



THE
POWER
TO KNOW.

SAS[®] 9.2 OLAP Server MDX Guide



The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2009. *SAS® 9.2 OLAP Server: MDX Guide*. Cary, NC: SAS Institute Inc.

SAS® 9.2 OLAP Server: MDX Guide

Copyright © 2009, SAS Institute Inc., Cary, NC, USA

ISBN 978-1-59994-339-8

All rights reserved. Produced in the United States of America.

For a hard-copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a Web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

U.S. Government Restricted Rights Notice. Use, duplication, or disclosure of this software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227-19 Commercial Computer Software-Restricted Rights (June 1987).

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

1st electronic book, February 2009

2nd electronic book, September 2009

1st printing, March 2009

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/publishing or call 1-800-727-3228.

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

Contents

<i>What's New</i>	v
Overview	v
OLAP Cube Building Features	v
OLAP Cube Maintenance Features	vi
Cube Aggregation Tuning Features	vii
Incremental Cube Update Features	vii
Security For Cubes	vii
SAS Tree View Features	vii
New OLAP Procedure Options	viii
Chapter 1 \triangle MDX Introduction and Overview	1
MDX Overview	1
Basic MDX and Cube Concepts	1
Additional MDX Concepts and Expressions - Tuples and Sets	2
Chapter 2 \triangle MDX Queries and Syntax	5
Basic MDX Queries and Syntax	5
MDX Drillthrough	7
Basic MDX DDL Syntax	8
SAS Functions	9
Using Derived Statistics with the Aggregate Function	14
SAS OLAP Security Totals and Permission Conditions	19
Chapter 3 \triangle MDX Usage Examples	23
Basic Examples	23
Calculated Member Examples	27
Query-Calculated Member Examples	28
Session-Level Calculated Member Examples	29
Drill-Down Examples	31
Session-Named Set Examples	34
Appendix 1 \triangle MDX Functions	41
Dimension Functions	41
Hierarchy Functions	42
Level Functions	42
Logical Functions	42
Member Functions	43
Numeric Functions	44
Set Functions	47
String Functions	57
Tuple Functions	58
Miscellaneous Functions and Operators	59

Appendix 2 \triangle Recommended Reading 61

Recommended Reading 61

Glossary 63

Index 69

What's New

Overview

The SAS OLAP Server contains various new functions, tools, and features for SAS 9.2.

- OLAP cube building features
- OLAP cube maintenance features
- cube aggregation tuning features
- incremental cube update features
- security for cubes
- SAS tree view features
- OLAP procedure options

These functions are accessed from SAS OLAP Cube Studio. In addition, some of these functions have equivalent OLAP procedure statements or options.

OLAP Cube Building Features

- The data table selection process in the Cube Designer wizard has been simplified and reorganized into an easier and more intuitive order. Tables are now grouped based on the libraries that the tables are registered to. The table selection pages also contain functions that enable you to access the properties for a table and view the data for a table.
- You can now create multiple hierarchies for a dimension with the Define a Hierarchy dialog box in the Dimension Designer. You can also designate one of the hierarchies as a default hierarchy.
- You can now build a time dimension with user-supplied time hierarchies that auto-populate the levels and formats for the Time dimension. The Add-Supplied dialog box enables you to select from suggested time hierarchies that are based on a single date-formatted column.

The COLUMN= option is used on the LEVEL statement when the Add-Supplied function is used.

- If the metadata server connection is lost during a SAS OLAP Cube Studio session, a new metadata server connection is automatically attempted during the same SAS OLAP Cube Studio session. This enables you to continue your work in that same session without losing any data changes you are making.

OLAP Cube Maintenance Features

- The Calculated Members Wizard is now fully integrated into SAS OLAP Cube Studio. The addition of the advanced expression builder enables you to specify user-defined formats, set the solve order, and build a custom MDX formula for a calculated member.
- The View Cube function enables you to verify the cube build process and visualize the data structure of a cube.
- SAS OLAP cube metadata can now be exported and imported as part of a SAS package or in batch mode. Cubes built with SAS OLAP Cube Studio 4.2 can be copied or moved between SAS 9.2 systems with the Export SAS Package and Import SAS Package functions that are part of SAS OLAP Cube Studio 4.2. Cubes that were built with a version of SAS that is earlier than 9.2 can also be moved to the SAS 9.2 environment by using the ExportCubes and ImportCubes batch tools.

Note: The ExportCubes batch tool is supported in a hot fix that is applied to SAS 9.1.3 SP4. Cube metadata that is exported with the ExportCubes tool can then be imported in the second maintenance release after SAS 9.2 with the ImportCubes batch tool. △

- The Synchronize Column Changes 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. It then changes the internal cube files to match the input table column names.
- The Refresh Cube function enables the SAS OLAP Server to access calculated members that were created or updated since the cube was last accessed by the SAS OLAP Server. This function notifies running SAS OLAP Server instances to capture calculated member changes made to the cube since it was last loaded by those instances.
- You can now quickly change the OLAP schema for a cube from within SAS OLAP Cube Studio.
- Cubes can now be disabled to make necessary changes and updates and then enabled again. Disabling a cube makes it transparent to the users connected to any active OLAP Server instances. A cube is disabled in order to make edits to the cube or to perform administrative tasks on the cube. Enabling a cube brings it back into production.
- You can now choose to export the code generated by PROC OLAP in either a long form or short form. The Export Code function enables you to store in a text file the PROC OLAP code that is used to edit or build the cube. The long form contains all PROC code that is used to create a cube. The short form text file contains only the PROC OLAP statement, the cube name, and the METASVR statement and obtains the remainder of the cube structure from an existing registration on the metadata server.
- You can now perform impact analysis for a cube. This function enables you to view the relationship of a cube to the objects that it is associated with. Impact analysis shows the potential impact of changes that you might make to a cube.
- A cube or cube-related object can now have documents and notes attached to provide additional information about the cube or object.

- Cube jobs are now automatically generated when a cube is built. A cube job is a collection of SAS tasks that create output. When you deploy a cube job, the code needed to create the cube is generated.
- You can now change the name of a cube or other object (such as a job, document, or note) by selecting that object in the tree view and then selecting the Rename function.

Cube Aggregation Tuning Features

- The newly designed Aggregation Tuning wizard enables you to build, edit, and customize aggregations for a cube. You can generate new aggregations for a cube based on the ARM log or cardinality, or you can use the manual aggregation builder.

Incremental Cube Update Features

- You can now add data and members to a cube without having to rebuild the cube. To support this feature, new PROC OLAP statements and options have been added. Within SAS OLAP Cube Studio, the Incremental Update wizard has been added.
- The Coalesce Aggregations function is part of the Incremental Update function. It enables you to combine all of the individual rack tables that result from incremental updates into a single aggregation table. The COALESCE_AGGREGATIONS option has been added to support this function. In SAS OLAP Cube Studio, the Coalesce Aggregations wizard has been added.

Security For Cubes

- You can now define member-level security in SAS OLAP Cube Studio. The Add Authorization dialog box enables you to add MDX expressions that limit access to a cube dimension. This ensures that only designated portions of the data are visible to a user or group of users.

The SECURITY_SUBSET= option is now available in the OLAP procedure. The option determines whether security totals will be used. The subset that is defined is then applied when calculating the cell values.
- Identity-driven security enables you to substitute identity values in a permission condition. It enables you to insert a placeholder into the permission condition that, at query time, gets resolved to a string that represents the user identity.

SAS Tree View Features

- The tree view is now divided into two tabs, the **Folders** tab and **Inventory** tab. The **Inventory** tab lists the cubes and their components, such as schemas, tables, and libraries. The **Folders** tab displays a folder view of the components of the cube. This view shows which folders a cube, source tables, and libraries are located in.
- You can now access functions that are specific to an object in the tree view. The functions that are listed in the context menu are specific to that object and the current status of that object. You can access the context menu by right-clicking an object.

- In SAS OLAP Cube Studio, you can now copy or move objects between folders. You can also find the folder location of an object in the **Inventory** tab with the Find In Folders function.

New OLAP Procedure Options

The following OLAP procedure options are new for SAS OLAP 9.2:

PROC OLAP Statement Options

- ADD_DATA
- COALESCE_AGGREGATIONS
- ESRI_MAP_SERVER=
- ESRI_REPLACE
- MAX_RETRIES=
- MAX_RETRY_WAIT=
- MIN_RETRY_WAIT=
- NONUPDATEABLE
- OUTCUBE=
- OUTSCHEMA=
- RENAME
- SECURITY_SUBSET= YES|NO
- SYNCHRONIZE_COLUMNS (replacing SYNCHRONIZE_LEVELS)
- UPDATE_DISPLAY_NAMES
- UPDATE_IN_PLACE

Dimension Statement Options

- MAP_SERVICE=
- UPDATE_DIMENSION=

LEVEL Statement Options

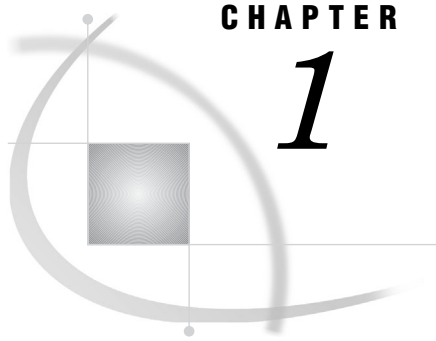
- COLUMN=
- ESRI_MAP_LAYER=
- FORMAT=

MEASURE Statement Options

- INCLUDE_CALCULATED_MEMBERS
- NOINCLUDE_CALCULATED_MEMBERS

PROPERTY Statement Options

- ESRI_MAP_FIELD=



CHAPTER

1

MDX Introduction and Overview

<i>MDX Overview</i>	1
<i>Basic MDX and Cube Concepts</i>	1
<i>Dimensions</i>	2
<i>Hierarchies</i>	2
<i>Levels</i>	2
<i>Members and Measures</i>	2
<i>Additional MDX Concepts and Expressions - Tuples and Sets</i>	2

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 9.2 OLAP Server technology uses MDX to create OLAP cubes and data queries. MDX is part of the underlying foundation for the SAS 9.2 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 descending in a dimension is a component of the previous level. For example, a time dimension could include the following levels: Year, Quarter, Month, Week, and Day.

Members and Measures

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

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

Query scope calculated member	is 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.

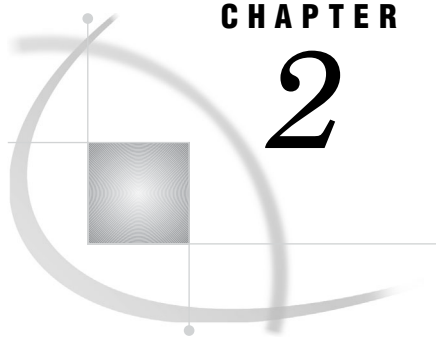
Calculated members can be created in the Measures dimension and can include any combination of members. Calculated members can also be created in any other dimension and are known as *nonmeasure-based calculated members*. Examples of measures include sales counts, profit margins, and distribution costs.

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.



CHAPTER

2

MDX Queries and Syntax

<i>Basic MDX Queries and Syntax</i>	5
<i>MDX Syntax</i>	6
<i>MDX Drillthrough</i>	7
<i>Specifying the Maximum Number of Drill-Through Rows</i>	7
<i>Ensuring That Tables Are Accessible at Query Time</i>	7
<i>Working with User-Defined Formats</i>	8
<i>Basic MDX DDL Syntax</i>	8
<i>SAS Functions</i>	9
<i>SetToList MDX Function</i>	9
<i>Function Arguments and Return Types</i>	9
<i>Numeric Precision</i>	9
<i>Magnitude versus Precision</i>	10
<i>Computational Considerations of Fractions</i>	10
<i>Using the TRUNC Function</i>	10
<i>Differences with Microsoft Analysis Services 2000</i>	10
<i>SAS MDX Reserved Keywords</i>	11
<i>Using Derived Statistics with the Aggregate Function</i>	14
<i>Example 1</i>	14
<i>Example 2</i>	14
<i>Example 3</i>	15
<i>Example 4</i>	16
<i>Example 5</i>	17
<i>Example 6</i>	17
<i>Example 7</i>	18
<i>Standard Statistics</i>	18
<i>Derived Statistics</i>	18
<i>SAS OLAP Security Totals and Permission Conditions</i>	19
<i>Example 1—Applying the SECURITY_SUBSET Option to an MDX Query</i>	19
<i>Example 2—Applying the SECURITY_SUBSET Option to an MDX Query</i>	20
<i>Default Member and the All Member</i>	21
<i>Virtual Members and Security Totals</i>	22

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

Δ

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 <i>slicer</i> . 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. Δ

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

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

MDX Syntax

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

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

Note: For more information on reserved words see “SAS MDX Reserved Keywords” on page 11. Δ

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

- Single and double quotation marks should balance.

MDX Drillthrough

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

Here is the syntax for the DRILLTHROUGH statement:

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

- MAXROWS indicates the maximum number of rows that should be returned by the resulting rowset.
- FIRSTROWSET specifies the first rowset to return.

Specifying the Maximum Number of Drill-Through Rows

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

- 1 In the tree view for the Server Manager plug-in of SAS Management Console, select the node for your OLAP server. This is the physical OLAP server and is located by drilling down from the top of the Server Manager tree view.
- 2 After selecting the physical OLAP server, right-click and select **Properties**.
- 3 At the SAS OLAP Server Properties dialog box, select the **Options** tab, and then the **Advanced Options** button.
- 4 At the Advanced Options dialog box, select the **Server** tab and enter the needed value for the **Maximum number of flattened rows** field. The default setting is 300,000 rows.

Ensuring That Tables Are Accessible at Query Time

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

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

To ensure that the necessary tables are accessible at query time, the applicable library names need to be allocated 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: Data Administration Guide* and the *SAS Intelligence Platform: System Administration Guide*.

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

Working with User-Defined Formats

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

- The preferred solution is to name the format catalog **formats.sas7bcat** and to place the catalog in the following directory:

```
path-to-configuration-directory\Levl\SASMain\SASEnvironment\SASFormats
```

- An alternative method of making user-defined formats “visible” is to follow this procedure:

- 1 Add a line to the configuration file

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

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

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

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

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

Basic MDX DDL Syntax

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

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

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

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

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

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

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

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

SAS Functions

When building MDX expressions, you can use all SAS functions, functions included in other SAS products, and user-written functions created with the SAS/TOOLKIT. 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)
```

Note: If you use a SAS function that has the same name as an MDX function (for example: SUM()), you must prefix it with **SAS!**. Δ

SetToList MDX Function

For SAS functions that take variable-length parameter lists, the SetToList MDX function can be useful as it enables you to convert MDX sets into variable-length parameter lists. See the SetToList function description for further information.

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, the SAS OLAP Server stores all numeric values as floating-point representation. *Floating-point representation* is an implementation of scientific notation, in which numbers are represented as numbers between 0 and 1 times a power of 10.

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

Magnitude versus Precision

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

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

Computational Considerations of Fractions

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

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

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

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

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

Using the TRUNC Function

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

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

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

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

Differences with Microsoft Analysis Services 2000

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

If you use a client-side function, then all the execution is on the client. The SAS OLAP Server is a thin-client system that is designed for high volume and scalability,

with all evaluation done on the server. Therefore, external function libraries such as SAS functions can 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: The SAS OLAP Server currently does not support the use of square brackets in cube, dimension, hierarchy, level, or member names or captions.

Δ

(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	SOLVE_ORDER
CALL	LAG	STDDEV
CAPTION	LASTCHILD	STDDEVP
CASE	LASTPERIODS	STDEV
CATALOG_NAME	LASTSIBLING	STDEVP
CELL	LEAD	STRIPCALCULATEDMEMBERS
CELL_ORDINAL	LEAVES	STRTOMEMBER
CHAPTERS	LEVEL	STRTOSET
CHILDREN	LEVELS	STRTOTUPLE
CHILDREN_CARDINALITY	LEVEL_NUMBER	STRTOVALUE

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

Using Derived Statistics with the Aggregate Function

Example 1

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

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

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

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

This example returns the following:

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

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

actual_max	It takes the MAX of the actual_max values associated with the children of 1994.
actual_min	It takes the MIN of the actual_min values associated with the children of 1994.
actual_sum	It summarizes the actual_sum values associated with the children of 1994.
actual_n	It counts the actual_n values associated with the children of 1994.
actual_avg	It divides the aggregated sum of the children of 1994 by the aggregated count value of the children of 1994.
calculated measure calc	It takes the MAX of the actual_max values for the children of 1994 (1000) and subtracts the MIN of the actual_min values for the children of 1994 (3).

Example 2

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

```

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

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

```

This example returns the following:

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

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

actual_max	It takes the MAX of the actual_max values associated with the children of 1994 and adds 1 to it.
actual_min	It takes the MIN of the actual_max values associated with the children of 1994 and adds 1 to it.
actual_sum	It adds 1 to each of the actual_max values associated with the children of 1994 and SUMs the values.
actual_n	It adds 1 to each of the actual_max values associated with the children of 1994 and SUMs the values.
actual_avg	It cannot compute a derived measure based on another measure, so it returns a missing value.

Example 3

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

```

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

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

```

This example returns the following:

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

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

actual_max	It subtracts the actual_min value associated with each child of 1994 from the corresponding actual_max value. It 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 is the actual_avg measure.

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

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

This example returns the following:

	actual_sum	actual_n	agg complexfunc
1	\$89,763.00	180	501.1944444444444
2	\$93,359.00	180	501.1944444444444
3	\$89,049.00	180	501.1944444444444
4	\$88,689.00	180	501.1944444444444

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

summed together. This total is then divided by the number of children of 1994 to give the aggregate value.

Example 5

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

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

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

This example returns the following:

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

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

Example 6

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

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

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

This example returns the following:

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

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

Example 7

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

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

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

This example returns the following:

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

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

Standard Statistics

Here are the standard statistics and how they are aggregated.

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

Derived Statistics

For measures associated with these statistics, the system will use the values that are being aggregated to determine the result value based on the statistic. For example, for

AVG, it will take the SUM of the values and divide it by the N of the values. Here are the derived statistics:

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

SAS OLAP Security Totals and Permission Conditions

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

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

Example 1— Applying the SECURITY_SUBSET Option to an MDX Query

Below is an example of an MDX query:

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

This query has the following applied permission condition:

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

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

<i>MeasuresLevel</i>			<i>sales_sum</i>
<i>All Dealers</i>	<i>dealer</i>	<i>dest</i>	
[-] [v] All Dealers	All Dealers		\$229,000.00
	[-] [v] Smith	Smith	\$108,000.00
		CA	\$12,000.00
		CT	\$15,000.00
		NC	\$25,000.00
		NJ	\$56,000.00

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

<i>MeasuresLevel</i>			<i>sales_sum</i>
<i>All Dealers</i>	<i>dealer</i>	<i>dest</i>	
[-] [v] All Dealers	All Dealers		\$108,000.00
	[-] [v] Smith	Smith	\$108,000.00
		CA	\$12,000.00
		CT	\$15,000.00
		NC	\$25,000.00
		NJ	\$56,000.00

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

Example 2— Applying the SECURITY_SUBSET Option to an MDX Query

Here is a second example of an MDX query:

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

This query has the following applied permission condition:

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

Here is the resulting data table:

<i>MeasuresLevel</i>		sales_sum
<i>All Date</i>	<i>dte</i>	
<input type="checkbox"/> <input checked="" type="checkbox"/> All Date	All Date	\$229,000.00
	January	\$54,000.00
	February	\$64,000.00
	March	\$59,000.00
	April	\$34,000.00
	May	\$18,000.00

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

<i>MeasuresLevel</i>		sales_sum
<i>All Date</i>	<i>dte</i>	
<input type="checkbox"/> <input checked="" type="checkbox"/> All Date	All Date	\$108,000.00
	January	\$10,000.00
	February	\$25,000.00
	March	\$49,000.00
	April	\$24,000.00

Note that the *members* in the output data are the same for both queries. It is the *date* member values that are different. The table values in the first data set reflect sales values for All Dealers, even if the dealers are not displayed. The table values in the second data set reflect sales values for Smith only. Specifically, you can see that there were no sales for Smith during May. It is when the SECURITY_SUBSET option is applied in the second data table that the sales values reflect only dealer Smith.

Default Member and the All Member

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

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

Virtual Members and Security Totals

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

For example, here is a sample data set.

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

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

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

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

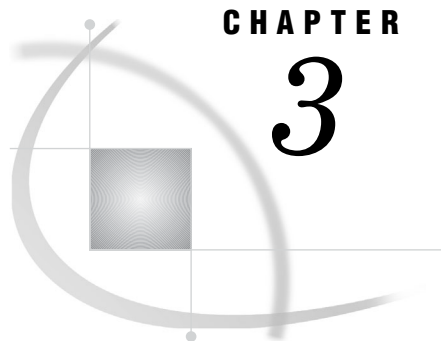
Here is a possible query of that data:

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

Here is the resulting data from that query:

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

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



CHAPTER

3

MDX Usage Examples

<i>Basic Examples</i>	23
<i>Additional Basic Examples</i>	25
<i>Joins and Extractions for Queries Examples</i>	26
<i>Examples of Displaying Multiple Dimensions on Columns and Eliminating Empty Cells</i>	26
<i>Calculated Member Examples</i>	27
<i>Query-Calculated Member Examples</i>	28
<i>Example 1</i>	28
<i>Example 2</i>	28
<i>Example 3</i>	29
<i>Session-Level Calculated Member Examples</i>	29
<i>Example 1</i>	30
<i>Example 2</i>	30
<i>Example 3</i>	30
<i>Example 4</i>	31
<i>Drill-Down Examples</i>	31
<i>Example 1</i>	31
<i>Example 2</i>	32
<i>Example 3</i>	33
<i>Example 4</i>	33
<i>Session-Named Set Examples</i>	34
<i>Example 1</i>	34
<i>Example 2</i>	35
<i>Example 3</i>	35
<i>Example 4</i>	36
<i>Example 5</i>	37
<i>Example 6</i>	38
<i>Example 7</i>	38
<i>Additional Named Set Examples</i>	38

Basic Examples

This topic shows several basic MDX queries. For detailed information on the MDX functions used in these examples see “Basic MDX Queries and Syntax” on page 5 and Appendix 1, “MDX Functions,” on page 41.

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

Example of a simple two-dimensional query:

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

Example of how you can flip the rows and columns:

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

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

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

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

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

Example of the .MEMBERS function:

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

example of the .CHILDREN function:

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

Example of selecting more than one dimension in a tuple:

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


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

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

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

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

Additional Basic Examples

Example of a basic two-dimension table:

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

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

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

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

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

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

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

Joins and Extractions for Queries Examples

Example of the CROSSJOIN function:

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

Example of the UNION function:

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

Example of the EXCEPT function as a query:

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

Example of the EXTRACT function:

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

Examples of Displaying Multiple Dimensions on Columns and Eliminating Empty Cells

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

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

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

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

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

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

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

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

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

Example of the COALESCE EMPTY function:

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

Calculated Member Examples

Example of the WITH MEMBER statement:

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

Example of the WITH MEMBER statement and Format:

```
WITH MEMBER [measures].[target_difference] AS
' [measures].[actualsalessum]-[measures].[predictedsalessum] ' ,
format_string="dollar20.2"
```

Example of the CREATE GLOBAL MEMBER statement:

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

Example of the DEFINE MEMBER statement:

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

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

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

```

SELECT {CROSSJOIN({[time].[yqm].[all yqm]}, {[measures].
[actualsalesum]})} 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. 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

Region	Qty	Total	Average Price
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 goods members from the family level.

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

FROM sales

Here is the resulting output:

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

Example 2

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

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

Here is the resulting output:

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

Example 3

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

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

Here is the resulting output:

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

Example 4

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

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

```

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

```

Here is the resulting output:

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” and “prod in SE”. It shows the quantity and total sales for products that customers in the Northeast and the Southeast purchased.

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

Here is the resulting output:

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

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

Example 6

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

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

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

Example 7

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

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

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

Additional Named Set Examples

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

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

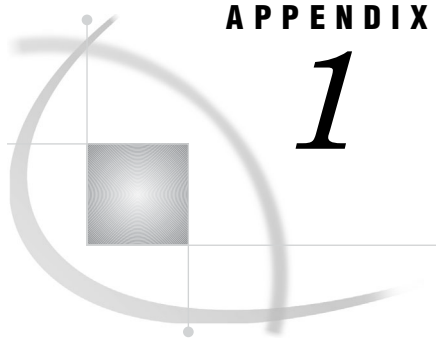
```
        {[products].[all products].[books]} ON ROWS
      FROM [booksnall]
    );
  disconnect from olap;
quit;
```

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

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

(
  DROP SET booksnall.threeyears
) by olap;

disconnect from olap;
quit;
```

APPENDIX

1

MDX Functions

<i>Dimension Functions</i>	41
<i>Hierarchy Functions</i>	42
<i>Level Functions</i>	42
<i>Logical Functions</i>	42
<i>Member Functions</i>	43
<i>Numeric Functions</i>	44
<i>Set Functions</i>	47
<i>String Functions</i>	57
<i>Tuple Functions</i>	58
<i>Miscellaneous Functions and Operators</i>	59

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.
	<code><Member>.Hierarchy</code>
	<code><Level>.Hierarchy</code>

Level Functions

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

Level	returns the level of a member.
	<code><Member>.Level</code>
Levels	returns levels that are specified by a numeric or string expression.
	<code><Dimension>.Levels(<NumericExpression>)</code>
	<code>Levels(<StringExpression>)</code>

Logical Functions

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

IsEmpty	returns TRUE if the evaluated expression is an empty cell value. Otherwise, FALSE is returned.
	<code>IsEmpty(<Value Expression>)</code>
IS	returns TRUE if two compared objects are equivalent. Otherwise, FALSE is returned.
	<code><Object 1>IS Null</code>
	<code><Object 1>IS <Object 2></code>
IsAncestor	returns TRUE if a specified member is an ancestor of another specified member. Otherwise, FALSE is returned.
	<code>IsAncestor(<Member1>,<Member2>)</code>
IsLeaf	returns TRUE if a specified member is a leaf member. Otherwise, FALSE is returned.
	<code>IsLeaf(<Member>)</code>
IsSibling	returns TRUE if a specified member is a sibling of another specified member. Otherwise, FALSE is returned.

IsSibling(<Member1>,<Member2>)

Member Functions

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

Ancestor	returns the ancestor of a member at a specified level or distance. Ancestor(<Member>,<Level>) Ancestor(<Member>,<Numeric Expression>)
ClosingPeriod	returns the last sibling among the descendants of a member to a specified level. ClosingPeriod([<Level>[,<Member>]])
Cousin	returns the child member with the same relative position under its parent member as the specified child member. Cousin (<Member1>,<Member2>)
CurrentMember	returns the current member of a dimension or hierarchy during an iteration over a set of members of that dimension or hierarchy. <Dimension>.CurrentMember <Hierarchy>.CurrentMember
DataMember	returns a system-generated data member that is associated with a non-leaf member of a dimension. <Member>.DataMember
DefaultMember	returns the default member of a dimension or hierarchy. <Dimension>.DefaultMember <Hierarchy>.DefaultMember
FirstChild	returns the first child of a specified member. <Member>.FirstChild
FirstSibling	returns the first child of the parent of a specified member. <Member>.FirstSibling
Item	returns a member from a specified tuple. Alternatively, it returns a tuple from a set.

`<Tuple>.Item(<Index>)`

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

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	<p>returns the average value of a numeric expression that is evaluated over a set.</p> <pre>Avg(<Set>[,<Numeric Expression>])</pre> <p>Example: The following example shows a moving average across all dimensions of time.</p> <pre>Avg(time.currentmember.lag (if(time.currentmember.level is time.month_num,2, if(time.currentmember.level is time.quarter,1,0))) :time.currentmember, measures.[total_retail_pricesum])</pre> <p>The Total_Retail_PriceSUM is included in the following query to see the difference between the moving average and the total retail price.</p> <pre>SELECT {[measures].[movingaverage],[measures]. [total_retail_pricesum] } ON COLUMNS , {[time].[yqm].[all yqm].children } ON ROWS FROM [orionstar]</pre>
CoalesceEmpty	<p>returns a coalesced value. This value is derived when an empty cell value is coalesced to a number or string.</p> <pre>CoalesceEmpty(<Numeric Expression>[,<Numeric Expression>])</pre>
Correlation	<p>returns the correlation of two series that are evaluated over a set.</p> <pre>Correlation(<Set>,<Numeric Expression>[,<Numeric Expression>])</pre>
Count	<p>depending on the collection, returns the number of items in a collection.</p> <pre><Dimension> <Hierarchy>.Levels.Count</pre> <pre><Tuple>.Count</pre> <pre><Set>.Count</pre> <pre>Count(<Set>[,ExcludeEmpty IncludeEmpty])</pre>
Covariance	<p>returns the population covariance of two series that are evaluated over a set by using the biased population formula.</p> <pre>Covariance(<Set>,<Numeric Expression>[,<Numeric Expression>])</pre>
CovarianceN	<p>returns the sample covariance of two series that are evaluated over a set by using the unbiased population formula.</p> <pre>CovarianceN(<Set>,<Numeric Expression>[,<Numeric Expression>])</pre>
DistinctCount	<p>returns the number of distinct, non-empty tuples in a set.</p> <pre>DistinctCount(<Set>)</pre>
Iif	<p>returns one of two numeric or string values that are determined by a logical test.</p> <pre>IIF(<Logical Expression>, <Numeric Expression1>, <Numeric Expression2>)</pre>

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

LinRegIntercept	calculates the linear regression of a set and returns the value of b in the regression line $y = ax + b$. <code>LinRegIntercept(<Set>,<Numeric Expression>[,<NumericExpression>])</code>
LinRegPoint	calculates the linear regression of a set and returns the value of y in the regression line $y = ax + b$. <code>LinRegPoint(<NumericExpression>,<Set>,<NumericExpression>[,<Numeric Expression>])</code>
LinRegR2	calculates the linear regression of a set and returns R^2 (the coefficient of determination). <code>(Set, Numeric Expression[, Numeric Expression])</code>
LinRegSlope	calculates the linear regression of a set and returns the value of a in the regression line $y = ax + b$. <code>LinRegSlope(<Set>,<NumericExpression>[,<NumericExpression>])</code>
LinRegVariance	calculates the linear regression of a set and returns the variance associated with the regression line $y = ax + b$. <code>(Set, Numeric Expression[, Numeric Expression])</code>
Max	returns the maximum value of a numeric expression that is evaluated over a set. <code>Max(<Set>[,<Numeric Expression>])</code>
Median	returns the median value of a numeric expression that is evaluated over a set. <code>Median(<Set>[,<Numeric Expression>])</code>
Min	returns the minimum value of a numeric expression that is evaluated over a set. <code>Min(<Set>[,<Numeric Expression>])</code>
Ordinal	returns the zero-based ordinal value that is associated with a level. <code><Level>.Ordinal</code>
Range	returns the range, which is the difference between the maximum and minimum value of a numeric expression that is evaluated over a set. <code>Range (<Set>[,<Numeric Expression>])</code>
Rank	returns the one-based rank of a specified tuple in a specified set. <code>Rank(<Tuple>,<set>[,<Calc Expression>])</code>
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. <code>RollupChildren(<Member>,<String Expression>)</code>

Stdev	using the unbiased population formula, returns the sample standard deviation of a numeric expression that is evaluated over a set. <code>Stdev(<set>[,<Numeric Expression>])</code>
StdevP	using the biased population formula, returns the population standard deviation of a numeric expression that is evaluated over a set. <code>StdevP(<set>[,<Numeric Expression>])</code>
StrToValue	returns a value from a string expression. <code>StrToValue(<StringExpression>)</code>
Sum	returns the sum of a numeric expression that is evaluated over a set. <code>Sum(<Set>[,<Numeric Expression>])</code>
Value	returns the value of a measure. <code><Member>.Value</code>
Var	using the unbiased population formula, returns the sample variance of a numeric expression that is evaluated over a set. <code>Var(<Set>[,<Numeric Expression>])</code>
VarP	using the biased population formula, returns the population variance of a numeric expression that is evaluated over a set. <code>VarP(<Set>[,<Numeric Expression>])</code>

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
```

The following example references 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>])
```

The following example shows retrieving the Ancestor at a particular level.

```
WITH MEMBER [measures].[product family sales] AS
  '(ancestor([product].currentmember,[product].
  [product family]),[measures].[unit sales])'
```

```
SELECT
  {[measures].[unit sales],[measures].[product family sales]}
ON COLUMNS,
  {[product].members} ON ROWS
FROM [sales]
```

Level

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

```
Ancestors(<member>,[<level>[ , <Anc_flags>]])
```

If no *<Level>* or *<Anc_flags>* arguments are specified, then the function behaves as in the following syntax:

```
Ancestors(<member>, <member>.Level, SELF_BEFORE_AFTER)
```


Distance

returns the set of ancestors of a member. The set of ancestors is specified by *<member>* and is *<distance>* steps away in the hierarchy. Optionally, the set is modified by a flag that is specified in *<Anc_flags>*. Specifying a *<Distance>* of 0 returns a set consisting only of the member that is specified in *<Member>*.

```
Ancestors(<member>, <distance>[, <Anc_flags>])
```

Table A1.1 Ancestor Flag Options

Options	Value Returned
AFTER	Returns ancestor members from all levels between <i><Level></i> and <i><Member></i> , including <i><Member></i> itself, but not member(s) found at <i><Level></i> .
BEFORE	Returns ancestor members from all levels above <i><Level></i> .
BEFORE_AND_AFTER	Returns ancestor members from all levels above the level of <i><Member></i> except members from <i><Level></i> .
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 <i><Level></i> only. Includes <i><Member></i> , if and only if <i><Level></i> that is specified is the level of <i><Member></i> .
SELF_AND_AFTER	Returns ancestor members from <i><Level></i> and all levels below <i><Level></i> , down to and including <i><Member></i> .
SELF_AND_BEFORE	Returns ancestor members from <i><Level></i> and all levels between and above <i><Member></i> .
SELF_BEFORE_AFTER	Returns ancestor members from all levels above the level of <i><Member></i> , including <i><Member></i> and member(s) at <i><Level></i> .

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

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

Table A1.2 Ancestor Expressions and Returned Values

Expressions	Value Returned
Ancestors (USA)	All members
Ancestors (Wake, Counties)	USA

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

Ascendants returns all ancestors of the specified member up through the root level, including the member itself.

`Ascendants (<Member>)`

Example: The following example shows retrieving the member at the specific level above and the current member dynamically using the Ascendants function (set function).

```
SELECT
  {Ascendants([time].[all yqm].[2002])} ON COLUMNS ,
  {Descendants([geography].[global].[all global],2)} ON ROWS
FROM [orionstar]
WHERE [measures].[total_retail_pricesum]
```

Axis returns a set that is defined in an axis. Axis (0) pertains to *row* members, where Axis (1) pertains to *column* members.

`Axis (<Numeric Expression>)`

Example:

```
Axis (0)
Axis (1)
```

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

BottomCount returns a specified number of items from the bottom of a set.

`BottomCount(<Set>,<Count>[,Numeric Expression][,<True|False>])`

Example: The following example shows displaying the bottom 5 products in the Clothes product category for 2002Q4 using the BottomCount function.

```
SELECT
  {[time].[yqm].[all yqm].[2002].[2002q4]} ON COLUMNS ,
  {BottomCount([product].[all product].[clothes & shoes].[clothes].
  children ,5,[measures].[total_retail_pricesum])} ON ROWS
FROM [orionstar]
WHERE [measures].[Total_retail_pricesum]
```

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

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

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

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

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

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

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

BottomSum sorts a set by using a numeric expression and returns the specified number of

bottommost elements whose sum is at least a specified value.

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

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

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

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

Children returns the children of a member.

<Member>.Children

Crossjoin returns the cross-product of two sets.

Crossjoin(<Set1>,<Set2>)

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

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

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

Table A1.3 Descendants Flag Options

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

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>*.

	<pre>DrilldownLevelBottom(<Set>,<Count>[,[<Level>] [,<Numeric Expression>]])</pre>
	<p><i>Note:</i> Constant numeric expressions should not be entered for this function. △</p>
Drilldown LevelTop	<p>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>.</p>
	<pre>DrilldownLevelTop(<Set>,<Count>[,[<Level>] [,<Numeric Expression>]])</pre>
	<p><i>Note:</i> Constant numeric expressions should not be entered for this function. △</p>
Drilldown Member	<p>drills down to the members in a specified set that are present in a second specified set.</p>
	<pre>DrilldownMember(<Set1>,<Set2>[,Recursive])</pre>
Drilldown MemberBottom	<p>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.</p>
	<pre>DrilldownMemberBottom(<Set1>,<Set2>,<Count>[, [<Numeric Expression>] [,Recursive]])</pre>
	<p><i>Note:</i> Constant numeric expressions should not be entered for this function. △</p>
Drilldown MemberTop	<p>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.</p>
	<pre>DrilldownMemberTop(<Set1>,<Set2>,<Count>[, [<Numeric Expression>] [,Recursive]])</pre>
	<p><i>Note:</i> Constant numeric expressions should not be entered for this function. △</p>
DrillupLevel	<p>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.</p>
	<pre>DrillupLevel(<Set>[,<Level>])</pre>
DrillupMember	<p>drills to the members in a specified set that are present in a second specified set.</p>
	<pre>DrillupMember(<Set1>,<Set2>)</pre>
Except	<p>locates the difference between two sets and optionally retains duplicates.</p>
	<pre>Except(<Set1>,<Set2>[,All])</pre>
Extract	<p>returns a set of tuples from extracted dimension elements.</p>

	<code>Extract(<Set>,<Dimension>[,<Dimension>...])</code>
Filter	returns the set that results from filtering a specified set that is based on a search condition. <code>Filter(<Set>,<Search Condition>)</code>
Generate	applies a set to each member of another set and is joined to the resulting sets. <code>Generate(<Set1>,<Set2>[,<All>])</code>
Head	returns the first specified number of elements in a set. <code>Head(<Set>[,<Numeric Expression>])</code>
Hierarchize	orders the members of a set in a hierarchy. <code>Hierarchize(<Set>)</code>
Intersect	returns the intersection of two input sets and optionally retains duplicates. <code>Intersect(<Set1>,<Set2>[,<All>])</code>
LastPeriods	returns a set of members prior to and including a specified member. <code>LastPeriods(<Index>[,<Member>])</code>
Members	returns the set of members in a dimension, level, or hierarchy. <code><Dimension>.Members</code> <code><Level>.Members</code> <code><Hierarchy>.Members</code>
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 <code>PeriodsToDate()</code> function with the level defined as <code>Month</code> . <code>Mtd([<Member>])</code>
NameToSet	returns a set that contains a single member. The set is based on a string expression that contains a member name. <code>NameToSet(<Member Name>)</code>
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. <code>NonEmptyCrossjoin(<Set1>[,<Set2>][,<Set3>...] [,<Crossjoin Set Count>])</code>

Order	<p>arranges members of a specified set and optionally preserves or breaks the hierarchy.</p> <pre>Order(<Set>[, [<Numeric Expression>][, BASC BDESC]])</pre> <pre>Order(<Set>[, [<String Expression>][, BASC BDESC]])</pre> <pre>Order(<Set>, <Numeric Expression>[, ASC DESC])</pre> <pre>Order(<Set>, <String Expression>[, ASC DESC])</pre> <p><i>Note:</i> Constant numeric expressions should not be entered for this function. △</p>
PeriodsToDate	<p>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.</p> <pre>PeriodsToDate([<Level>[, <Member>]])</pre>
Qtd	<p>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.</p> <pre>Qtd([<Member>])</pre>
SetToList	<p>takes an MDX set as input and returns a list of variable-length parameters. Functions that take variable-length parameter lists include the descriptive statistics functions (SUM(), MEAN(), PCTL(), etc.), COALESCE(), and CHOOSEC/N(). In addition, it takes the character parameters CAT/S/T/X() and COALESCEC().</p> <pre>Settolist(<Set>[, <Numeric Expression Character Expression>])</pre>
Siblings	<p>returns the siblings of a specified member, including the member itself.</p> <pre><Member>.Siblings</pre>
StripCalculated Members	<p>returns a set that is generated by removing calculated members from a specified set.</p> <pre>StripCalculatedMembers(<Set>)</pre>
StrToSet	<p>returns a set that is constructed from a specified string expression in Multidimensional Expressions (MDX) format.</p> <pre>StrToSet (<String Expression>)</pre>
Subset	<p>returns a subset of tuples from a specified set.</p> <pre>Subset(<Set>, <Start>[, <Count>])</pre>
Tail	<p>returns a subset from the end of a set.</p> <pre>Tail(<Set>[, <Count>])</pre>

ToggleDrillState	<p>Toggles the drill state of members.</p> <pre>ToggleDrillState(<Set1>,<Set2>[,RECURSIVE])</pre> <p><i>Note:</i> 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. Δ</p>
TopCount	<p>returns a specified number of items from the topmost members of a specified set.</p> <pre>TopCount(<Set>,<Count>[,<Numeric Expression>,<True False>])</pre> <p><i>Note:</i> 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. Δ</p> <p><i>Note:</i> Constant numeric expressions should not be entered for this function. Δ</p>
TopPercent	<p>sorts a set and returns the topmost elements, whose cumulative total is at least a specified percentage.</p> <pre>TopPercent(<Set>,<Percentage>[,<Numeric Expression>])</pre> <p><i>Note:</i> Constant numeric expressions should not be entered for this function. Δ</p>
TopSum	<p>sorts a set and returns the topmost elements whose cumulative total is at least a specified value.</p> <pre>TopSum(<Set>,<Value>[,<Numeric Expression>])</pre> <p><i>Note:</i> Constant numeric expressions should not be entered for this function. Δ</p>
Union	<p>returns a set that is generated by the union of two sets. Optionally, duplicate members are retained.</p> <pre>Union(<Set1>,<Set2>[,All])</pre>
VisualTotals	<p>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.</p> <pre>VisualTotals (<Set>,<Pattern>)</pre>
Wtd	<p>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.</p> <pre>Wtd([<Member>])</pre>
Ytd	<p>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.</p> <pre>Ytd([<Member>])</pre>

String Functions

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

CoalesceEmpty coalesces an empty cell value to a number or string and returns the coalesced value.

```
CoalesceEmpty(<String Expression>[,<String Expression>]...)
```

Generate returns a concatenated string that is created by evaluating a string expression over a set.

```
Generate(<Set>,<String Expression>[,Delimiter])
```

Iif returns one of two numeric or string values that are determined by a logical test.

```
Iif(<Logical Expression>,<String Expression1>,  
<String Expression2>)
```

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

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. Δ
Put	returns a string that contains the formatted output based on a SAS format. <code>Put(<Numeric Expression>,<String Expression>)</code> <code>Put(<String Expression>,<String Expression>)</code>
SetToStr	constructs a string in Multidimensional Expressions (MDX) format from a set. <code>SetToStr(<Set>)</code>
TupleToStr	returns a string in Multidimensional Expressions (MDX) format from a specified tuple. <code>TupleToStr(<Tuple>)</code>
UniqueName	returns the unique name of a specified level, dimension, member, or hierarchy. <code><Level>.UniqueName</code> <code><Dimension>.UniqueName</code> <code><Member>.UniqueName</code> <code><Hierarchy>.UniqueName</code>
UserName	returns the domain name and user name of the current connection. <code>UserName</code>
<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. <code><Set>.Current</code>
---------	---

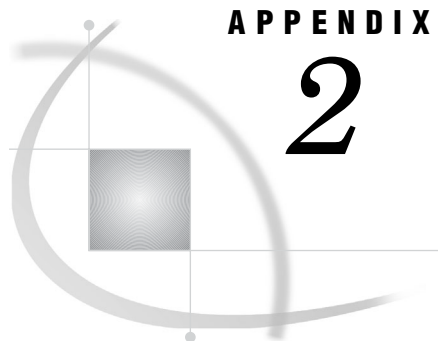
Item	returns a member from a specified tuple. Alternatively, it returns a tuple from a set. <code><Set>.Item(<Index>)</code> <i>Note:</i> If a member is returned, then it is a member function, not a tuple function. △
StrToTuple	constructs a tuple from a specified string expression in Multidimensional Expressions (MDX) format. <code>StrToTuple(<String expression>)</code>

Miscellaneous Functions and Operators

Miscellaneous Functions and Operators

Function or Operator	Explanation
, (comma operator)	An operator that is used to combine tuples to construct sets. For example: <code>{[time].[all time].[2001].[january], [time].[all time].[2001].[February],[time].[all time].[2001].[march]}</code> , or to combine members to construct tuples such as <code>([Time].[January 2001],[Geography].[U.S.A])</code>
: (colon operator)	An operator that is used in part to construct sets and tuples. It replaces a series of comma operators. For example: <code>{[Time].[all Time].[2001].[January] : [Time].[all Time].[2001].[March]}</code>
{ } (braces)	The SET constructor operator.
* (asterisk operator)	The alternative for CrossJoin. If you use a single operator, it is a direct replacement. You can nest CrossJoins by stringing additional operators and sets. For example, these expressions are equivalent: <code>A * B – CrossJoin(A, B)</code> <code>A * B * C – CrossJoin(A, CrossJoin(B, C))</code>
+ (plus operator for sets)	An alternative to union().
+ (plus operator for strings)	A concatenation of two strings.
/**/ (style comments)	Cause the OLAP server to ignore anything between the initial token and final token. These comments can span lines. NOTE: these comments do NOT nest.

Function or Operator	Explanation
// (style comments)	Cause the OLAP server to ignore anything after the double slash until the end of the line. These comments can NOT span lines.
- - (style comments)	Cause the OLAP server to ignore anything after the double dash until the end of the line. These comments can NOT span lines. These are essentially the same as the double-slash comments.
NON EMPTY	When applied to an axis, causes the OLAP Server to post-process the results of the axis set and remove any tuples which have empty results. For the SAS OLAP Server, it works with CrossJoin to provide "Optimize NON EMPTY CrossJoin" performance improvements.
<Set> AS aliasname	Creates an alias for an intermediate set. An intermediate set is one which is generated during the evaluation of an axis and has one or more functions operating on it.
Supports TRUE and FALSE	TRUE and FALSE conditional operators are supported in MDX queries
Call<UDF Name>	Executes a void returning user-defined function.
MAX SET SIZE	Limits the size of sets that the OLAP server creates. A value of 0 indicates there is no limit. The default is 1,000,000 components, where components are defined as the number of tuples in the set times the number of dimensions in each tuple. This function enables the administrator to control the system resources that are used by individual queries.



APPENDIX

2

Recommended Reading

Recommended Reading 61

Recommended Reading

Here is the recommended reading list for this title:

- SAS OLAP Server: User's Guide*
- SAS Providers for OLE DB: Cookbook*
- SAS Language Reference: Concepts*
- SAS Language Reference: Dictionary*
- SAS Intelligence Platform: Overview*
- SAS Intelligence Platform: Administration Guide*
- SAS Intelligence Platform: Data Administration Guide*
- SAS Intelligence Platform: System Administration Guide*
- SAS Intelligence Platform: Application Server Administration Guide*
- SAS Intelligence Platform: Security Administration Guide*
- SAS Intelligence Platform: Web Application Administration Guide*
- SAS Intelligence Platform: 9.1.3 to 9.2 Migration Guide*
- SAS Intelligence Platform: Desktop Application Administration Guide*
- SAS Data Integration Studio: User's Guide*
- SAS Web OLAP Viewer for Java: Help*
- SAS OLAP Server Monitor Plug-In for SAS Management Console: Help*
- SAS Management Console Main Application: Help*
- SAS Companion that is specific to your operating environment

For a complete list of SAS publications, go to support.sas.com/bookstore. If you have questions about which titles you need, please contact a SAS Publishing Sales Representative at:

SAS Publishing Sales
 SAS Campus Drive
 Cary, NC 27513
 Telephone: 1-800-727-3228
 Fax: 1-919-531-9439
 E-mail: sasbook@sas.com
 Web address: support.sas.com/bookstore

Customers outside the United States and Canada, please contact your local SAS office for assistance.

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.

Application Response Measurement

the name of 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. Short form: ARM.

ARM

See Application Response Measurement.

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 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 and 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 language

See multidimensional expressions 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 server

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

MOLAP

See multidimensional online analytical processing.

multidimensional expressions language

a standardized, high-level language that is used to query multidimensional data sources. The MDX language is the multidimensional equivalent of SQL (Structured Query Language). Short form: MDX language.

multidimensional online analytical processing

a type of OLAP that stores aggregates in multidimensional database structures. Short form: MOLAP.

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.

ODBO

See OLE DB for OLAP.

OLAP

See online analytical processing.

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. Short form: ODBO. See also multidimensional expressions language.

online analytical processing

a software technology that enables users to dynamically analyze data that is stored in multidimensional database (MDDDB) tables. Short form: OLAP.

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 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

See symmetric multiprocessing.

SQL

See Structured Query Language.

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. Short form: SQL.

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. Short form: SMP.

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 symmetric multiprocessing, thread-enabled operating system, and 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

A

aggregate function
 derived statistics with 14
 analysis variables 25

C

calculated members 2
 examples 27
 .CHILDREN function 24
 COALESCE EMPTY function 27
 CREATE DDL statement 8
 CREATE GLOBAL MEMBER statement 27
 CROSSJOIN function 25, 26
 cubes
 concepts 1

D

DDL syntax 8
 DEFINE MEMBER statement 27
 derived statistics 18
 with aggregate function 14
 dimension functions 41
 dimensionality 3
 dimensions 2
 displaying multiple dimensions on columns 26
 more than one in a tuple 24
 drillthrough 7
 examples 31
 maximum number of rows 7
 table access at query time 7
 user-defined formats and 8
 DRILLTHROUGH statement 7
 DROP DDL statement 9
 DROP MEMBER statement 28

E

empty values 26
 examples
 basic examples 23
 calculated members 27
 drill-down 31
 joins and extractions for queries 26
 query-calculated members 28
 session-level calculated members 29
 session-named sets 34

EXCEPT function 26
 expressions
 function arguments and return types 9
 SAS functions and 9
 EXTRACT function 26

F

flipping rows and columns 24
 floating-point representation 9
 formats
 user-defined formats and drillthrough 8
 fractions 10
 functions
 arguments and return types 9
 .CHILDREN 24
 COALESCE EMPTY 27
 CROSSJOIN 25, 26
 dimension functions 41
 EXCEPT 26
 EXTRACT 26
 hierarchy functions 42
 in MDX expressions 9
 level functions 42
 lists and tables of 41
 logical functions 42
 member functions 43
 .MEMBERS 24
 NON EMPTY 26
 numeric functions 44
 numeric precision 9
 operators 59
 set functions 47
 string functions 57
 tuple functions 58
 UNION 26

G

global scope calculated members 2

H

hierarchies 2
 hierarchy functions 42

J

joins 26

K

keywords
 NON EMPTY 25
 reserved 11

L

level functions 42
 levels 2
 logical functions 42

M

MDX 1
 concepts 1
 measures 2
 default measure 24
 member functions 43
 members 2
 getting a range of 24
 .MEMBERS function 24
 Microsoft Analysis Services 2000 10
 Multidimensional Expressions
 See MDX

N

NON EMPTY function 26
 NON EMPTY keyword 25
 nonmeasure-based calculated members 2
 numeric functions 44
 numeric precision 9
 fractions 10
 magnitude versus precision 10
 TRUNC function 10

O

operators 59

Q

queries 5
 examples 24, 26
 syntax 5
 query scope calculated members 2
 examples 28

R

reserved keywords 11

S

SELECT clause 6
 SELECT statement 5
 session-named sets 34
 session scope calculated members 2
 examples 29
 set functions 47
 sets 3
 session-named 34
 slicer 6
 statistics
 derived statistics 18
 derived statistics with aggregate function 14
 standard statistics and aggregation 18
 string functions 57

T

tables
 two-dimensional 25
 TRUNC function 10
 tuple functions 58
 tuples 2
 making combinations 25
 more than one dimension in 24
 two-dimensional tables 25
 selecting specific columns 25
 selecting specific rows 25
 with analysis variable 25

U

UNION function 26
 user-defined formats
 drillthrough and 8

W

WHERE clause 6
 WITH clause 5
 WITH MEMBER statement 27

Your Turn

We welcome your feedback.

- If you have comments about this book, please send them to **yourturn@sas.com**. Include the full title and page numbers (if applicable).
- If you have comments about the software, please send them to **suggest@sas.com**.

