SAS Publishing

SAS® 9.1.3 OLAP Server

MDX Guide

Third Edition

The Power to Know.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2006. *SAS® 9.1.3 OLAP Server: MDX Guide, Third Edition*. Cary, NC: SAS Institute Inc.

SAS® 9.1.3 OLAP Server: MDX Guide, Third Edition

ISBN-13: 9781–59047–820–2
ISBN-10: 1–59047–820–7

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, March 2006

SAS Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at **support.sas.com/pubs** or call 1-800-727-3228.

Contents

What's New

Overview

The SAS OLAP Server enables users to develop and deploy scalable Online Analytical Processing (OLAP) applications. In addition, automated data loading and cube building are available through the use of a new administration interface called SAS OLAP Cube Studio, which was developed using Java technology.

OLAP queries are performed using the Multidimensional Expressions (MDX) query language in client applications that are connected to the SAS OLAP Server by using the following:

- the SQL Pass-Through Facility for OLAP, which is designed to process MDX queries within the PROC SQL environment
- open access technologies such as OLE DB for OLAP, ADO MD, and Java

New and enhanced features in SAS OLAP Server include the following:

- new tools for data loading and cube building
- new options added to the PROC OLAP statement
- new functions
- new tuning capabilities for the query thread pool
- new tuning options window
- improved performance
- improved querying capability
- improved aggregation tuning
- additional enhancements

Note: This section describes the features of the SAS OLAP Server that are new or enhanced since SAS 8.2. △

Note: The documentation for SAS OLAP Server has been expanded into three titles: *SAS OLAP Server: User's Guide*, *SAS OLAP Server: Administrator's Guide*, and *SAS OLAP Server: MDX Guide*. △

New Tools for Data Loading and Cube Building

The OLAP procedure, in addition to cube building, includes options for handling ragged hierarchies, defining global calculated members and named sets, assigning properties to levels, and optimizing cube creation and query performance. It also supports multiple hierarchies and drill-through tables.

SAS OLAP Cube Studio is an alternative Java interface to the OLAP procedure. This interface is also integrated with SAS Data Integration Studio.

New Options Added to the PROC OLAP Statement

The following options have been added to the PROC OLAP statement:

COMPACT_NWAY
when building a cube from a star schema, this option enables additional summarizations during the cube build that can decrease the size of the NWAY aggregation.

IGNORE_MISSING_DIMKEYS=TERSE | VERBOSE
when building a cube from a star schema, this option enables the continuation of the build when the fact table is found to contain keys that are not present in any of the dimension tables. The log receives information about the number and location of the missing keys.

For more information about these new options, see the *SAS OLAP Server: User's Guide*.

Note: Version 8 of the SAS OLAP Server can be used with SAS® 9. For help, see "Using V8 SAS OLAP Server" in SAS Help and Documentation. See also "Conversion and Migration Issues from Release 8.2 to SAS 9.1" in the *SAS OLAP Server: Administrator's Guide*. △

New Functions

The following functions are new:

- A Specify Map function has been added to SAS OLAP Cube Studio. The Specify Map function enables you to store ESRI Geographic Information System (GIS) spatial map information in the SAS Metadata Repository. This GIS information can then be read by the SAS OLAP Server and returned during a cube query.
- The Export Cube and Import Cube functions enable you to copy cube metadata from a source repository to a target repository, and if needed, to another server. The Export Cube function enables you to extract a cube's metadata from the source repository and save it in a file that you specify. All information about a cube, including its dimensions, hierarchies, levels, measures, notes, properties, calculated measures, aggregations, and security settings, will be extracted. The Import Cube function then enables you to save the cube metadata to another metadata repository on another metadata server.
- The Define Distinct Count function enables you to store NUNIQUE (Distinct Count) statistics as measures with a SAS OLAP cube. You can add or delete a distinct count measure either with PROC OLAP or in the Cube Designer wizard, within SAS OLAP Cube Studio. You can add or delete a Distinct Count measure.

You can also modify the name, caption, format, units, and description of the measure.

- The Synchronize Levels function enables you to synchronize a cube when the input table for an existing cube has encountered a column name change. This function finds the name differences between the cube and its input table and changes the hierarchy level names to match the input table column names.

New Tuning Capabilities for the Query Thread Pool

New tuning capabilities for the query thread pool are now available for each of your SAS OLAP Servers. Tuning the query thread pool enables you to optimize performance based on the number and frequency of query requests that are received by a SAS OLAP Server.

New Tuning Options Window

A tuning Options window has been added to the Manual Tuning function and the Advanced Aggregation Tuning plug-in. The Options window enables you to specify certain performance options on the PROC OLAP statement when adding or modifying aggregations for a cube. In this window you can view the current values and change the values when needed.

Improved Performance

Server performance is recorded and analyzed by using the Application Response Measurement (ARM) system.

The new multi-threaded data storage and server functionality provide faster cube performance. The data can be stored in a multidimensional form (MOLAP) or in a form that includes existing aggregations from presummarized data sources.

Improved Querying Capability

An SQL Pass-Through Facility for OLAP is available in SAS for use in querying cubes.

A subquery cache can now be enabled, disabled, and sized for each of your SAS OLAP Servers. The new cache stores subqueries that generate empty result sets.

Improved Aggregation Tuning

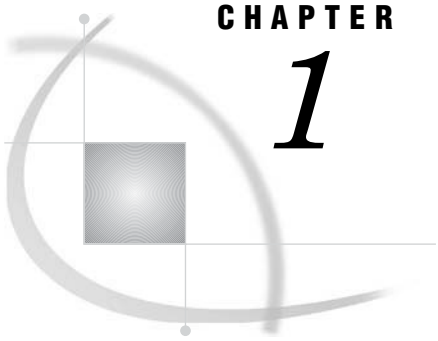
Aggregations can be added to or deleted from existing cubes. In addition, further enhancements for aggregation tuning have been made:

- The AGGREGATION statement and SAS OLAP Cube Studio's Manual Tuning function have been enhanced for use with cubes that employ aggregated data from other tables or for use with cubes that have no NWAY aggregation.

- The Advanced Aggregation Tuning plug-in provides a point-and-click interface that enables you to create and add aggregations to the list of existing aggregations that might already be defined for the cube.

Additional Enhancements

- The metadata structure is improved, and metadata is stored with the cube.
- Caching and logging can be enabled or disabled.
- Support for ad hoc calculations and time dimensions is improved.
- The start-up file generator has been removed from the SAS OLAP Server Monitor plug-in for SAS Management Console. SAS OLAP Servers are now automatically installed as Windows services; server start-up options are set by an INI file that is installed by the SAS Deployment wizard.
- When creating a cube dimension in SAS OLAP Cube Studio's Cube Designer wizard, a hierarchy for a dimension will automatically be created when no hierarchy has been explicitly defined and you click the **`Finish`** button in this window. This default hierarchy includes all levels that were specified for the current dimension and the order they were listed in for the dimension.
- The Calculated Members plug-in provides a point-and-click interface that enables you to add new measures or modify existing measures for the selected cube.



CHAPTER

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 9.1 OLAP Server technology uses MDX to create OLAP cubes and data queries. MDX is part of the underlying foundation for the SAS 9.1 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

To better understand the MDX language and the OLAP technology it supports, a basic understanding of the OLAP cube components is required.

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 that is descended 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 only available 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

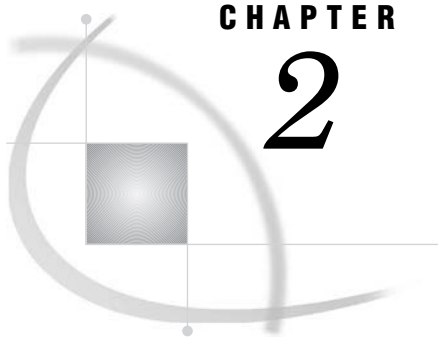
MDX extracts multidimensional views of data. A *tuple* is a slice of data from a cube. It is a selection of members (or cells) across dimensions in a cube. It can also be viewed as a cross-section or vector of member data in a cube. A tuple can be composed of

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.

Additional MDX Documentation

In addition to the MDX usage examples, functions and related topics that are found in this documentation, a supplementary text for the SAS OLAP Server is available. The *SAS OLAP Server: Concepts and Excerpts from "MDX Solutions with Microsoft SQL Server Analysis Services"* includes basic MDX information such as the MDX data model, MDX construction, comments in MDX, and a complete MDX function and operator reference. You can locate this text at **`support.sas.com/publishing`**.



CHAPTER

2

MDX Queries and Syntax

Basic MDX Queries and Syntax

Basic MDX queries use the SELECT statement to identify a data set that contains a subset of multidimensional data. The basic MDX SELECT statement is composed 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 may encounter a syntax error when a member name containing a single quote is used for a calculated member in an MDX query. To prevent this, include an additional single quote in the member name that contains the quote

  △
- 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: 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 Drillthrough

The DRILLTHROUGH statement is used in Multidimensional Expressions (MDX) to retrieve the source rowset(s) 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. A Multidimensional Expressions (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 drillthrough. Only cubes that have a drill-through table that is specified at cube creation support drillthrough.

Here is the syntax for the DRILLTHROUGH statement:

```
<drillthrough>    :=DRILLTHROUGH [<MaxRows>][<FirstRowset>] <MDX select>
   <MaxRows>     := MAXROWS <positive number>
   <FirstRowset> := FIRSTROWSET <positive number>
```

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

FIRST_ROWSET 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.

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. After selecting the physical OLAP server, right-click and select **`Properties`**. At the SAS OLAP Server Properties dialog box, select the **`Options`** tab, and then the **`Advanced Options`** button. 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 SAS OLAP Server under the following conditions:

- If the cube does not include an NWAY, then 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 SAS OLAP Server must have access to the drill-through table.
- If the cube uses pre-summarized aggregation tables, then 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 when the OLAP server that is associated with the OLAP schema that contains the cubes is invoked. For more information, see the *SAS Intelligence Platform: Planning and Administration Guide*.

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

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-config-dir***\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-config-dir***\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-config-dir\Lev1\SASMain\userfmt.cfg
     ```

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

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

     This makes the format catalog **fmtlib1.orionfmt** available. For more information, see the *SAS Intelligence Platform: Planning and 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. △

SAS Functions

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"
```

Available SAS Functions Exposed for Use in MDX Expressions

Table 2.1 SAS Functions Exposed 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»

Function	Description	Argument
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»
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>)

Function	Description	Argument
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)
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

To store numbers of large magnitude and to perform computations that require many digits of precision to the right of the decimal point, 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 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 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. 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 only be installed on the server. Additionally, 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: SAS OLAP Server currently does not support the use of square brackets in cube, dimension, hierarchy, level, or member names or captions.
△

(	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
CASE
CATALOG_NAME
CELL
CELL_ORDINAL
CHAPTERS
CHILDREN
CHILDREN_CARDINALITY
CLOSINGPERIOD
COALESCEEMPTY
COLUMNS
CORRELATION
COUNT
COUSIN
COVARIANCE
COVARIANCEN
CREATE
CROSSJOIN
CUBE_NAME
CURRENT
CURRENTMEMBER
DATAMEMBER
DEFAULTMEMBER
DESC
DESCENDANTS
DESCRIPTION
DIMENSION
DIMENSIONS
DIMENSION_UNIQUE_NAME
LASTCHILD
LASTPERIODS
LASTSIBLING
LEAD
LEAVES
LEVEL
LEVELS
LEVEL_NUMBER
LEVEL_UNIQUE_NAME
LIBRARY
LINKMEMBER
LINREGINTERCEPT
LINREGPOINT
LINREGR2
LINREGSLOPE
LINREGVARIANCE
LOOKUPCUBE
MAX
MAXROWS
MEDIAN
MEMBER
MEMBERS
MEMBERTOSTR
MEMBER_CAPTION
MEMBER_GUID
MEMBER_NAME
MEMBER_ORDINAL
MEMBER_TYPE
MEMBER_UNIQUE_NAME
STDDEVP
STDEV
STDEVP
STRIPCALCULATEDMEMBERS
STRTOMEMBER
STRTOSET
STRTOTUPLE
STRTOVALUE
SUBSET
SUM
TAIL
THEN
TOGGLEDRILLSTATE
TOPCOUNT
TOPPERCENT
TOPSUM
TRUE
TUPLETOSTR
UNION
UNIQUENAME
USE
USERNAME
VALIDMEASURE
VALUE
VAR
VARIANCE
VARIANCEP
VARP
VISUALTOTALS

DISPLAY_INFO	MIN	WHEN
DISTINCT	MTD	WHERE
DISTINCTCOUNT	NAME	WITH
DRILLDOWNLEVEL	NAMETOSET	WTD
DRILLDOWNLEVELBOTTOM	NEXTMEMBER	XOR
DRILLDOWNLEVELTOP	NON	YTD

External Functions

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 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
```

Defining External Functions in Java

SAS runs on many different types of computers. As a result, you can write external functions in the Java language. Here is a simple Java class that implements the **`addOne()`** function from earlier:

```
public class Hello
{
    public double addOne(double d)
    {
       return d+1.0;
    }
}
```

To prepare this class for execution you must obtain and compile a copy of the Java Development Kit (JDK), which is available on the World Wide Web. Here is an example:

```
C:> javac Hello.java
```

After compiling the JDK, you install the resulting Hello.class file in a location where the server can find it. Currently this means you must list the directory that contains the .class file in the CLASSPATH environment variable before you start the server.

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 9.1 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), while 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 OS/390 use out-of-process JVMs. In later releases, hosts should be able to use in-process JVMs. OS/390 will use a shared address space so 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 13.

Supported Versions of Java

SAS OLAP Server 9.1 supports the same version of Java that SAS 9.1 does. For example, under Windows, SAS OLAP Server 9.1 and SAS 9.1 require Java Version 1.4.1.

Using Derived Statistics with the Aggregate Function

Example 1

When the aggregate function is used in a calculated member, the statistic associated with 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], [Measures].[ACTUAL_N],
   [Measures].[ACTUAL_AVG], measures.calc} on 1
FROM [prdmddb]
```

This example returns:

	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

When the current measure is :

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:

```
                      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
```

When the current measure is :

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:

```
                      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
```

When the current measure is :

actual_max — it subtracts the actual_min value associated with each child of 1994 from the corresponding actual_max value. It picks 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 picks 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 was 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:

```
          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:

```
           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 calculated the correct value, so it simply returns missing.

Example 6

This example shows what happens when the query was 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:

```
           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]
```

	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 measures 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.

Standard Statistic	How it is aggregated
MAX	get the max of the values
MIN	get the min 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

CHAPTER 3

MDX Usage Examples

Basic Examples

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,
   non empty { [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
NON EMPTY(CROSSJOIN ([Time].[YQM].[All YQM].Children ,[Measures].[Total_Retail_PriceSUM]))
ON COLUMNS,
NON EMPTY({[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
NONEMPTYCROSSJOIN ([Time].[All YQM].Children,[Measures].[Total_Retail_PriceSUM],
{[Demographics].[All Demographics].Female},3) ON COLUMNS,
NON EMPTY({[Geography].[Global].[All Global].Children }) ON ROWS
FROM [OrionStar]
```

Example of the COALESCE EMPTY function:

```
WITH MEMBER [Measures].[Quantity_nomiss] AS 'CoalesceEmpty(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

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:

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:

Date	Chevy	Chrysler	Ford	Toyota
All date	13500	20000	12285.71	8444.45
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 cars. 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:

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

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:

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

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:

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

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 members down 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:

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

Item	Qty
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 members down 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:

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 members down 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:

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 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:

Item	Qty
Computers, Peripherals	78,263
TV, DVD, Video	47,725
Outdoor & Sporting	304,345
Sports Equipment	51,214

Session-Named Set Examples

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:

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
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:

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
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." 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:

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
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
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

State	Product	Qty	Total
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;
```

APPENDIX

1

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 — if the evaluated expression is an empty cell value, then TRUE is returned. Otherwise, FALSE is returned.

```
IsEmpty(<Value Expression>)
```

IS — if two compared objects are equivalent, then TRUE is returned. Otherwise, FALSE is returned.

```
<Object 1>IS Null

<Object 1>IS <Object 2>
```

IsAncestor — if a specified member is an ancestor of another specified member, then TRUE is returned. Otherwise, FALSE is returned.

```
IsAncestor(<Member1>,<Member2>)
```

IsLeaf — if a specified member is a leaf member, then TRUE is returned. Otherwise, FALSE is returned.

```
IsLeaf(<Member>)
```

IsSibling — if a specified member is a sibling of another specified member, then TRUE is returned. 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: This example shows a moving average across all dimensions of time.

```
Avg(Time.CurrentMember.Lag
(Iif(time.CurrentMember.Level is Time.Month_Num,2,
Iif(Time.CurrentMember.Level is Time.Quarter,1,0)))
:Time.CurrentMember, Measures.[Total_Retail_PriceSUM])
```

The Total_Retail_PriceSUM is included in this 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>.Count
```

```
Count(<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.

AddCalculated Members — 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
```

Example: Referencing All levels/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>])
```

Example: This 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 are specified by <*Member*> and are <*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 A1.1 Ancestor Flag Options

Options	Returns
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(s) 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 A1.2 Ancestor Expressions and Returns

Expressions	Returns
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: This example shows retrieving the member at the specific level above and the current member dynamically using 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 while 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: This 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: This 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: This 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_flags>])
```

Table A1.3 Descendants Flag Options

Options	Returns
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>)
```

Drilldown Level — 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}])
```

Drilldown LevelBottom — 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. △

Drilldown LevelTop	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. △

Drilldown Member	drills down to the members in a specified set that are present in a second specified set.

```
DrilldownMember(<Set1>,<Set2[,Recursive])
```

Drilldown MemberBottom	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. △

Drilldown MemberTop	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])
```

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>)`
NonEmpty Crossjoin	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
```

StripCalculated Members — 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. Alternatively, it 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. △

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 non-standard MDX format.
<dimension> .caption	returns the caption of the member. It is in non-standard MDX format.
<hierarchy> .caption	returns the caption of the member. It is in non-standard MDX format.
<level> .caption	returns the caption of the member. It is in non-standard MDX format.
<member> .caption	returns the caption of the member. It is in non-standard 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>)`

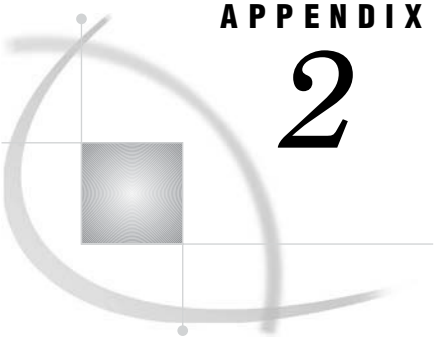
Miscellaneous Functions and Operators

, (comma operator)	an operator to combine tuples to construct sets such as {[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]).
: (colon operator)	an operator to specify ranges of tuples to contract sets such as {[Time].[all Time].[2001].[January] : [Time].[all Time].[2001].[March]}. It is the set constructor operator.
{} (braces)	an alternative to crossjoin().
* (asterisk operator)	an alternative to nested crossjoins.
+ (plus operator for sets)	an alternative to union().
+ (plus operator for strings)	a concatenation of two strings.
/* */ (style comments)	
// (style comments)	
– (style comments)	

NON EMPTY	
<Set> AS aliasname	
Supports TRUE and FALSE	
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 enables the administrator to control the system resources that are used by individual queries.

Additional MDX Documentation

In addition to the MDX usage examples, functions and related topics that are found in this documentation, a supplementary text for the SAS OLAP Server is available. The *SAS OLAP Server: Concepts and Excerpts from "MDX Solutions with Microsoft SQL Server Analysis Services"* includes basic MDX information such as the MDX data model, MDX construction, comments in MDX, and a complete MDX function and operator reference. You can locate this text at **`support.sas.com/publishing`**.



APPENDIX

2

Recommended Reading

Recommended Reading

Here is the recommended reading list for this title:

- *Administrator for Enterprise Clients: User's Guide*
- *SAS Data Providers: ADO/OLE DB Cookbook*
- *SAS Language Reference: Concepts*
- *SAS Language Reference: Dictionary*
- *SAS Management Console: User's Guide*
- *SAS Intelligence Platform: Administration Guide*
- *SAS Open Metadata Architecture Reference*
- *SAS OLAP Server: Concepts and Excerpts from "MDX Solutions with Microsoft SQL Server Analysis Services"*
- *SAS OLAP Server: Administrator's Guide*
- SAS Companion that is specific to your operating environment

For a complete list of SAS publications, see the current *SAS Publishing Catalog*. To order the most current publications or to receive a free copy of the catalog, contact a SAS representative at

SAS Publishing Sales
SAS Campus Drive
Cary, NC 27513
Telephone: (800) 727-3228*
Fax: (919) 677-8166
E-mail: **sasbook@sas.com**
Web address: **support.sas.com/publishing**
* For other SAS Institute business, call (919) 677-8000.

Customers outside the United States should contact their local SAS office.

Glossary

aggregation
a summary of detail data that is stored with or referred to by a cube. Aggregations support rapid and efficient answers to business questions.

ancestor
within a dimension hierarchy, a member that resides at a higher level in relation to other members in the hierarchy. For example, if a Geography dimension includes the levels Continent, Country, and City, then Europe and France would be ancestors of Paris, and Asia and Thailand would be ancestors of Bangkok.

ARM (Application Response Measurement)
an application programming interface that was developed by an industry partnership and which is used to monitor the availability and performance of software applications. ARM monitors the application tasks that are important to a particular business.

calculated member
in a dimension, a member whose value is derived from the values of other members.

cell
in a cube, the intersection that is defined by selecting one member from each dimension of that cube.

child
within a dimension hierarchy, a descendant in level n-1 of a member that is at level n. For example, if a Geography dimension includes the levels Country and City, then Bangkok would be a child of Thailand, and Hamburg would be a child of Germany.

cube
a logical set of data that is organized and structured in a hierarchical, multidimensional arrangement. A cube is a directory structure, not a single file. A cube includes measures, and it can have numerous dimensions and levels of data.

descendant
in a dimension hierarchy, a member that resides at a lower level in relation to other members in the hierarchy. For example, if a Geography dimension includes the levels Country, State, and City, then California and Los Angeles would be descendants of USA.

detail data

nonsummarized (or partially summarized) factual information that pertains to a single area of interest, such as sales figures, inventory data, or human-resource data.

dimension
a group of closely related hierarchies. Hierarchies within a dimension typically represent different groupings of information that pertains to a single concept. For example, a Time dimension might consist of two hierarchies: (1) Year, Month, Date, and (2) Year, Week, Day. See also hierarchy.

dimension table
in a star schema, a table that contains the data for one of the dimensions. The dimension table is connected to the star schema's fact table by a primary key. The dimension table contains fields for each level of each hierarchy that is included in the dimension.

drill down
in a view of an OLAP cube, to start at one level of a dimension hierarchy and to click through one or more lower levels until you reach the data that you are interested in.

drill up
in a view of an OLAP cube, to start at one level of a dimension hierarchy and to click through one or more higher levels until you reach the level of summarized data that you are interested in.

drill-through table
a view, data set, or other data file that contains data that is used to define a cube. Drill-through tables can be used by client applications to provide a view from processed data into the underlying data source.

fact
a single piece of factual information in a data table. For example, a fact can be an employee name, a customer's phone number, or a sales amount. It can also be a derived value such as the percentage by which total revenues increased or decreased from one year to the next.

fact table
the central table in a star schema. The fact table contains the individual facts that are being stored in the database as well as the keys that connect each particular fact to the appropriate value in each dimension.

hierarchy
an arrangement of members of a dimension into levels that are based on parent-child relationships. Members of a hierarchy are arranged from more general to more specific. For example, in a Time dimension, a hierarchy might consist of the members Year, Quarter, Month, and Day. In a Geography dimension, a hierarchy might consist of the members Country, State or Province, and City. More than one hierarchy can be defined for a dimension. Each hierarchy provides a navigational path that enables users to drill down to increasing levels of detail. See also member, level.

leaf member
the lowest-level member of a hierarchy. Leaf members do not have any child members.

level
an element of a dimension hierarchy. Levels describe the dimension from the highest (most summarized) level to the lowest (most detailed) level. For example, possible levels for a Geography dimension are Country, Region, State or Province, and City.

MDX (multidimensional expressions) language

a standardized, high-level language that is used for querying multidimensional data sources. The MDX language is the multidimensional equivalent of SQL (Structured Query Language).

measure
a special dimension that contains summarized numeric data values that are analyzed. Total Sales and Average Revenue are examples of measures. For example, you might drill down within the Clothing hierarchy of the Product dimension to see the value of the Total Sales measure for the Shirts member.

member
a name that represents a particular data element within a dimension. For example, September 1996 might be a member of the Time dimension. A member can be either unique or non-unique. For example, 1997 and 1998 represent unique members in the Year level of a Time dimension. January represents non-unique members in the Month level, because there can be more than one January in the Time dimension if the Time dimension contains data for more than one year.

metadata repository
a collection of related metadata objects, such as the metadata for a set of tables and columns that are maintained by an application. A SAS Metadata Repository is an example.

metadata server
a server that provides metadata management services to one or more client applications. A SAS Metadata Server is an example.

MOLAP (multidimensional online analytical processing)
a type of OLAP that stores aggregates in multidimensional database structures.

NWAY aggregation
the aggregation that has the minimum set of dimension levels that is required for answering any business question. The NWAY aggregation is the aggregation that has the finest granularity. See also granularity.

OLAP (online analytical processing)
a software technology that enables users to dynamically analyze data that is stored in multidimensional database (MDDB) tables.

OLAP schema
a group of cubes. A cube is assigned to an OLAP schema when it is created, and an OLAP schema is assigned to a SAS OLAP Server when the server is defined in the metadata. A SAS OLAP Server can access only the cubes that are in its assigned OLAP schema.

OLE DB for OLAP
an OLAP API that is used to link OLAP clients and servers by means of a multidimensional expressions (MDX) language. See also MDX (multidimensional expressions) language.

parent
within a dimension hierarchy, the ancestor in level n of a member in level n-1. For example, if a Geography dimension includes the levels Country and City, then Thailand would be the parent of Bangkok, and Germany would be the parent of Hamburg. The parent value is usually a consolidation of all of its children's values.

result set
the set of rows or records that a server or other application returns in response to a query.

SAS ARM interface

an interface that can be used to monitor the performance of SAS applications. In the SAS ARM interface, the ARM API is implemented as an ARM agent. In addition, SAS supplies ARM macros, which generate calls to the ARM API function calls, and ARM system options, which enable you to manage the ARM environment and to log internal SAS processing transactions. See also ARM (Application Response Measurement).

SAS format
a pattern or set of instructions that SAS uses to determine how the values of a variable (or column) should be written or displayed. SAS provides a set of standard formats and also enables you to define your own formats.

SAS informat
a pattern or set of instructions that SAS uses to determine how data values in an input file should be interpreted. SAS provides a set of standard informats and also enables you to define your own informats.

SAS Management Console
a Java application that provides a single user interface for performing SAS administrative tasks.

SAS Metadata Repository
a repository that is used by the SAS Metadata Server to store and retrieve metadata. See also SAS Metadata Server.

SAS Metadata Repository
one or more files that store metadata about application elements. Users connect to a SAS Metadata Server and use the SAS Open Metadata Interface to read metadata from or write metadata to one or more SAS Metadata Repositories. The metadata types in a SAS Metadata Repository are defined by the SAS Metadata Model.

SAS OLAP Cube Studio
a Java interface for defining and building OLAP cubes in SAS System 9 or later. Its main feature is the Cube Designer wizard, which guides you through the process of registering and creating cubes.

SAS OLAP Server
a SAS server that provides access to multidimensional data. The data is queried using the multidimensional expressions (MDX) language.

SAS Open Metadata Architecture
a general-purpose metadata management facility that provides metadata services to SAS applications. The SAS Open Metadata Architecture enables applications to exchange metadata, which makes it easier for these applications to work together.

schema
a map or model of the overall data structure of a database. An OLAP schema specifies which group of cubes an OLAP server can access.

slice
a subset of data from a cube, where the data in the slice pertains to one or more members of one or more dimensions. For example, from a cube that contains data about customer feedback, one slice might pertain to feedback on one particular product (one member of the Product dimension). Another slice might pertain to feedback on that product from customers residing in particular geographic areas who submitted their feedback during a certain time period (one member of the Product dimension, multiple members of the Geography dimension, one or more members of the Time dimension).

SMP (symmetric multiprocessing)

a hardware and software architecture that can improve the speed of I/O and processing. An SMP machine has multiple CPUs and a thread-enabled operating system. An SMP machine is usually configured with multiple controllers and with multiple disk drives per controller.

SQL (Structured Query Language)
a standardized, high-level query language that is used in relational database management systems to create and manipulate database management system objects.

thread
a single path of execution of a process in a single CPU, or a basic unit of program execution in a thread-enabled operating system. In an SMP environment, which uses multiple CPUs, multiple threads can be spawned and processed simultaneously. Regardless of whether there is one CPU or many, each thread is an independent flow of control that is scheduled by the operating system. See also SMP (symmetric multiprocessing), thread-enabled operating system, threading.

Time dimension
a dimension that divides time into levels such as Year, Quarter, Month, and Day.

tuple
a data object that contains two or more components. In OLAP, 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 of member data in a cube. For example, ([time].[all time].[2003], [geography].[all geography].[u.s.a.], [measures].[actualsum]) is a tuple that contains data from the Time, Geography, and Measures dimensions.

wizard
an interactive utility program that consists of a series of dialog boxes, windows, or pages. Users supply information in each dialog box, window, or page, and the wizard uses that information to perform a task.

Index

N

O

P

Q

R

S

T

U

W

Your Turn

If you have comments or suggestions about *SAS 9.1.3 OLAP Server: MDX Guide, Third Edition*, please send them to us on a photocopy of this page, or send us electronic mail.

For comments about this book, please return the photocopy to

SAS Publishing
SAS Campus Drive
Cary, NC 27513
E-mail: **yourturn@sas.com**

For suggestions about the software, please return the photocopy to

SAS Institute Inc.
Technical Support Division
SAS Campus Drive
Cary, NC 27513
E-mail: **suggest@sas.com**