER
BEFORE_AND_AFTER	ISANCESTOR	SESSION
BOTTOMCOUNT	ISEMPTY	SET
BOTTOMPERCENT	ISGENERATION	SETTOARRAY
BOTTOMSUM	ISLEAF	SETTOSTR
CALCULATIONCURRENTPASS	ISSIBLING	SIBLINGS
CALCULATIONPASSVALUE	ITEM	S OLVE_ORDER
CALL	LAG	STDDEV
CAPTION	LASTCHILD	STDDEVP
CASE	LASTPERIODS	STDEV
CATALOG_NAME	LASTSIBLING	STDEVP
CELL	LEAD	STRIPCALCULATEDMEMB ERS
CELL_ORDINAL	LEAVES	STRTOMEMBER
CHAPTERS	LEVEL	STRTOSET
CHILDREN	LEVELS	STRTOTUPLE
CHILDREN_CARDINALITY	LEVEL_NUMBER	STRTOVALUE

CLOSINGPERIOD	LEVEL_UNIQUE_NAME	SUBSET
COALESCEEMPTY	LIBRARY	SUM
COLUMNS	LINKMEMBER	TAIL
CORRELATION	LINREGINTERCEPT	THEN
COUNT	LINREGPOINT	TOGGLEDRILLSTATE
COUSIN	LINREGR2	TOPCOUNT
COVARIANCE	LINREGSLOPE	TOPPERCENT
COVARIANCEN	LINREGVARIANCE	TOPSUM
CREATE	LOOKUPCUBE	TRUE
CROSSJOIN	MAX	TUPLETOSTR
CUBE_NAME	MAXROWS	UNION
CURRENT	MEDIAN	UNIQUENAME
CURRENTMEMBER	MEMBER	USE
DATAMEMBER	MEMBERS	USERNAME
DEFAULTMEMBER	MEMBERTOSTR	VALIDMEASURE
DESC	MEMBER_CAPTION	VALUE
DESCENDANTS	MEMBER_GUID	VAR
DESCRIPTION	MEMBER_NAME	VARIANCE
DIMENSION	MEMBER_ORDINAL	VARIANCEP
DIMENSIONS	MEMBER_TYPE	VARP
DIMENSION_UNIQUE_NAME	MEMBER_UNIQUE_NAME	VISUALTOTALS
DISPLAY_INFO	MIN	WHEN
DISTINCT	MTD	WHERE
DISTINCTCOUNT	NAME	WITH
DRILLDOWNLEVEL	NAMETOSET	WTD

DRILLDOWNLEVELBOTTOM	NEXTMEMBER	XOR
DRILLDOWNLEVELTOP	NON	YTD

External Functions

External Function Example

External functions are functions that can be written on a server that clients can later reference in MDX expressions. External functions can be written by most MDX users. External function names are case sensitive, and unlike internal functions, they are more limited in the arguments that they can take. Here is an example of an MDX query that uses an external function called **addOne()**, which takes one parameter, a double argument, and adds one (1) to it. It then returns another double argument:

```
WITH MEMBER measures.x AS
'addone(measures.sales_sum)'
SELECT {cars.members} ON 0 FROM Mddbcars
WHERE (measures.x)
```

The resulting cells look like this:

```
0.0[0]: 229001
0.0[1]: 27001
0.0[2]: 40001
0.0[3]: 86001
0.0[4]: 76001
0.0[5]: 17001
0.0[6]: 10001
0.0[7]: 20001
0.0[8]: 20001
0.0[9]: 10001
0.0[10]: 44001
0.0[11]: 17001
0.0[12]: 15001
0.0[13]: 4001
0.0[14]: 14001
0.0[15]: 58001
```

Here is the query and the resulting cells without the external **addOne()** function:

```
SELECT {cars.members} ON 0
FROM Mddbcars
WHERE (measures.sales_sum)
```

```
Array(0)=229000
Array(1)=27000
Array(2)=17000 Array(3)=10000
```

```
Array(4)=40000 Array(5)=20000
Array(6)=20000 Array(7)=86000
Array(8)=10000 Array(9)=44000
Array(10)=17000 Array(11)=15000
Array(12)=76000 Array(13)=4000
Array(14)=14000 Array(15)=58000
```

Gaining Access to an External Function Library or Class

Before you can use a function in a query, you must define or open the library for the current session. To do this, you execute the USE statement in MDX:

```
USE LIBRARY "Hello"
```

You do not add the .class extension, because it is automatically provided. When the session ends, the library is released. You can use a DROP statement to release the library before the session ends:

```
DROP LIBRARY "Hello"
```

State Information

The class is instantiated when the USE statement is first encountered in a session, and then it is released when the session ends or the DROP statement is executed. As a result, the state can be kept in a normal class and static variables can be maintained. Here is an example:

```
public
class Hello
{
   static int count = 0;
   int instance;
   int iteration = 0;
   public Hello()
    {
      instance = count++;
      System.out.println("Hello constructor " + instance);
    }
    public double addOne(double d)
    {
       System.out.println("addOne, world! " + instance + " " +
iteration++);
       return d+1.0;
    }
    public void finalize()
    {
       System.out.println("Hello finalize");
    }
 }
```

Note: `System.out` is used in the above example for illustration and cannot be used in a real function except for debugging.

Here is an example of the debugging output that is generated:

```
Hello constructor 0
addOne, world! 0 0
addOne, world! 0 2
Hello constructor 1
addOne, world! 0 3
addOne, world! 1 0
addOne, world! 1 1
```

Each time a new session (a user or client connection) uses this class, the Java constructor is called and a new `Hello` object is created. The count is incremented so that `instance` has a unique value. Example items that you might want to save in a real application include file handles, shopping cart lists, and database connection handles.

Although cleanup is automatic, you can have an optional *finalize* method for special circumstances. Normal Java garbage collection of the class occurs some time after the class is no longer needed. The finalize method should then be called. However, in accordance with Java standards, it is possible that the finalize method will never be called (for example, if the server is shut down early, or the class never needs to be removed by the garbage collector).

Function Arguments and Return Types

Only floating-point (double) arguments and return values are supported by SAS 9.4 OLAP Server. Java function overloading is also supported and there is no limit to the number of arguments that are supported.

SAS OLAP Server looks at the parameters that are passed to an external function and creates a Java signature from that. It then looks up the function and signature in the class. In the `addOne()` example that was mentioned earlier, there is one parameter. Also, because it is a double argument, it looks for the signature “D(D)”.

Performance

Certain OLAP hosts use an in-process Java virtual machine (JVM), whereas other OLAP hosts use an out-of-process JVM. An out-of-process JVM is much less efficient because each method call has to be packaged (marshaled) and transmitted to the JVM process. It is then unpackaged (unmarshaled) and run, and a return packet is sent back. Currently HP-UX, OpenVMS, and z/OS use out-of-process JVMs. In later releases, hosts should be able to use in-process JVMs. z/OS will use a shared address space so that it can be optimized.

Although synthetic benchmarks show that calling Java is considerably slower than calling built-in functions, real-world performance tests show that the performance impact of calling Java methods was negligible (at least with in-process Java

implementations). If you encounter a problem, reducing the number of function calls per output cell, the number of cells queried, and the number of parameters to the function can all boost performance.

Deployment

To make a Java class available, copy the .class file to a directory that is listed in the CLASSPATH environment variable when the server is started. The CLASSPATH can contain any number of directories that are separated by semicolons (;). The current release of SAS OLAP Server does not contain a method to make the server reload a .class file after it has been loaded. Therefore, if you update the .class file after using it one time, the server will continue to use the old version. Currently you need either to restart the server or give the new class a different name.

It is possible that later releases of SAS OLAP Server will not use CLASSPATH. A benefit of using Java for external functions is that the .class files are portable. As a result, you can use JavaC to compile your class one time, and deploy it on different machines without recompiling.

Security

Because the Java classes are loaded from the server's local file system, they have full access to the server's system (under the ID that started the server). Any public methods (on any classes) in the CLASSPATH can be invoked by any client. As a result, use caution when you decide which classes and directories to make visible.

Differences with Microsoft Analysis Server (AS2K)

See “Differences with Microsoft Analysis Services 2000” on page 19

Using Derived Statistics with the Aggregate Function

Example 1

When the aggregate function is used in a calculated member, the statistic associated with the current measure will determine how the values are aggregated. For example:

```
WITH

  MEMBER [measures].[calc] AS '
    [measures].[actual_max]-[measures].[actual_min]'

  MEMBER [time].[agg complexfunc] AS
    'aggregate([time].[all time].[1994].children)'

SELECT
  {[time].[all time].[1994].children, [time].[agg complexfunc]} ON 0,
  {measures].[actual_max], [measures].[actual_min],
   [measures].[actual_sum], [masures].[actual_n],
   [measures].[actual_avg], measures.calc} on 1
FROM [prdmddb]
```

This example returns the following:

```
                  1            2            3            4 agg complexfunc
actual_max    $1,000.00      $987.00      $992.00    $1,000.00       $1,000.00
actual_min       $13.00        $3.00       $20.00       $15.00           $3.00
actual_sum   $89,763.00   $93,359.00   $89,049.00   $88,689.00     $360,860.00
actual_n            180          180          180          180             720
actual_avg      $498.68      $518.66      $494.72      $492.72         $501.19
     calc           987          984          972          985             997
```

For each current measure listed on the left, the aggregate function does the following:

actual_max
: It takes the MAX of the actual_max values associated with the children of 1994.

actual_min
: It takes the MIN of the actual_min values associated with the children of 1994.

actual_sum
: It summarizes the actual_sum values associated with the children of 1994.

actual_n
: It counts the actual_n values associated with the children of 1994.

actual_avg
: It divides the aggregated sum of the children of 1994 by the aggregated count value of the children of 1994.

calculated measure calc
: It takes the MAX of the actual_max values for the children of 1994 (1000) and subtracts the MIN of the actual_min values for the children of 1994 (3).

Example 2

This example shows what happens when the query is changed to include a numeric expression:

```
WITH

  MEMBER [time].[agg complexfunc] AS
    'aggregate([time].[all time].[1994].children, measures.actual_max + 1)'
```

```
SELECT
  {[time].[all time].[1994].children, [time].[agg complexfunc]} ON 0,
  {[measures].[actual_max], [measures].[actual_min],
   [measures].[actual_sum], [measures].[actual_n],
   [measures].[actual_avg]} on 1
FROM [prdmddb]
```

This example returns the following:

	1	2	3	4	agg complexfunc
actual_max	$1,000.00	$987.00	$992.00	$1,000.00	$1,001.00
actual_min	$13.00	$3.00	$20.00	$15.00	$988.00
actual_sum	$89,763.00	$93,359.00	$89,049.00	$88,689.00	$3,983.00
actual_n	180	180	180	180	3983
actual_avg	$498.68	$518.66	$494.72	$492.72	

For each current measure listed on the left, the aggregate function does the following::

actual_max
: It takes the MAX of the actual_max values associated with the children of 1994 and adds 1 to it.

actual_min
: It takes the MIN of the actual_max values associated with the children of 1994 and adds 1 to it.

actual_sum
: It adds 1 to each of the actual_max values associated with the children of 1994 and SUMs the values.

actual_n
: It adds 1 to each of the actual_max values associated with the children of 1994 and SUMs the values.

actual_avg
: It cannot compute a derived measure based on another measure, so it returns a missing value.

Example 3

This example shows what happens when the query is changed to include a numeric expression with measures that aggregate differently.

```
WITH

  MEMBER [time].[agg complexfunc] AS
    'aggregate([time].[all time].[1994].children, measures.actual_max -
measures.actual_min)'

SELECT
  {[time].[all time].[1994].children, [time].[agg complexfunc]} ON 0,
  {[measures].[actual_max], [measures].[actual_min],
```

```
   [measures].[actual_sum], [measures].[actual_n],
   [measures].[actual_avg]} on 1
FROM [prdmddb]
```

This example returns the following:

```
                      1            2
3            4 agg complexfunc
actual_max    $1,000.00      $987.00      $992.00    $1,000.00         $987.00
actual_min       $13.00        $3.00       $20.00       $15.00         $972.00
actual_sum   $89,763.00   $93,359.00   $89,049.00   $88,689.00       $3,928.00
actual_n            180          180          180          180             3928
actual_avg      $498.68      $518.66      $494.72      $492.72
```

For each current measure listed on the left, the aggregate function does the following:

actual_max
: It subtracts the actual_min value associated with each child of 1994 from the corresponding actual_max value. It selects the MAX of these values (1000–13).

actual_min
: It subtracts the actual_min value associated with each child of 1994 from the corresponding actual_max value. It selects the MIN of these values (992 - 20).

actual_sum
: It subtracts the actual_min value associated with each child of 1994 from the corresponding actual_max value. It then sums all of these values.

actual_n
: It subtracts the actual_min value associated with each child of 1994 from the corresponding actual_max value. It then sums all of these values.

actual_avg
: It cannot compute a derived measure based on other measures, so it returns a missing value.

Example 4

This example shows what happens when the query is changed to have the aggregate function on a calculated measure, and the numeric expression is the actual_avg measure.

```
WITH

  MEMBER [measures].[agg complexfunc] AS
    'aggregate([time].[all time].[1994].children, measures.actual_avg)'

SELECT
  {[measures].[actual_sum], [measures].[actual_n],
    [measures].[agg complexfunc]}ON 0,
  {[time].[all time].[1994].children} ON 1
FROM [prdmddb]
```

This example returns the following:

```
actual_sum  actual_n  agg complexfunc
```

```
       1   $89,763.00       180 501.194444444444
       2   $93,359.00       180 501.194444444444
       3   $89,049.00       180 501.194444444444
       4   $88,689.00       180 501.194444444444
```

The current measure is the calculated measure [agg complexFunc]. However, using this would cause infinite recursion, so the aggregate function aggregates based only on the numeric expression. In this case, the statistic is average, which divides the sum by the count. For each child of 1994, the sum is divided by the count, and these values are summed together. This total is then divided by the number of children of 1994 to give the aggregate value.

Example 5

This example shows what happens when the numeric expression is changed to an expression that used a derived statistic.

```
WITH

  MEMBER [measures].agg complexfunc] AS
    'aggregate([time].[all time].[1994].children, measures.actual_avg + 12)'

SELECT
  {[measures].[actual_sum], [measures].[actual_n],
    [measures].[agg complexfunc]} ON 0,
  {[time].[all time].[1994].children} ON 1
FROM [prdmddb]
```

This example returns the following:

```
actual_sum     actual_n agg complexfunc
       1   $89,763.00          180
       2   $93,359.00          180
       3   $89,049.00          180
       4   $88,689.00          180
```

In this case, the value of the aggregation is missing. When measures that are associated with derived statistics are used in an expression for the aggregate function, it is not able to calculate the correct value. Therefore,it simply returns missing.

Example 6

This example shows what happens when the query is changed to have a standard statistic in the expression.

```
WITH

  MEMBER [measures].[agg complexfunc] AS
    'aggregate([time].[all time].[1994].children, measures.actual_max + 12)'
```

```
SELECT
  {[measures].[actual_max],
    [measures].[agg complexfunc]} ON 0,
  {[time].[all time].[1994].children} ON 1
FROM [prdmddb]
```

This example returns the following:

```
             actual_max agg complexfunc
       1    $1,000.00            1012
       2      $987.00            1012
       3      $992.00            1012
       4    $1,000.00            1012
```

In this case, the aggregate function looks for the max value associated with the actual_max measure for the children of 1994. Then 12 is added to this value.

Example 7

This example shows what happens when the query is changed to still have the aggregate function on a calculated measure, and it has a numeric expression that includes measures that aggregate differently.

```
WITH

  MEMBER [measures].[agg complexfunc] AS
    'aggregate([time].[all time].[1994].children, measures.actual_max +
measures.actual_min)'

SELECT
  {[measures].[actual_max], measures.actual_min,
    [measures].[agg complexfunc]} ON 0,
  {[time].[all time].[1994].children} ON 1
FROM [prdmddb]
```

This example returns the following:

```
            actual_max   actual_min agg complexfunc
       1    $1,000.00       $13.00
       2      $987.00        $3.00
       3      $992.00       $20.00
       4    $1,000.00       $15.00
```

In this case, one measure is a max and the other a min. It is unclear how to aggregate the values, so a missing value is returned.

Standard Statistics

Here are the standard statistics and how they are aggregated.

Table 2.3 Standard Statistics

Standard Statistic	How it is aggregated
MAX	Get the MAXIMUM of the values
MIN	Get the MINIMUM of the values
N	Count the values
NMISS	Sum the NMISS values
SUM	Sum the SUM values
SUMWGT	Sum the SUMWGT values
USS	Sum the USS values
UWSUM	Sum the UWSUM values

Derived Statistics

For measures associated with these statistics, the system will use the values that are being aggregated to determine the result value based on the statistic. For example, for AVG, it will take the SUM of the values and divide it by the N of the values. Here are the derived statistics:

- AVG
- RANGE
- CSS
- VAR
- STD
- ERR
- CV
- T
- PRT
- LCLM
- UCLM
- NUNIQUE

SAS OLAP Security Totals and Permission Conditions

SECURITY_SUBSET Option

As part of the SAS security model, SAS OLAP cubes can have member-level authorizations applied as permission conditions. Permission conditions limit access to a cube dimension so that only designated portions of the data is visible to a user or group of users. These permission conditions can affect the rolled-up values for measures at query time. In order for a cube to control the roll-up values for designated members, the PROC OLAP option SECURITY_SUBSET = YES must be set when the cube is built. In addition, users who access the cube must have the necessary permissions to see the members in the roll-up values. If the PROC OLAP option SECURITY_SUBSET = YES is set for a cube, then the rolled-up values will include only those members that the user has permission to see.

When you create MDX queries for security totals, there is no designated MDX code that needs to be written in order to apply security totals to a cube. The only difference between a query written against a cube without the SECURITY_SUBSET option and the same query written against a cube with the SECURITY_SUBSET option is in the values of the output.

Example 1 Applying the SECURITY_SUBSET Option to an MDX Query

Below is an example of an MDX query:

```
SELECT measures
on_columns,
dealers.members on_rows
FROM mddbcars
```

This query has the following applied permission condition:

```
{[dealers].[all dealers],
descendants([dealers].[all dealers].[smith])}
```

Here is the resulting data table if the SECURITY_SUBSET option has not been set.

		MeasuresLevel	sales_sum
All Dealers	dealer	dest	
All Dealers	All Dealers		$229,000.00
	Smith	Smith	$108,000.00
		CA	$12,000.00
		CT	$15,000.00
		NC	$25,000.00
		NJ	$56,000.00

However, if the SECURITY_SUBSET option has been set to YES, then here is the resulting data table:

		MeasuresLevel	sales_sum
All Dealers	dealer	dest	
All Dealers	All Dealers		$108,000.00
	Smith	Smith	$108,000.00
		CA	$12,000.00
		CT	$15,000.00
		NC	$25,000.00
		NJ	$56,000.00

Note that the members displayed in both resulting data tables do not change. This is because both data tables were built with the same permission condition. It is the final value for the All Dealers member that changes from $229,000 in the first table to $108,000 in the second table and shows the sales value of Smith only. The $229,000 in the first table includes sales figures for all dealers.

Example 2 Applying the SECURITY_SUBSET Option to an MDX Query

Here is a second example of an MDX query:

```
SELECT
measures on 0,
date.members on 1
FROM mddbcars
```

This query has the following applied permission condition:

```
{[dealers].[all dealers],
descendants([dealers].[all dealers].[smith])}
```

Here is the resulting data table:

	MeasuresLevel	sales_sum
All Date	dte	
All Date	All Date	$229,000.00
	January	$54,000.00
	February	$64,000.00
	March	$59,000.00
	April	$34,000.00
	May	$18,000.00

However, if the SECURITY_SUBSET option is applied, then the resulting data table is as follows:

	MeasuresLevel	sales_sum
All Date	date	
All Date	**All Date**	$108,000.00
	January	$10,000.00
	February	$25,000.00
	March	$49,000.00
	April	$24,000.00

Note that the members in the output data are the same for both queries. It is the date values that are different. The table values in the first data set reflect sales values for All Dealers, even if the dealers are not displayed. It is when the SECURITY_SUBSET option is applied in the second data table that the sales values reflect only dealer Smith.

Default Member and the All Member

Every dimension for a cube has a default member. That member is implicitly used if no other members of that dimension are explicitly selected in a cube query. In addition, if you do not have permission to see the default member, then the default member will be the first member in the permission condition set. Usually, the All member of a dimension is also the default member.

Note: The All member (parent of the highest level node in the cube) is a system-generated member. It does not have a corresponding column in the underlying data table.

Virtual Members and Security Totals

Virtual members are associated with those records that have missing values in one or more columns. The values associated with virtual members will be included in the roll-up for security totals if you have permission conditions set to see the virtual parent of the virtual member.

For example, here is a sample data set.

```
DISTRICT
REGION
   ACTUAL   BUDGET
.          .             30      5
Atlantic   .             10     35
Atlantic   Eastern US     5      .
Atlantic   Great Britain 12      .
Atlantic   France         8      .
Atlantic   Spain          5      .
Pacific    .             -5     30
Pacific    Western US     8      .
```

```
Pacific   Japan         12      .
Pacific   Korea         10      .
```

In this data set, if the user has a permission conditions to see the following:

```
{[salesregion].[all regions],
[salesregion].[all regions].[atlantic],
[salesregion].[all regions].[atlantic].children}
```

then the value for the [salesregion].[all regions] member would include records from rows 1 through 6.

Here is a possible query of that data:

```
SELECT
    measures on 0,
    salesregion.members on 1
FROM nonleaf
```

Here is the resulting data from that query:

```
                sum of actual
     all regions            70
        atlantic            40
      eastern us             5
   great britain            12
          france             8
           spain             5
```

These are the values that you will see when the permission condition is set and the SECURITY_SUBSET option is set to YES. Note that if the permission condition is set, but the SECURITY_SUBSET option is not set, then the values will be different.

3

MDX Usage Examples

Basic Examples

Basic MDX queries

This topic shows several basic MDX queries. For detailed information about the MDX functions used in these examples see “Basic MDX Queries and Syntax” on page 6 .

The data that is used in these simple examples is from a company that sells various makes and models of cars. The company needs to report sales figures for different months.

Example of a simple two-dimensional query:

```
SELECT
   { [cars].[all cars].[chevy], [cars].[all cars].[ford] } ON COLUMNS,
   { [date].[all date].[march], [date].[all date].[april] } ON ROWS
FROM mddbcars
```

Example of how you can flip the rows and columns:

```
SELECT
   { [cars].[all cars].[chevy], [cars].[all cars].[ford] } ON ROWS,
   { [date].[all date].[march], [date].[all date].[april] } ON COLUMNS
FROM mddbcars
```

Example of selecting a different measure (sales_n) to be the default:

```
SELECT
   { [cars].[all cars].[chevy], [cars].[all cars].[ford] } ON COLUMNS,
   { [date].[all date].[march], [date].[all date].[april] } ON ROWS
FROM mddbcars
WHERE ([measures].[sales_n])
```

Example of using ":" to get a range of members:

```
SELECT
   { [cars].[all cars].[chevy], [cars].[all cars].[ford] } ON COLUMNS,
   { [date].[all date].[january] : [date].[all date].[april] } ON _ROWS
FROM mddbcars
```

Example of the .MEMBERS function:

```
SELECT
   { [cars].[all cars].[chevy], [cars].[all cars].[ford] } ON COLUMNS,
   { [date].members } ON ROWS
FROM mddbcars
```

example of the .CHILDREN function:

```
SELECT
   { [cars].[all cars].[ford].children } ON COLUMNS,
   { [date].members } ON ROWS
FROM mddbcars
```

Example of selecting more than one dimension in a tuple:

```
SELECT
   { ( [cars].[all cars].[chevy], [measures].[sales_sum] ),
     ( [cars].[all cars].[chevy], [measures].[sales_n] ),
     ( [cars].[all cars].[ford], [measures].[sales_sum] ),
     ( [cars].[all cars].[ford], [measures].[sales_n] )
   } ON COLUMNS,
   { [date].members } ON ROWS
FROM mddbcars
```

Example of how the CROSSJOIN function makes tuple combinations for you:

```
SELECT
   { CROSSJOIN ( { [cars].[all cars].[chevy],  [cars].[all cars].[ford] },
                   { [measures].[sales_sum], [measures].[sales_n] }  )
   } ON COLUMNS,
   { [date].members } ON ROWS
FROM mddbcars
```

Example of using the NON_EMPTY keyword to discard the row with no sales:

```
SELECT
   { CROSSJOIN ( { [cars].[all cars].[chevy],  [cars].[all cars].[ford] },
                   { [measures].[sales_sum], [measures].[sales_n] }  )
   } ON COLUMNS,
   NONEMPTY { [date].members } ON ROWS
FROM mddbcars
```

Additional Basic Examples

Example of a basic two-dimension table:

```
SELECT {[time].[all yqm]}  ON
COLUMNS  ,
{[geography].[global].[all global] } ON ROWS
FROM [orionstar]
```

Example of a basic two-dimension table with an analysis variable (Measures):

```
 SELECT {[time].[all yqm]}  ON COLUMNS  ,
{[geography].[global].[all global]} ON ROWS
FROM [orionstar]
WHERE [measures].[total_retail_pricesum]
```

Example of a basic two-dimension table with specific columns selected within the same level:

```
SELECT {[time].[all
yqm].[2001] , [time].[all yqm].[2002]}  ON COLUMNS  ,
{[geography].[global].[all global] } ON ROWS
FROM [orionstar]
WHERE [measures].[total_retail_pricesum]
```

Example of a basic two-dimension table with specific rows selected within the same level:

```
SELECT {[time].[all yqm].[2001] ,
[time].[all yqm].[2002]}  ON COLUMNS  ,
```

```
{[geography].[global].[all global].[europe] ,
[geography].[global].[all global].[asia] } ON ROWS
FROM [orionstar]
WHERE [measures].[total_retail_pricesum]
```

Joins and Extractions for Queries Examples

Example of the CROSSJOIN function:

```
SELECT
{CROSSJOIN({[yqm].[all yqm]},
{[measures].[actualsalessum]})} ON COLUMNS,
{[geography].[all geography]} ON ROWS
FROM [booksnall]
```

Example of the UNION function:

```
SELECT { union([yqm].[all
yqm].[1999],
[yqm].[all yqm].[2000],all)} ON COLUMNS, {
[geography].[all geography] } ON ROWS
FROM [booksnall]
WHERE [measures].[predictedsalessum]
```

Example of the EXCEPT function as a query:

```
SELECT
{except({[yqm].[qtr].members},
{([yqm].[all yqm].[1998].[1]): ([yqm].[all yqm].[2001].[1])})} ON COLUMNS,
{[geography].[all geography]} ON ROWS
FROM
[booksnall]
```

Example of the EXTRACT function:

```
SELECT {extract ({([yqm].[all
yqm].[1998]),
([yqm].[all yqm].[2000])},time )} ON COLUMNS,
{[geography].[all geography]} ON ROWS
FROM [booksnall]
```

Examples of Displaying Multiple Dimensions on Columns and Eliminating Empty Cells

Example of displaying a measure as a column by using the CROSSJOIN function:

```
SELECT
CROSSJOIN ([time].[yqm].[year_id].members ,
[measures].[total_retail_pricesum]) ON COLUMNS,
{[geography].[global].[all global].children } ON ROWS
FROM [orionstar]
```

Example of eliminating empty values by using the NON EMPTY function (in the previous example there are several missing values for Year and Regions):

```
SELECT
NONEMPTY (CROSSJOIN ([time].[yqm].[all yqm].children ,
[measures].[total_retail_pricesum])) ON COLUMNS,
NONEMPTY({[geography].[global].[all global].children }) ON ROWS
FROM [orionstar]
```

Example of using a second CROSSJOIN to combine all three values:

```
SELECT
non_empty(CROSSJOIN(CROSSJOIN ([time].[yqm].[all yqm].children ,
[measures].[total_retail_pricesum]),{[demographics].[all demographics].
female} )) ON COLUMNS,
non_empty({[geography].[global].[all global].children }) ON ROWS
FROM [orionstar]
```

Example of executing the previous step with one function and adding a third set:

Note: The number controls which columns are viewed as well as the crossjoins.

```
SELECT
NONEMPTY CROSSJOIN ([time].[all
yqm].children,[measures].[total_retail_pricesum],
{[demographics].[all demographics].female},3) ON COLUMNS,
NONEMPTY({[geography].[global].[all global].children }) ON ROWS
FROM [orionstar]
```

Example of the COALESCE EMPTY function:

```
WITH MEMBER
[measures].[quantity_nomiss] as 'COALESCE EMPTY
(measures.[quantitysum], 0)'
```

Calculated Member Examples

Example of the WITH MEMBER statement:

```
WITH MEMBER [measures].[target_difference]  AS
    '[measures].[actualsalessum]-[measures].[predictedsalessum]'
  SELECT
  CROSSJOIN([yqm].[all yqm].[2000],
         {[measures].[actualsalessum],
          [measures].[predictedsalessum],
          [measures].[target_difference]}) ON COLUMNS ,

{[geography].[all geography].[mexico],
 [geography].[all geography].[canada]} ON ROWS

FROM [booksnall]
```

Example of the WITH MEMBER statement and Format:

```
WITH MEMBER [measures].[target_difference]  AS
```

```
    '[measures].[actualsalessum]-[measures].[predictedsalessum]' ,
    format_string="dollar20.2"
```

Example of the CREATE GLOBAL MEMBER statement:

```
CREATE GLOBAL MEMBER
[booksnall].[measures].[percentage_increase] AS
'([measures].[actualsalessum] - [measures].[predictedsalessum])/
  [measures].[actualsalessum]',
  format_string="Percent8.2"
```

Example of the DEFINE MEMBER statement:

```
DEFINE MEMBER
[booksnall].[Measures].[Percentage_Increase] AS
   '([Measures].[ActualSalesSUM] - [Measures].[PredictedSalesSUM])/
     [Measures].[ActualSalesSUM]' ,
     format_string="Percent8.2"
```

Example of defining a member with a dimension other than Measures:

```
WITH MEMBER [geography].[all geography].[non usa]
AS
'SUM({[geography].[all geography].[canada],[geography].
[all geography].[mexico]})'

SELECT {CROSSJOIN({[time].[yqm].[all yqm]}, {[measures].
[actualsalessum]})} ON COLUMNS  ,
{[geography].[all geography].[u.s.a], [geography].[all geography].
[non usa]} ON ROWS
FROM
[booksnall]
```

Example of the DROP MEMBER statement:

```
DROP MEMBER
[booksnall].[measures].[percentage_increase]
```

Query-Calculated Member Examples

Example Data

The data that is used in these examples is from a company that sells various makes and models of cars. The company needs to report on sales figures for different months.

Example 1

This query creates a calculation for the average price of a car. The average price of a car is calculated by dividing the sales_sum by the count (sales_n). The query returns the sales_sum, sales_n, and the average price for March and April.

```
WITH
   MEMBER[measures].[avg price] AS
 '[measures].[sales_sum] / [measures].[sales_n]'
SELECT
   { [measures].[sales_sum] , [measures].[sales_n], [measures].[avg price] }
 ON COLUMNS,
   { [date].[all date].[march], [date].[all date].[april] } ON ROWS
FROM mddbcars
```

Here is the resulting output:

Table 3.1 *Query Results*

Date	Sales_sum	Sales_n	Avg Price
March	$59,000.00	4	14750
April	$34,000.00	3	11333.33

Example 2

This query has the same calculation that was created in example 1. This time the calculation is put in the slicer instead of an axis. In this query, the types of cars that were sold are on the column and the months that the cars were sold are on the rows. The value in the cells is the average price of the car for that month.

```
WITH
   MEMBER [measures].[avg price] as '[measures].[sales_sum] /
[measures].[sales_n]'
SELECT
   { [cars].[car].members } ON COLUMNS,
   { [date].members } ON ROWS
FROM mddbcars
WHERE ([measures].[avg price])
```

Here is the resulting output:

Table 3.2 *Query Results*

Date	Chevy	Chrysler	Ford	Toyota
All date	13500	20000	12285.71	8444.45

Date	Chevy	Chrysler	Ford	Toyota
January		20000	10000	8000
February		20000		11000
March	17000		14000	
April	10000		12000	
May			10000	4000

Example 3

This query adds the values of the Chevy, Chrysler, and Ford cars and combines them into one calculation called US. The query shows the sales SUM for the U.S. cars and the Toyota for January through May.

```
WITH
  MEMBER[cars].[all cars].[us] AS '
      SUM( { [cars].[all cars].[chevy],
                 [cars].[all cars].[chrysler],
                 [cars].[all cars].[ford]
               } ) '
SELECT
   { [cars].[all cars].us, [cars].[all cars].toyota } ON COLUMNS,
   { [date].members } ON ROWS
FROM mddbcars
```

Here is the resulting output:

Table 3.3 Query Results

Date	U.S.	Toyota
All Date	$153,000.00	$76,000.00
January	$ 30,000.00	$24,000.00
February	$ 20,000.00	$44,000.00
March	$ 59,000.00	
April	$ 34,000.00	
May	$ 10,000.00	$ 8,000.00

Session-Level Calculated Member Examples

Example Data

The data that is used in these examples is from a company that sells electronics and outdoor and sporting goods equipment.

Example 1

This example creates the session-level calculated member called avg_price in the sales cube on the Measures dimension. This calculated measure shows the average price:

```
create session
    member [sales].[measures].[avg_price] as
      '[Measures].[total] / [Measures].[qty]'
```

Nothing is returned when you create a session-level calculated member.

Example 2

This example uses the session-level calculated member called “avg_price.” It shows the quantity, total, and average price of goods sold from 1998 through 2000.

```
SELECT
    {[measures].[qty], [measures].[total],
 [measures].[avg_price]} ON COLUMNS,
    {[time].[all time].children} ON ROWS
 FROM sales
```

Here is the resulting output:

Table 3.4 *Query Results*

Year	Qty	Total	Average Price
1998	440,852	10,782,352.94	24.4579880322648
1999	539,433	14,080,419.58	26.1022584454418

Year	Qty	Total	Average Price
2000	32,267	859,108.83	26.6249986053863

Example 3

This example uses the session-level calculated member called “avg_price.” It shows the quantity, total, and average price of goods sold in different customer regions.

```
SELECT
    {[measures].[qty], [measures].[total],
 [measures].[avg_price]} ON COLUMNS,
    {[customer].[all customer].children} ON ROWS
 FROM sales
```

Here is the resulting output:

Table 3.5 Query Results

Region	Qty	Total	Average Price
Central	157,659	3,942,290.26	25.0051710336866
Mid-Atlantic	79,555	2,011,008.77	25.2782197222048
Midwest	259,759	6,614,999.09	25.4659091311562
Mountains	32,768	838,064.62	25.5757025146485
Northeast	143,934	3,658,452.99	25.4175732627454
South-Central	64,943	1,662,479.79	25.5990605607995
Southeast	122,888	3,134,589.55	25.5076944046611
West	151,046	3,859,996.28	25.5551042728705

Example 4

This example uses the session-level calculated member called “avg_price.” It shows the quantity, total, and average price of goods sold in the different product groups.

```
SELECT
    {[measures].[qty], [measures].[total],
 [measures].[avg_price]} On COLUMNS,
```

```
    {[product].[all product].children} ON ROWS
 FROM sales
```

Here is the resulting output:

Table 3.6 Query Results

Product	Qty	Total	Average Price
Doing	191,321	4,850,302.26	25.3516459771797
Electronics	330,977	8,426,846.64	25.4605203382712
Health & Fitness	185,909	4,717,790.80	25.3768822380842
Outdoor & Sporting	304,345	7,726,941.65	25.3887583170415

Drill-Down Examples

Example Data

The data that is used in these examples is from a company that sells electronics and outdoor and sporting goods equipment.

Example 1

This example drills down on the electronics and outdoor and sporting goods members from the family level.

```
SELECT
    {[measures].[qty]} on  0,
  drilldownlevel
     (
       {[product].[all product].[electronics],
        [product].[all product].[outdoor & sporting]
       },
        [product].[family]
     ) on 1
FROM sales
```

Here is the resulting output:

Table 3.7 Query Results

Item	Qty
Electronics	330,977
Auto Electronics	13,862
Computers, Peripherals	78,263
Digital Photography	9,008
Home Audio	38,925
Personal Electronics	31,979
Phones	59,964
Portable Audio	27,645
TV, DVD, Video	47,725
Video Games	23,606
Outdoor & Sporting	304,345
Bikes, Scooters	45,297
Camping, Hiking	63,362
Exercise, Fitness	50,700
Golf	41,467
Outdoor Gear	52,305
Sports Equipment	51,214

Example 2

This example drills down on the electronics and outdoor and sporting goods members to the family level, but it shows only the top two members at each level based on the value of Qty.

```
SELECT
   {[measures].[qty]} on 0,
  drilldownleveltop
    (
```

```
      {[product].[all product].[electronics],
       [product].[all product].[outdoor & sporting]
      },
      2,
      [product].[family],
      [measures].[qty]
    ) on 1
FROM sales
```

Here is the resulting output:

Table 3.8 Query Results

Item	Qty
Electronics	330,977
Computers, Peripherals	78,263
Phones	59,964
Outdoor & Sporting	304,345
Camping, Hiking	63,362
Outdoor Gear	52,305

Example 3

This example drills down on the electronics and outdoor and sporting goods members to the family level, but it shows only the bottom two members at each level based on the value of Qty.

```
SELECT
   {[measures].[qty]} on 0,
  drilldownlevelbottom
    (
      {[product].[all product].[electronics],
       [product].[all product].[outdoor & sporting]
      },
      2,
      [product].[family],
      [measures].[qty]
    ) on 1
FROM sales
```

Here is the resulting output:

Table 3.9 Query Results

Item	Qty
Electronics	330,977
Digital Photography	9,008
Auto Electronics	13,862
Outdoor & Sporting	304,345
Golf	41,467
Bikes, Scooters	45,297

Example 4

This example drills up to the members of the set that are below the category level. It returns only those members that are at the category level or higher.

```
SELECT
   {[measures].[qty]} on 0,
  drilluplevel
    (
      {[product].[all product].[electronics].[computers, peripherals],
       [product].[all product].[electronics].[tv, dvd, video],
       [product].[all product].[electronics].[video games].[gameplace],
       [product].[all product].[electronics].[video games].[play guy
color].[caller],
       [product].[all product].[outdoor & sporting],
       [product].[all product].[outdoor & sporting].[bikes, scooters].[kids'
bikes],
       [product].[all product].[outdoor &
sporting].[golf].[clubs].[designed],
       [product].[all product].[outdoor & sporting].[sports equipment],
       [product].[all product].[outdoor & sporting].[sports
equipment].[baseball]
      },
      [product].[category]
    ) on 1
FROM sales
```

Here is the resulting output:

Table 3.10 Query Results

Item	Qty
Computers, Peripherals	78,263

Item	Qty
TV, DVD, Video	47,725
Outdoor & Sporting	304,345
Sports Equipment	51,214

Session-Named Set Examples

Example Data

The data that is used in these examples is from a company that sells electronics and outdoor and sporting goods equipment.

Example 1

This example creates the session-named set called “prod in SE” in the sales cube. This named set shows the crossing of the product family with the customer members in the Southeast.

```
CREATE
SESSION
   set sales.[prod in se] as '
    CROSSJOIN
      (
        [customer].[all customer].[southeast].children,
        [product].[family].members
      )'
```

Nothing is returned when you create a session-named set.

Example 2

This example creates the session-named set called “prod in NE” in the sales cube. This named set shows the crossing of the product family with the customer members in the Northeast.

```
CREATE
SESSION
   set sales.[prod in ne] as '
```

```
CROSSJOIN
  (
    [customer].[all customer].[northeast].children,
    [product].[family].members
  )'
```

Nothing is returned when you create a session-level named set.

Example 3

This example uses the session-named set called “prod in SE.” It shows the quantity and total sales for products that customers in the Southeast purchased.

```
SELECT
    {[measures].[qty], [measures].[total]} ON COLUMNS,
    [prod in se] ON ROWS
FROM sales
```

Here is the resulting output:

Table 3.11 *Query Results*

State	Product	Qty	Total
FL	Doing	21,091	550,672.41
FL	Electronics	31,056	794,730.61
FL	Health & Fitness	16,321	415,708.57
FL	Outdoor & Sporting	30,065	742,907.85
GA	Doing	1,907	44,360.08
GA	Electronics	2,316	61,577.03
GA	Health & Fitness	1,318	35,589.84
GA	Outdoor & Sporting	2,458	68,438.03
NC	Doing	235	5,404.65
NC	Electronics	3,727	101,688.42
NC	Health & Fitness	1,228	31,310.45
NC	Outdoor & Sporting	835	21,312.83
SC	Doing 1	,266	31,596.69
SC	Electronics	2,646	66,565.97

State	Product	Qty	Total
SC	Health & Fitness	3,483	89,633.82
SC	Outdoor & Sporting	2,936	73,092.30

Example 4

This example uses the session-named set called “prod in NE.” It shows the quantity and total sales for products that customers in the Northeast purchased.

```
SELECT
   {[measures].[qty], [measures].[total]} ON COLUMNS,
   [prod in ne] ON ROWS
 FROM sales
```

Here is the resulting output:

Table 3.12 Query Results

State	Product	Qty	Total
CT	Doing	844	20,961.12
CT	Electronics	2,659	69,540.52
CT	Health & Fitness	969	22,995.63
CT	Outdoor & Sporting	2,569	61,528.35
MA	Doing	7,918	206,472.36
MA	Electronics	11,184	281,371.34
MA	Health & Fitness	4,339	105,356.59
MA	Outdoor & Sporting	10,076	250,323.21
ME	Doing	1,362	35,151.55
ME	Electronics	4,496	110,153.94
ME	Health & Fitness	2,218	58,342.02
ME	Outdoor & Sporting	3,014	79,426.68
NH	Doing	141	4,207.76

State	Product	Qty	Total
NH	Electronics	466	10,750.48
NH	Health & Fitness	1,095	26,158.29
NH	Outdoor & Sporting	603	14,893.73
NY	Doing	17,493	435,513.26
NY	Electronics	29,246	759,166.44
NY	Health & Fitness	13,880	347,481.77
NY	Outdoor & Sporting	26,714	692,416.36
RI	Doing	265	6,437.18
RI	Electronics	833	22,723.54
RI	Health & Fitness	693	17,760.85
RI	Outdoor & Sporting	857	19,320.02

Example 5

This example uses both of the session-named sets called “prod in NE” and “prod in SE”. It shows the quantity and total sales for products that customers in the Northeast and the Southeast purchased.

```
SELECT
    {[measures].[qty], [measures].[total]} ON COLUMNS,
    {[prod in ne], [prod in se]} ON ROWS
 FROM sales
```

Here is the resulting output:

Table 3.13 Query Results

State	Product	Qty	Total
CT	Doing	844	20,961.12
CT	Electronics	2,659	69,540.52
CT	Health & Fitness	969	22,995.63
CT	Outdoor & Sporting	2,569	61,528.35

State	Product	Qty	Total
MA	Doing	7,918	206,472.36
MA	Electronics	11,184	281,371.34
MA	Health & Fitness	4,339	105,356.59
MA	Outdoor & Sporting	10,076	250,323.21
ME	Doing	1,362	35,151.55
ME	Electronics	4,496	110,153.94
ME	Health & Fitness	2,218	58,342.02
ME	Outdoor & Sporting	3,014	79,426.68
NH	Doing	141	4,207.76
NH	Electronics	466	10,750.48
NH	Health & Fitness	1,095	26,158.29
NH	Outdoor & Sporting	603	14,893.73
NY	Doing	17,493	435,513.26
NY	Electronics	29,246	759,166.44
NY	Health & Fitness	13,880	347,481.77
NY	Outdoor & Sporting	26,714	692,416.36
RI	Doing	265	6,437.18
RI	Electronics	833	22,723.54
RI	Health & Fitness	693	17,760.85
RI	Outdoor & Sporting	857	19,320.02
FL	Doing	21,091	550,672.41
FL	Electronics	31,056	794,730.61
FL	Health & Fitness	16,321	415,708.57
FL	Outdoor & Sporting	30,065	742,907.85

State	Product	Qty	Total
GA	Doing	1,907	44,360.08
GA	Electronics	2,316	61,577.03
GA	Health & Fitness	1,318	35,589.84
GA	Outdoor & Sporting	2,458	68,438.03
NC	Doing	235	5,404.65
NC	Electronics	3,727	101,688.42
NC	Health & Fitness	1,228	31,310.45
NC	Outdoor & Sporting	835	21,312.83
SC	Doing	1,266	31,596.69
SC	Electronics	2,646	66,565.97
SC	Health & Fitness	3,483	89,633.82
SC	Outdoor & Sporting	2,936	73,092.30

Example 6

This example removes (drops) the session-named set called “prod in SE” in the sales cube.

```
DROP SET
sales.[prod in SE]
```

Nothing is returned when you drop a session-named set.

Example 7

This example removes (drops) the session-named set called “prod in NE” in the sales cube.

```
DROP SET
[sales].[prod in NE]
```

Nothing is returned when you drop a session-named set.

Additional Named Set Examples

Example of a session set using SQL pass-through and CREATE SET:

```
proc sql;
  connect to olap (host=host-name port=port-number
          user="userid" pass="password");
      execute
        (
        create set booksnall.threeyears as
        {[YQM].[All YQM].[1999] :[YQM].[All YQM].[2001]}
   ) by olap;
      create table temp as select * from connection to olap
(
   SELECT  threeyears ON COLUMNS  ,
       {[products].[all products].[books]} ON ROWS
       FROM [booksnall]
   );
   disconnect from olap;
quit;
```

Example of how to drop a set by using DROP SET:

```
proc sql;
connect to olap (host=host-name port=port-number user="userid"

                 pass="password");
execute

(
   DROP SET booksnall.threeyears
) by olap;

disconnect from olap;
quit;
```

4

MDX Functions

Dimension Functions

The MDX functions that are listed here indicate their return type.

Dimension
: returns a dimension that contains a specified member, level, or hierarchy.`<Member>.Dimension<Level>.Dimension<Hierarchy>.Dimension`

Dimensions
: returns a dimension that is specified by a numeric or string expression.`Dimensions(<Numeric Expression>)Dimensions(<String Expression>)`

Hierarchy Functions

The MDX functions that are listed here indicate their return type.

Hierarchy
: returns a hierarchy that contains a specified member or level.
`<Member>.Hierarchy<Level>.Hierarchy`

Level Functions

The MDX functions that are listed here indicate their return type.

Level
: returns the level of a member. `<Member>.Level`

Levels
: returns levels that are specified by a numeric or string expression.
`<Dimension>.Levels(<NumericExpression>)Levels(<StringExpression>)`

Logical Functions

The MDX functions that are listed here indicate their return type.

IsEmpty
: returns TRUE if the evaluated expression is an empty cell value. Otherwise, FALSE is returned. `IsEmpty(<Value Expression>)`

IS
: returns TRUE if two compared objects are equivalent. Otherwise, FALSE is returned. `<Object 1>IS Null<Object 1>IS <Object 2>`

IsAncestor
: returns TRUE if a specified member is an ancestor of another specified member. Otherwise, FALSE is returned. `IsAncestor(<Member1>,<Member2>)`

IsLeaf
: returns TRUE if a specified member is a leaf member. Otherwise, FALSE is returned. `IsLeaf(<Member>)`

IsSibling
: returns TRUE if a specified member is a sibling of another specified member. Otherwise, FALSE is returned. `IsSibling(<Member1>,<Member2>)`

Member Functions

The MDX functions that are listed here indicate their return type.

Ancestor
: returns the ancestor of a member at a specified level or distance.`Ancestor(<Member>,<Level>)Ancestor(<Member>,<Numeric Expression>)`

ClosingPeriod
: returns the last sibling among the descendants of a member to a specified level. `ClosingPeriod([<Level>[,<Member>]])`

Cousin
: returns the child member with the same relative position under its parent member as the specified child member. `Cousin (<Member1>,<Member2>)`

CurrentMember
: returns the current member of a dimension or hierarchy during an iteration over a set of members of that dimension or hierarchy. `<Dimension>.CurrentMember<Hierarchy>.CurrentMember`

DataMember
: returns a system-generated data member that is associated with a non-leaf member of a dimension. `<Member>.DataMember`

DefaultMember
: returns the default member of a dimension or hierarchy. `<Dimension>.DefaultMember<Hierarchy>.DefaultMember`

FirstChild
: returns the first child of a specified member. `<Member>.FirstChild`

FirstSibling
: returns the first child of the parent of a specified member. `<Member>.FirstSibling`

Item
: returns a member from a specified tuple. Alternatively, it returns a tuple from a set. `<Tuple>.Item(<Index>)`

Note: If a tuple is returned, then it is a tuple function, not a member function.

Lag
: returns a member that is located at a specified number of positions before a designated member at the same level as that member. `<Member>.Lag(<Numeric Expression>)`

LastChild
: returns the last child of a specified member. `<Member>.LastChild`

LastSibling
: returns the last child of the parent of a specified member. `<Member>.LastSibling`

Lead
returns a member that is located at a specified number of positions before a designated member at the same level as that member.
`<Member>.Lead(<Numeric Expression>)`

NextMember
returns the next member of the level that contains the specified member.
`<Member>.NextMember`

OpeningPeriod
returns the first sibling among the descendants of a specified member at the specified level. `OpeningPeriod([<Level>[,<Member>]])`

ParallelPeriod
returns a member at the level of the specified member that is in the same relative position under its ancestor at the specified level.`ParallelPeriod([<Level>[,Numeric Expression>[,<Member>]]])`

Parent
returns the parent of a member. `<Member>.Parent`

PrevMember
returns the previous member at the level of the specified member.
`<Member>.PrevMember`

StrToMember
returns a member from a string expression in Multidimensional Expressions (MDX) format. `StrToMember(<String Expression>)`

Numeric Functions

The MDX functions that are listed here indicate their return type.

Aggregate
returns a calculated value by using the appropriate aggregate function, which is based on the aggregation type of the member. `Aggregate(<Set[,<Numeric Expression>])`

Avg
returns the average value of a numeric expression that is evaluated over a set. `Avg(<Set>[,<Numeric Expression>])` Example: The following example shows a moving average across all dimensions of time.`Avg(time.currentmember.lag (if(time.currentmember.level is time.month_num,2, if(time.currentmember.level is time.quarter,1,0))) :time.currentmember, measures. [total_retail_pricesum])` The Total_Retail_PriceSUM is included in the following query to see the difference between the moving average and the total retail price.`SELECT {[measures].[movingaverage],[measures]. [total_retail_pricesum] } ON COLUMNS , {[time].[yqm].[all yqm].children } ON ROWS FROM [orionstar]`

CoalesceEmpty
returns a coalesced value. This value is derived when an empty cell value is coalesced to a number or string. `CoalesceEmpty(<Numeric Expression>[,<Numeric Expression>])`

Correlation
returns the correlation of two series that are evaluated over a set.
`Correlation(<Set>,<Numeric Expression>[,<Numeric Expression>])`

Count
depending on the collection, returns the number of items in a collection.
`<Dimension>|<Hierarchy>.Levels.Count<Tuple>.Count<Set>.CountCount(<Set>[,ExcludeEmpty | IncludeEmpty])`

Covariance
returns the population covariance of two series that are evaluated over a set by using the biased population formula.`Covariance(<Set>,<Numeric Expression>[,<Numeric Expression>])`

CovarianceN
returns the sample covariance of two series that are evaluated over a set by using the unbiased population formula. `CovarianceN(<Set>,<Numeric Expression>[,<Numeric Expression>])`

DistinctCount
returns the number of distinct, non-empty tuples in a set. `DistinctCount(<Set>)`

IIf
returns one of two numeric or string values that are determined by a logical test.
`IIF(<Logical Expression>, <Numeric Expression1>, <Numeric Expression2>)`

Note: If a string is returned, then it is a string function, not a numeric function.

LinRegIntercept
calculates the linear regression of a set and returns the value of b in the regression line y = ax + b. `LinRegIntercept(<Set>,<Numeric Expression>[,<NumericExpression>])`

LinRegPoint
calculates the linear regression of a set and returns the value of y in the regression line y = ax + b.
`LinRegPoint(<NumericExpression>,<Set>,<NumericExpression>[,<Numeric Expression>])`

LinRegR2
calculates the linear regression of a set and returns R^2 (the coefficient of determination).`(Set, Numeric Expression[, Numeric Expression])`

LinRegSlope
calculates the linear regression of a set and returns the value of a in the regression line y = ax + b.
`LinRegSlope(<Set>,<NumericExpression>[,<NumericExpression>])`

LinRegVariance
calculates the linear regression of a set and returns the variance associated with the regression line y = ax + b. `(Set, Numeric Expression[, Numeric Expression])`

Max
returns the maximum value of a numeric expression that is evaluated over a set.
`Max(<Set>[,<Numeric Expression>])`

Median
: returns the median value of a numeric expression that is evaluated over a set. `Median(<Set>[,<Numeric Expression>])`

Min
: returns the minimum value of a numeric expression that is evaluated over a set. `Min(<Set>[,<Numeric Expression>])`

Ordinal
: returns the zero-based ordinal value that is associated with a level. `<Level>.Ordinal`

Range
: returns the range, which is the difference between the maximum and minimum value of a numeric expression that is evaluated over a set. `Range (<Set>[,<Numeric Expression>])`

Rank
: returns the one-based rank of a specified tuple in a specified set. `Rank(<Tuple>,<set>[,<Calc Expression>])`

RollupChildren
: returns a value that is generated by rolling up the values of the children of a specified member by using the specified unary operator. `RollupChildren(<Member>,<String Expression>)`

Stdev
: using the unbiased population formula, returns the sample standard deviation of a numeric expression that is evaluated over a set. `Stdev(<set>[,<Numeric Expression>])`

StdevP
: using the biased population formula, returns the population standard deviation of a numeric expression that is evaluated over a set. `StdevP(<set>[,<Numeric Expression>])`

StrToValue
: returns a value from a string expression. `StrToValue(<StringExpression>)`

Sum
: returns the sum of a numeric expression that is evaluated over a set. `Sum(<Set>[,<Numeric Expression>])`

Value
: returns the value of a measure. `<Member>.Value`

Var
: using the unbiased population formula, returns the sample variance of a numeric expression that is evaluated over a set. `Var(<Set>[,<Numeric Expression>])`

VarP
: using the biased population formula, returns the population variance of a numeric expression that is evaluated over a set. `VarP(<Set>[,<Numeric Expression>])`

Set Functions

The MDX functions that are listed here indicate their return type.

AddCalculatedMembers
returns a set that includes calculated members that meet the criteria of a given set definition (by default, calculated members are not returned by set functions). `AddCalculatedMembers(<Set>)` Example: `WITH MEMBER [geography].[geography].[all geography].[u.s.a]. [north u.s.a] AS ' SUM({[geography].[geography].[all geography].[u.s.a].[north central], [geography].[geography].[all geography].[u.s.a].[north east], [geography].[geography].[all geography].[u.s.a].[north west]})' SELECT [Measures].ActualSalesSUM ON COLUMNS, AddCalculatedMembers({[geography].[geography].[All geography].[u.s.a].[north central]}) ON ROWS FROM [booksnall]`

AllMembers
returns a set that contains all members of the specified dimension, hierarchy, or level, including calculated members. `<Dimension>.AllMembers<Hierarchy>.AllMembers<Level>.AllMembers` The following example references all levels and members below a specific level using AllMembers Function. You can include calculated Members (set function). `SELECT {[time].AllMembers} ON COLUMNS , {[geography].[global].[all global].[europe]} ON ROWS FROM [orionstar] WHERE [measures].[total_retail_pricesum] SELECT {[measures].AllMembers} ON COLUMNS , [geography].[global].[all global].[europe] on ROWS FROM [orionstar]`

Ancestors
returns the set of ancestors of a member to a specified level or distance. This includes or excludes ancestors at other levels. Here is the syntax for the Ancestors function: `Ancestors(<member>, [<level>[,<anc_flags>]])` `Ancestors(<member>,<distance>[,<anc_flags>])` The following example shows retrieving the Ancestor at a particular level: `WITH MEMBER [measures].[product family sales] AS '(ancestor([product].currentmember,[product]. [product family]), [measures].[unit sales])' SELECT {[measures].[unit sales], [measures].[product family sales]} ON COLUMNS, {[product].members} ON ROWS FROM [sales]`

Level
returns the set of ancestors of a member that are specified by *<Member>* to the level that is specified by *<Level>*. Optionally, the set is modified by a flag that is specified in *<Anc_flags>*. `Ancestors(<member>,[<level>[, <Anc_flags>]])`

If no *<Level>* or *<Anc_flags>* arguments are specified, then the function behaves as in the following syntax: `Ancestors(<member>, <member>.Level, SELF_BEFORE_AFTER)`

Distance
returns the set of ancestors of a member. The set of ancestors is specified by *<member>* and is *<distance>* steps away in the hierarchy. Optionally, the set is modified by a flag that is specified in *<Anc_flags>*. Specifying a *<Distance>*

of 0 returns a set consisting only of the member that is specified in *<Member>*.`Ancestors(<member>, <distance>[,<Anc_flags>])`

Table 4.1 *Ancestor Flag Options*

Options	Value Returned
AFTER	Returns ancestor members from all levels between *<Level>* and *<Member>*, including *<Member>* itself, but not member(s) found at *<Level>.*
BEFORE	Returns ancestor members from all levels above *<Level>.*
BEFORE_AND_AFTER	Returns ancestor members from all levels above the level of *<Member>* except members from *<Level>.*
ROOT	Returns the root-level member. This flag is the opposite of the LEAVES flag for the Descendants function.
SELF (default)	Returns ancestor members from *<Level>* only. Includes *<Member>*, if and only if *<Level>* that is specified is the level of *<Member>.*
SELF_AND_AFTER	Returns ancestor members from *<Level>* and all levels below *<Level>*, down to and including *<Member>.*
SELF_AND_BEFORE	Returns ancestor members from *<Level>* and all levels between and above *<Member>.*
SELF_BEFORE_AFTER	Returns ancestor members from all levels above the level of *<Member>*, including *<Member>* and member(s) at *<Level>.*

Note: By default, only members at the specified level or distance are included. This function corresponds to an *<Anc_flags>* value of SELF. By changing the value of *<Anc_flags>*, you can include or exclude ancestors at the specified level or distance, the ancestors before or the ancestors after the specified level or distance (until the root node), as well as all requests of the root ancestor or ancestors regardless of the specified level or distance.

Assuming that the levels in the Location dimension are named in a hierarchical order, an example of levels would be All, Countries, States, Counties, and Cities.

Table 4.2 Ancestor Expressions and Returned Values

Expressions	Value Returned
Ancestors (USA)	All members
Ancestors (Wake, Counties)	USA
Ancestors (Wake, Counties, SELF)	USA
Ancestors (Wake, States, BEFORE)	USA, All
Ancestors (Wake, Counties, AFTER)	Wake (includes member itself), North Carolina
Ancestors (Raleigh, States, BEFORE_AND_AFTER)	Raleigh, Wake, USA, All members
Ancestors (Raleigh, States, SELF_BEFORE_AFTER)	Raleigh, Wake, NC, USA, All members
Ancestors (NC, Counties, Root)	All members
Ancestors (Wake, 1)	North Carolina
Ancestors (Wake, 2, SELF_BEFORE_AFTER)	Wake, NC, USA, All members

Ascendants

returns all ancestors of the specified member up through the root level, including the member itself. `Ascendants (<Member>)` Example: The following example shows retrieving the member at the specific level above and the current member dynamically using the Ascendants function (set function). `SELECT {Ascendants([time].[all yqm] .[2002])} ON COLUMNS , {Descendants([geography].[global].[all global],2)} ON ROWS FROM [orionstar] WHERE [measures].[total_retail_pricesum]`

Axis

returns a set that is defined in an axis. Axis (0) pertains to row members, where Axis (1) pertains to column members.`Axis (<Numeric Expression>)` Example: `Axis (0) Axis (1)`

Note: The Axis function is not allowed in session- or global-named sets or calculations.

BottomCount

returns a specified number of items from the bottom of a set.
`BottomCount(<Set>,<Count>[,Numeric Expression>[,<True|False>]])`

Example: The following example shows displaying the bottom 5 products in the Clothes product category for 2002Q4 using the BottomCount function.`SELECT {[time].[yqm].[all yqm].[2002].[2002q4] } ON COLUMNS , {BottomCount([product].[all product].[clothes & shoes].[clothes].`

```
children ,5,[measures].[total_retail_pricesum] ) } ON ROWS FROM
[orionstar] WHERE [measures].[Total_retail_pricesum]
```

Note: The True|False flag is for including duplicates. If it is set to TRUE, then any member that has the same value as the last member will also be returned. If it is set to FALSE, then it will work as it always did. The default value for the flag is FALSE.

Note: Constant numeric expressions should not be entered for this function.

BottomPercent

sorts a set and returns the specified number of bottommost elements whose cumulative total is at least a specified percentage.

```
(<Set>,<Percentage>[,<Numeric Expression>])
```

Example: The following example shows displaying the bottom 25% of Clothes Items in the Product Category for 2002 using the BottomPercent function.

```
SELECT
{[time].[yqm].[all yqm].[2002] } ON COLUMNS ,
{BottomPercent([product].[all product].[clothes & shoes].
[clothes].children ,25,[measures].[total_retail_pricesum] ) } ON
ROWS FROM [orionstar] WHERE [measures].[total_retail_pricesum]
```

Note: Constant numeric expressions should not be entered for this function.

BottomSum

sorts a set by using a numeric expression and returns the specified number of bottommost elements whose sum is at least a specified value.

```
(<Set>,<Value>[,<Numeric Expression>])
```

Example: The following example shows obtaining the items that have a cumulative total below the specified amount using BottomSum function.

```
SELECT
{[time].[yqm].[all yqm].[2002] } ON COLUMNS , {BottomSum([product].
[all product].[clothes & shoes]. [clothes].children ,6000000,
[measures].[total_retail_pricesum] ) } ON ROWS FROM [orionstar]
WHERE [measures].[total_retail_pricesum] WHERE [measures].
[total_retail_pricesum]
```

Note: Constant numeric expressions should not be entered for this function.

Children

returns the children of a member. `<Member>.Children`

Crossjoin

returns the cross-product of two sets. `Crossjoin(<Set1>,<Set2>)`

Descendants

returns the set of descendants of a member to a specified level or distance. Optionally, this includes or excludes descendants at other levels. By default, only members at the specified level or distance are included.

```
Descendants(<Member>,
[<Level>[,<Desc_flags>]])Descendants(<Member>,<Distance>[,<Desc_fla
gs>])
```

Table 4.3 Descendants Flag Options

Options	Value Returned
AFTER	Returns descendant members from all levels that are subordinate to *<Level>.*
BEFORE	Returns descendant members from all levels between *<Member>* and *<Level>*, not including members from *<Level>.*
BEFORE_AND_AFTER	Returns descendant members from all levels that are subordinate to the level of *<Member>,* except members from *<Level>.*
LEAVES	Returns leaf descendant members between *<Member>* and *<Level>* or *<Distance>*. This flag is the opposite of the ROOT flag for the Ancestors function.
SELF (default)	Returns descendant members from *<Level>* only. Includes *<Member>*, only if *<Level>* is specified at the level of *<Member>.*
SELF_AND_AFTER	Returns descendant members from *<Level>* and all levels subordinate to *<Level>.*
SELF_AND_BEFORE	Returns descendant members from *<Level>* and all levels between *<Member>* and *<Level>.*
SELF_BEFORE_AFTER	Returns descendant members from all levels that are subordinate to the level of *<Member>.*

Distinct
: returns a set by removing duplicate tuples from a specified set. Duplicates are eliminated from the tail.`Distinct(<Set>)`

DrilldownLevel
: drills down to the members of a set one level below the lowest level that is represented in the set, or to one level below an optionally specified level of a member that is represented in the set. `DrilldownLevel(<Set>[, {<Level>|,<Index}])`

DrilldownLevelBottom
: drills down the members of a set to one level below the lowest level that is represented in the set, or to one level below an optionally specified level of a member that is represented in the set. However, instead of including all children for each member at the specified <Level>, only the bottom <Count> of children is returned, based on <Numeric

Expression>.`DrilldownLevelBottom(<Set>,<Count>[,[<Level>] [,<Numeric Expression>]])`

Note: Constant numeric expressions should not be entered for this function.

DrilldownLevelTop
drills down the members of a set to one level below the lowest level that is represented in the set, or to one level below an optionally specified level of a member that is represented in the set. However, instead of including all children for each member at the specified <Level>, only the top <Count> of children is returned, based on <Numeric Expression>.`DrilldownLevelTop(<Set>,<Count>[,[<Level>] [,<Numeric Expression>]])`

Note: Constant numeric expressions should not be entered for this function.

DrilldownMember
drills down to the members in a specified set that are present in a second specified set. `DrilldownMember(<Set1>,<Set2[,Recursive])`

DrilldownMemberBottom
drills down to the members in a specified set that are present in a second specified set, therefore limiting the result set to a specified number of members. `DrilldownMemberBottom(<Set1>,<Set2>, <Count>[, [<Numeric Expression>] [,Recursive]])`

Note: Constant numeric expressions should not be entered for this function.

DrilldownMemberTop
drills to the members in a specified set that are present in a second specified set, therefore limiting the result set to a specified number of members. `DrilldownMemberTop(<Set1>,<Set2>, <Count>[, [<Numeric Expression>] [,Recursive]])`

Note: Constant numeric expressions should not be entered for this function.

DrillupLevel
removes all members in the set that are below the specified level. If the level is not given, then it determines the lowest level in the set and removes all members at that level. `DrillupLevel(<Set>[,<Level>])`

DrillupMember
drills to the members in a specified set that are present in a second specified set. `DrillupMember(<Set1>,<Set2>)`

Except
locates the difference between two sets and optionally retains duplicates. `Except(<Set1>,<Set2>[,All])`

Exists
returns the members of set 1 that exist. Set 1 is mandatory. You can also have a second set or a third set (the measures set). The measures set provides a set of measures to be used to evaluate the members of set 1. In addition, each argument should be a set.

`Exists(<Set1>[,<Set2>[,<MeasureSet>]])`

If set 2 is present, the Exists function returns the members in set 1 that are also present in set 2. If set 2 does not exist, the Exists function returns the matching members of set 1, based on the selection criteria in set 3 (the measures set).

If set 3 (the measures set) is present, the Exists function matches the specified measures of set 1 to set 3.

Note: With the Exists function, members with missing data will be kept.

Extract
: returns a set of tuples from extracted dimension elements.
`Extract(<Set>,<Dimension>[,<Dimension>...])`

Filter
: returns the set that results from filtering a specified set that is based on a search condition. `Filter(<Set>,<Search Condition>)`

Generate
: applies a set to each member of another set and is joined to the resulting sets.
`Generate(<Set1>,<Set2>[,All])`

Head
: returns the first specified number of elements in a set. `Head(<Set>[,<Numeric Expression>])`

Hierarchize
: orders the members of a set in a hierarchy. `Hierarchize(<Set>)`

Intersect
: returns the intersection of two input sets and optionally retains duplicates.
`Intersect(<Set1>,<Set2>[,All])`

LastPeriods
: returns a set of members prior to and including a specified member.
`LastPeriods(<Index>[,<Member>])`

Members
: returns the set of members in a dimension, level, or hierarchy.
`<Dimension>.Members<Level>.Members<Hierarchy>.Members`

Mtd
: returns the set of members that consist of the descendants of the Month level ancestor of the specified member, including the specified member itself. This function is analogous to the PeriodsToDate() function with the level defined as Month. `Mtd([<Member>])`

NameToSet
: returns a set that contains a single member. The set is based on a string expression that contains a member name. `NameToSet(<Member Name>)`

NonEmptyCrossjoin
: returns the cross-product of one or more sets as a set. This excludes empty tuples and tuples without associated fact table data.
`NonEmptyCrossjoin(<Set1>[,<Set2>][,<Set3>...] [,<Crossjoin Set Count>])`

Order
: arranges members of a specified set and optionally preserves or breaks the hierarchy. `Order(<Set>[,[<Numeric Expression>][,BASC|BDESC]])Order(<Set>[,[<String Expression>][,BASC|`

```
BDESC]])Order(<Set>,<Numeric Expression>[,ASC|
DESC])Order(<Set>,<String Expression>[,ASC|DESC])
```

Note: Constant numeric expressions should not be entered for this function.

PeriodsToDate
returns the set of members that consist of the descendants of the ancestor of the specified member at the specified level, including the specified member itself. `PeriodsToDate([<Level>[,<Member>]])`

Qtd
returns the set of members that consist of the descendants of the Quarter level ancestor of the specified member, including the specified member itself. This function is analogous to the PeriodsToDate() function with the level defined as Quarter. `Qtd([<Member>])`

Siblings
returns the siblings of a specified member, including the member itself. `<Member>.Siblings`

StripCalculatedMembers
returns a set that is generated by removing calculated members from a specified set. `StripCalculatedMembers(<Set>)`

StrToSet
returns a set that is constructed from a specified string expression in Multidimensional Expressions (MDX) format. `StrToSet (<String Expression>)`

Subset
returns a subset of tuples from a specified set. `Subset(<Set>,<Start>[,<Count>])`

Tail
returns a subset from the end of a set. `Tail(<Set>[,<Count>])`

ToggleDrillState
Toggles the drill state of members. `ToggleDrillState(<Set1>,<Set2>[,RECURSIVE])`

Note: In a graphical user interface, drilling up and down is often accomplished by double-clicking a label to expand or contract the information. Drilling down on a member causes the member's children to be returned; drilling up causes them to disappear from the results.

TopCount
returns a specified number of items from the topmost members of a specified set. `TopCount(<Set>,<Count>[,<Numeric Expression>[,<True|False>]])`

Note: The True|False flag is for including duplicates. If it is set to TRUE, then any member that has the same value as the last member will also be returned. If it is set to FALSE, then it will work as it always did. The default value for the flag is FALSE.

Note: Constant numeric expressions should not be entered for this function.

TopPercent
sorts a set and returns the topmost elements, whose cumulative total is at least a specified percentage. `TopPercent(<Set>,<Percentage>[,<Numeric Expression>])`

Note: Constant numeric expressions should not be entered for this function.

TopSum
sorts a set and returns the topmost elements whose cumulative total is at least a specified value. `TopSum(<Set>,<Value>[,<Numeric Expression>])`

Note: Constant numeric expressions should not be entered for this function.

Union
returns a set that is generated by the union of two sets. Optionally, duplicate members are retained. `Union(<Set1>,<Set2>[,All])`

VisualTotals
returns a set that is generated by dynamically totaling child members in a specified set. A pattern for the name of the parent member in the result set is used. `VisualTotals (<Set>,<Pattern>)`

Wtd
returns the set of members that consist of the descendants of the Week level ancestor of the specified member, including the specified member itself. This function is analogous to the PeriodsToDate() function with the level defined as Week. `Wtd([<Member>])`

Ytd
returns the set of members that consist of the descendants of the Year level ancestor of the specified member, including the specified member itself. This function is analogous to the PeriodsToDate() function with the level defined as Year. `Ytd([<Member>])`

String Functions

The MDX functions that are listed here indicate their return type.

CoalesceEmpty
coalesces an empty cell value to a number or string and returns the coalesced value. `CoalesceEmpty(<String Expression>[,<String Expression>]...)`

Generate
returns a concatenated string that is created by evaluating a string expression over a set. `Generate(<Set>,<String Expression>[,Delimiter>])`

IIf
returns one of two numeric or string values that are determined by a logical test. `IIf(<Logical Expression>,<String Expression1>, <String Expression2>)`

Note: If a numeric value is returned, then it is a numeric function, not a string function.

INSTR
searches a string expression for another string and returns the first occurrence of that string. `InStr([start, ]searched_string, search_string[, compare])`

The compare argument is an optional integer value. This argument is always ignored. It is defined for compatibility with other INSTR functions in other languages.

MemberToStr
returns a string in Multidimensional Expressions (MDX) format from a member. `MemberToStr(<Member>)`

Name
returns the name of a level, dimension, member, or hierarchy. `<Level>.Name<Dimension>.Name<Member>.Name<Hierarchy>.Name`

Properties
returns a string that contains a member property value. `<Member>.Properties(Caption)<Member>.Properties(Name)<Member>.Properties(UniqueName)<Member>.Properties(<String Expression>,<TRUE | FALSE>)`

Note: The raw data associated with the property can be either numeric or character, depending on the property type. If the parameter is set to TRUE, then the function returns the raw value for the property instead of the formatted value. If the parameter is set to FALSE, then the function returns the formatted string property. The default value is FALSE.

Put
returns a string that contains the formatted output based on a SAS format.`Put(<Numeric Expression>,<String Expression>)Put(<String Expression>,<String Expression>)`

SetToStr
constructs a string in Multidimensional Expressions (MDX) format from a set. `SetToStr(<Set>)`

TupleToStr
returns a string in Multidimensional Expressions (MDX) format from a specified tuple. `TupleToStr(<Tuple>)`

UniqueName
returns the unique name of a specified level, dimension, member, or hierarchy. `<Level>.UniqueName<Dimension>.UniqueName<Member>.UniqueName<Hierarchy>.UniqueName`

UserName
returns the domain name and user name of the current connection. `UserName`

<member>.member_caption
returns the caption of the member. It is in nonstandard MDX format.

<dimension>.caption
returns the caption of the member. It is in nonstandard MDX format.

<hierarchy>.caption
returns the caption of the member. It is in nonstandard MDX format.

<level>.caption
returns the caption of the member. It is in nonstandard MDX format.

<member>.caption
returns the caption of the member. It is in nonstandard MDX format.

Tuple Functions

The MDX functions that are listed here indicate their return type.

Current
returns the current tuple from a set during an iteration. `<Set>.Current`

Item
returns a member from a specified tuple. Alternatively, it returns a tuple from a set. `<Set>.Item(<Index>)`

Note: If a member is returned, then it is a member function, not a tuple function.

StrToTuple
constructs a tuple from a specified string expression in Multidimensional Expressions (MDX) format. `StrToTuple(<String expression>)`

Miscellaneous Functions and Operators

Table 4.4 Miscellaneous Functions and Operators

Function or Operator	Explanation
, (comma operator)	An operator that is used to combine tuples to construct sets. For example: `{[time].[all time].[2001].[january], [time].[all time].[2001].[February],[time].[all time].[2001].[march]},` or to combine members to construct tuples such as `([Time].[January 2001], [Geography].[U.S.A])`

Function or Operator	Explanation

: (colon operator)	An operator that is used in part to construct sets and tuples. It replaces a series of comma operators. For example:
	`{[Time].[all Time].[2001].[January]`
	`:`
	`[Time].[all Time].[2001].[March]}`

{} (braces)	The SET constructor operator.
* (asterisk operator)	The alternative for CrossJoin. If you use a single operator, it is a direct replacement. You can nest CrossJoins by stringing additional operators and sets. For example, these expressions are equivalent: A * B – CrossJoin(A, B) A * B * C – CrossJoin (A, CrossJoin(B, C))
+ (plus operator for sets)	An alternative to union().
+ (plus operator for strings)	A concatenation of two strings.
/**/ (style comments)	Cause the OLAP server to ignore anything between the initial token and final token. These comments can span lines. NOTE: these comments do NOT nest.
// (style comments)	Cause the OLAP server to ignore anything after the double slash until the end of the line. These comments can NOT span lines.
- - (style comments)	Cause the OLAP server to ignore anything after the double dash until the end of the line. These comments can NOT span lines. These are essentially the same as the double-slash comments.
NON EMPTY	When applied to an axis, causes the OLAP Server to post-process the results of the axis set and remove any tuples that have empty results. For the SAS OLAP Server, it works with CrossJoin to provide "Optimize NON EMPTY CrossJoin" performance improvements.

Function or Operator	Explanation
<Set> AS aliasname	Creates an alias for an intermediate set. An intermediate set is one that is generated during the evaluation of an axis and has one or more functions operating on it.
Supports TRUE and FALSE	TRUE and FALSE conditional operators are supported in MDX queries
Call<UDF Name>	Executes a void returning user-defined function.
MAX SET SIZE	Limits the size of sets that the OLAP server creates. A value of 0 indicates there is no limit. The default is 1,000,000 components, where components are defined as the number of tuples in the set times the number of dimensions in each tuple. This function enables the administrator to control the system resources that are used by individual queries.