Chapter 10

MDX

Chapter Outline

What You Will Learn in This Chapter

- How is data retrieved from a cube and used in calculations?
- What are the basic elements of an OLAP cube?
- What are the main objects in MDX queries?
- What is the primary structure of an MDX query?
- How are MDX queries written and what basic data do they provide?
- How are computations and new measures defined?
- How does MDX handle complex computations that cross levels or rows of data?
- What other MDX functions are commonly used in business problems?

Dallas Cowboys Merchandizing

Merchandising is an important element for any sports team, including those in the National Football League (NFL). The Dallas Cowboys are no exception—in fact, Bill Priakos notes that "nobody gives out their exact numbers, but we feel comfortable we are in the upper echelon," along with teams like Manchester United and the Chicago Bulls. As one of the top merchandizer, the Cowboys sold more than $100 million worth of gear—shirts, jerseys, and other items in 2009. The Tony Romo jersey was estimated to be the most popular sports jersey in the nation in 2008—selling half a million items alone. But, as any clothing retailer knows, retail sales are challenging. In fact, all of the NFL teams except the Cowboys outsource all of the retail operations to Reebok. In 2002, the Cowboys chose to control all manufacturing, sales, and distribution themselves. The merchandizing organization installed software from Microsoft to handle basic sales and data tasks, but ended up buying software from Tableau Software, Inc. to answer the harder questions. Creation of a digital dashboard was a key element in tying together all of the underlying data. Priakos notes that the software lets "us find answers instantly." Such as answering "How are Internet sales of our jerseys doing?" or "Where do our jerseys sell well outside of Texas?" Without the dashboards and visualization software from Tableau, it used to take 30 minutes to create queries to answer these types of questions. [Lai 2009]

Digital dashboards are constructed from key performance indicators and many of the steps can be automated using MDX tools.

Eric Lai, "BI Visualization Tool helps Dallas Cowboys Sell More Tony Romo Jerseys," *Computerworld*, October 8, 2009. http://www.computerworld.com/s/article/9139140/BI_visualization_tool_helps_Dallas_Cowboys_sell_more_Tony_Romo_jerseys

Introduction

How is data retrieved from a cube and used in calculations? **Multi-dimension expression or MDX** is a query language for OLAP cubes. It appears to have been initially defined by Microsoft but has been adopted by almost all of the major data mining tool providers, including SAS and IBM. Documentation and examples can be found on the Web both from official (corporate) sources and individual writers. The big question is whether you need to know how to write (or even read) MDX queries. For most business users of data mining, you can simply use the browsing tools created by various vendors and explore cubes with the graphical designers. Microsoft's PivotTable—widely available in Excel—is a good example of a tool that is easy to use with drag-and-drop features.

In some ways, this chapter is optional. However, it turns out that the foundations of MDX are relatively straightforward; and some advanced problems might be easier to solve using MDX directly instead of trying to force a graphical browser to do what you want. Also, some situations call for integrating OLAP data into other tasks, such as spreadsheets, Excel, or other tools. In these cases, the power of MDX is useful because it can be written as a text query, passed to the server, and the results returned directly to an application for further analysis or processing.

A few writers have suggested that MDX is similar to SQL, but these two tools are completely different. Confusingly, MDX uses similar keywords (SELECT, FROM, WHERE), but they have no relationship to anything in SQL. More importantly, the goals are completely different. SQL is designed to retrieve data from relational tables. MDX is designed to retrieve data from an OLAP cube. Consequently, one of the first steps in this chapter is to review the concepts of OLAP cubes and to create a sample cube. A key feature of OLAP cubes is that they are designed to work with aggregate data—or multiple subtotals. This difference is critical to MDX—all queries are designed to return subtotal data—and you almost never enter a SUM function; so the process is automatic.

This chapter is relatively short. It reviews basic concepts and terminology for OLAP cubes. Then a section highlights the main concepts in MDX—largely in terms of hierarchical dimensions. The main syntax of MDX is relatively short, but somewhat confusing. The best way to understand MDX is to work with examples. This chapter uses the Rolling Thunder Bicycle company database to generate a cube and show how to apply MDX. The examples use Microsoft SSAS (SQL Server Analysis Services) and Visual Studio to generate the OLAP cubes and process the MDX commands. Because of changes in SSAS and Visual Studio, PivotTables in Excel are also used to provide better displays of the cubes.

OLAP Cube Review

What are the basic elements of an OLAP cube? Recall the main concepts of OLAP cubes from Chapter 3. A **cube** is a way to visualize and explore data relationships. The main data in a cube consists of **measures** which represent variables that are important to the decision. Common business measures include sales revenue, profit, cost, and counts of items sold. Measure variables have to be numeric. A cube presents a way to quickly explore subtotals of the measure data relative to various attributes. Attributes can be derived from a sale (location, date), the product itself (color, size), the customer (age, income), or internal data such as salesperson or division. Ultimately, the question is to examine how the measure subtotals vary based on the different dimensions. For example, are sales in some states consistently higher than those in other states?

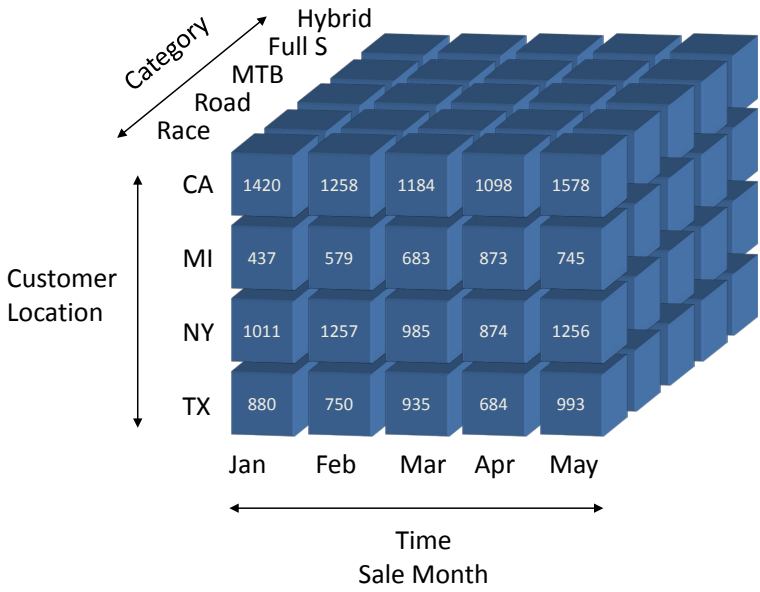


Figure 10.1

Sample OLAP cube for RT Bicycle sales. The numbers measure something of interest in decisions—in this case sales value. This cube has three dimensions: location (state), product category (model type), and time based on the order date.

Figure 10.1 shows a sample cube for Rolling Thunder Bicycles. The measure consists of sales value. The three dimensions pictured are: (1) product category or bicycle model type, (2) customer location or state, and (3) date or time of the sale—expressed in months. The values within a single cell of the cube show the sales value for a specific category to a given state in a specified month. Hyper cubes can have many dimensions—but it is difficult to draw anything above three dimensions. The point is that attributes are displayed as **dimensions** on the cube. Analysts can then use tools to choose dimensions, examine individual values on specified cells, and compute subtotals across dimensions (such as all states for a given month). It is important to remember that the cube displays subtotals.

A key aspect of dimensions is that many of them are **hierarchical**. Some of the hierarchies are "natural" in the sense that everyone commonly uses them. The two main examples are time and location. Cube browsers often have pre-built tools to handle these standard hierarchies. For example, time is often examined from the top down: Year – Quarter – Month – Date. Similarly, geographical location can be written in terms of Nation – State – County – City – ZIP Code. Both dimensions could have multiple hierarchies. For instance, some companies emphasize sales by week which creates a slightly different time hierarchy: Year – Week – Date. Other dimensions can also have hierarchies, which are customized based on the specific problem. For instance, product categories might fall into hierarchies, such as by department. And a company itself is probably organized into departments and sub-departments which could be important for some questions. These hierarchies have to be manually defined but they ultimately behave the same as the natural hierarchies.

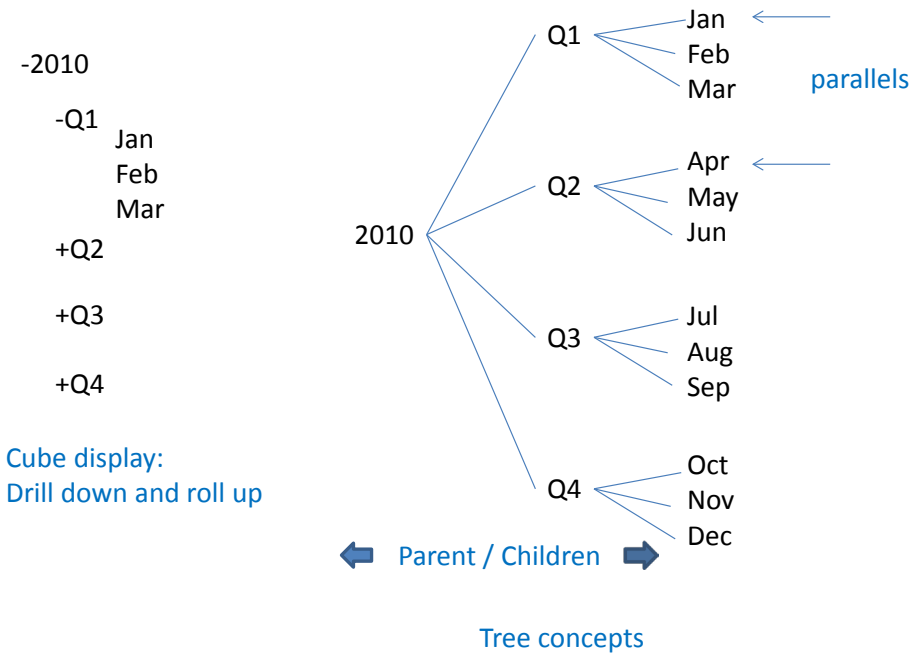


Figure 10.2

Concepts in hierarchies. Visual cube browsers often use drill down and roll up methods to show different levels of subtotals such as the +/- notations on the left side of the diagram. MDX treats hierarchical data as a tree where the higher level is the parent and lower level items are children.

Dimensions and Hierarchies

Cubes and MDX have many options to handle and explore hierarchical data. Figure 10.2 shows two of the main concepts. Cube browsers tend to be visual and the data is often displayed in an expansion format where analysts can drill down (expand) or roll up (compact) the data levels to look at specific items or higher-level subtotals. However, MDX treats hierarchical dimensions as trees. Trees have **levels** where increasing detail is shown at lower levels of the tree. The topmost level is the **root** and items in the bottom level are sometimes referred to as **leaves**. Functions exist to refer to a **parent** level (an item value immediately above the current level), and to list all **children** of any given node. An interesting concept is the ability to examine items in parallel. For instance, it might be necessary to compare data for each month within two different quarters. Parallel tracking could be used to compare the first month in quarters one and two (January versus April) and then move to the second month in each quarter at the same time. These advanced features are built into MDX and are difficult to handle with visual browsers.

Rolling Thunder Bicycles Cube

It is difficult to discuss and understand MDX in abstract terms. All of the concepts are easier to follow by illustrating them with actual data. Rolling Thunder Bicycles provides a good case example. It has enough data with a reasonable number

of dimensions and measures to illustrate most points without overburdening most systems. If you want the ability to write and test MDX queries, you should install the SQL Server version of the RT Bicycles data. Also be sure that Microsoft SSAS is installed on your computer along with at least the Visual Studio client components to develop new business intelligence projects. All of these components can be installed from the Developer version of SQL Server which is available through MSDN or with a time-limited free download version. If the Developer version is not available, use the Enterprise version. The SQL Server RT database is available from the book's Web site: http://www.JerryPost.com.

If you do not install the SQL Server elements, you should carefully check the following figures which show the steps in creating the Cube. The cube structure is important to understand the sample queries. The queries all refer to the structure and data in the database.

The main steps to creating a cube are: (1) In Visual Studio, start a New/Project: Analysis Services, Multidimensional…, (2) Add a new Data Source to connect to the SQL Server RT database, (3) Create a new Data Source View with the desired tables, and (4) Create a new Sales Cube with the desired measures and dimensions.

Creating the new project and the data source should be straightforward. Figure 10.3 shows the tables and relationships needed in the Data Source View for RT Sales. Clearly the Bicycle table is needed because it contains the data measure (Sale Price). The other tables provide data for the dimensions. One catch arises after adding the specified tables. Notice that three tables refer to the City table: Customer, Employee, and RetailStore. All three tables contain address information and link to the City table to get standardized information about cities. But, in terms of sales, only the link between the Customer and City table is useful. It is important to delete the links between Employee -> City and RetailStore -> City. If these links remain, the results will be severely constrained because the links would force all customers, employees, and retail stores to be in the same city. Simply select the two extraneous links and delete them.

The process of creating the Cube uses the wizard, and the main step is to select the tables correctly. First, the bicycle table is selected for its measures. The columns to use for measures need to include at least: Sale Price, List Price, Sales Tax, and Bicycle Count. A couple of other columns might be interesting, such as Frame Size if there is a reason to examine bicycle sizes across states. But by default the wizard selects all numerical columns and many of them will just clutter up the displays later so they should be unchecked. For the attribute dimensions, use all of the other selected tables—which are checked by default.

The Bicycle table includes an OrderDate column which is a useful measure for tracking when bicycles were sold. Some of the other dates might also be useful but just stick with OrderDate for now to keep the problem simpler. However, dates are a natural hierarchy, so it is important to create this date dimension. Right-click the Dimensions entry in the Solution Explorer and add a new dimension. Choose the option to "Generate a time table on the server." This new table becomes a lookup table that contains all possible dates. You need to set the starting and ending dates to January 1, 1994 and December 31, 2015. Then pick the time periods as: Year + Quarter + Month + Date. Change the name to: Date Hierarchy to remind you that it includes multiple levels.

An interesting aspect to dimensions in SSAS is that they standalone—a dimension is really just a lookup table that contains a distinct list of all possible values

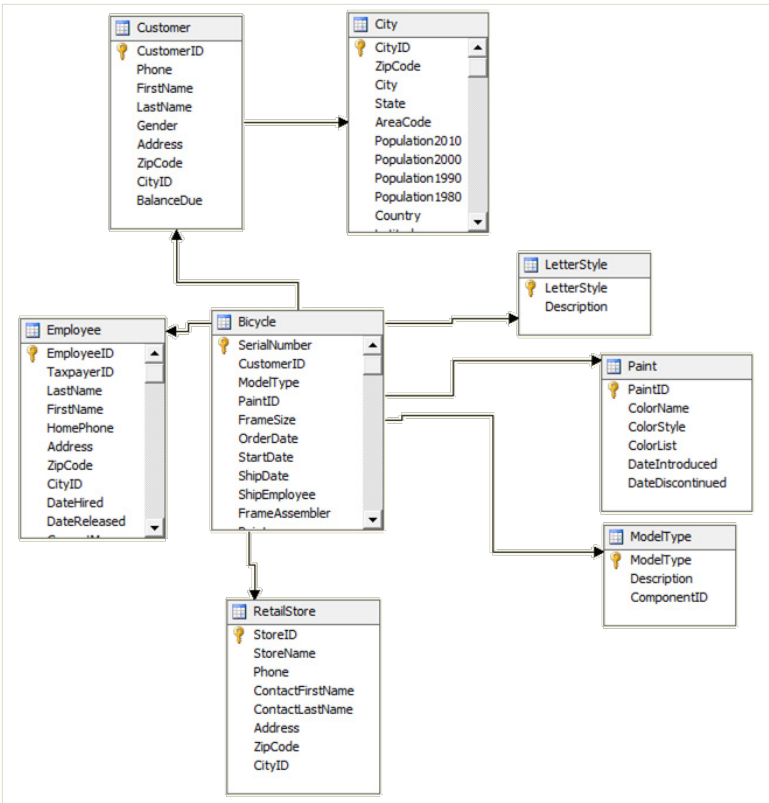


Figure 10.3

Data source view for RT Sales. The Bicycle table has the measures (Sale Price). Then add the tables with the desired attributes: Customer and City, Employee, RetailStore, LetterStyle, Paint, and ModelType. These tables are needed to provide lookup data.

for that dimension. The new Date Hierarchy dimension still needs to be added to the cube and connected to the data. Figure 10.4 shows the basic steps. Open the Cube window and examine its structure. Right-click the Dimensions window to add the new Date Hierarchy dimension.

It is also critical to click the "Dimension Usage" tab. Click the empty box next to the Date Hierarchy. As shown in Figure 10.5, the hierarchy needs to be assigned to the OrderDate column. First set the relationship type as Regular, and set the Granularity to Date if it is not set automatically. In the Measure Group Columns, pick the OrderDate column. This process assigns the Date Hierarchy specifically to the OrderDate column. Finish the wizard and return to the Cube Structure tab.

The last step is to add some elements to the Customer dimension. By default, the Customer dimension includes only ID values, but analysts will find those almost useless. As shown in Figure 10.6, edit the Customer dimension and add the more useful columns by dragging them to the attribute list. Include at least the columns: City, State; and Gender and ZIP Code from the City and Customer tables. By default, the Customer dimension includes only ID values which are Code.ludes only the ID values. Edit the dimension and dra Analysts might also find Income and population to be useful but they are not required for

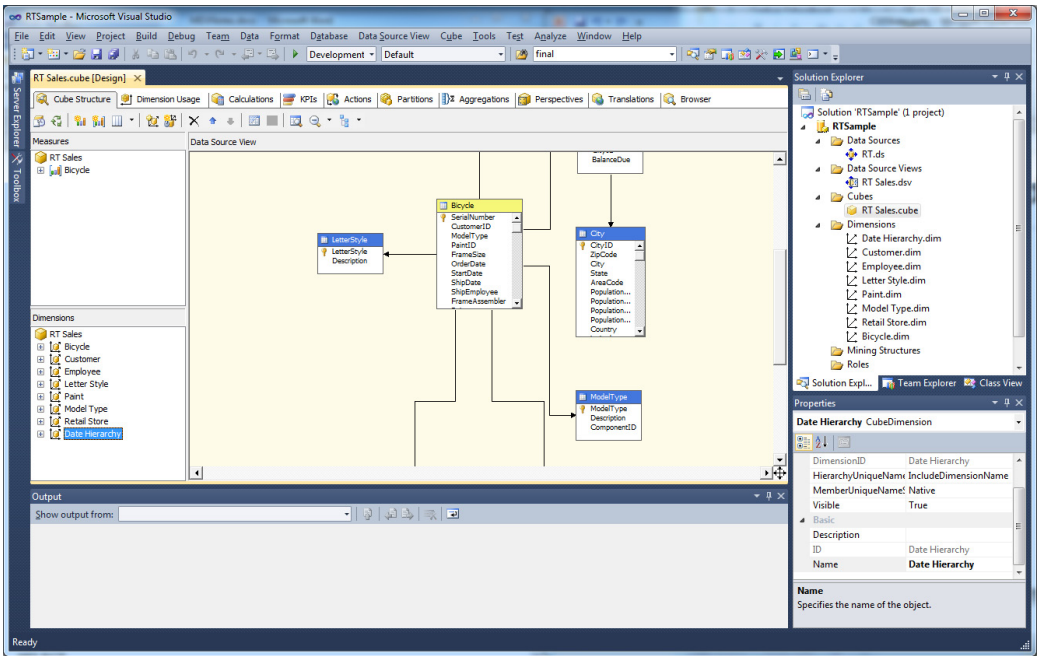


Figure 10.4

Add Date Hierarchy dimension to the cube. Right-click the left Dimensions window and add a new dimension. Select the newly created Date Hierarchy. Then click the Dimension Usage tab.

this demonstration cube. Similarly, you should eventually perform a similar process for the Employee and Retail Store dimensions to add the employee and store names. To verify your work and to populate the cube, it needs to be saved and processed.

The cube now contains measures—notably Sale Price; plus dimensions for Customer, Employee, Letter Style, Paint, Model Type, Retail Store, and the Order Date hierarchy. The name of the cube is RT Sales, but you might choose a different name. Keep this list handy, the exact names will be needed when building MDX queries.

Definitions and Concepts

What are the main objects in MDX queries? MDX is a text language designed to compute and display the subtotals displayed on the cube. It uses most of the same terminology and concepts as the cube. The concepts of totals, dimensions, and hierarchy trees are critical to MDX. Probably the most important thing to remember about MDX is that it is designed to query data from an OLAP cube—so all of the results are subtotals. If no constraints or details are specified, a simple MDX query will retrieve a single total. Specifying dimensions or values generates subtotals for those dimensional values. Think of a cube as a huge collection of SQL GROUP BY subtotals. MDX simplifies the syntax so that only the GROUP BY and WHERE conditions need to be provided.

MDX uses a few key concepts or terms. Measures are numeric values that will be displayed as results—generally summed. Most cubes have a specific measures

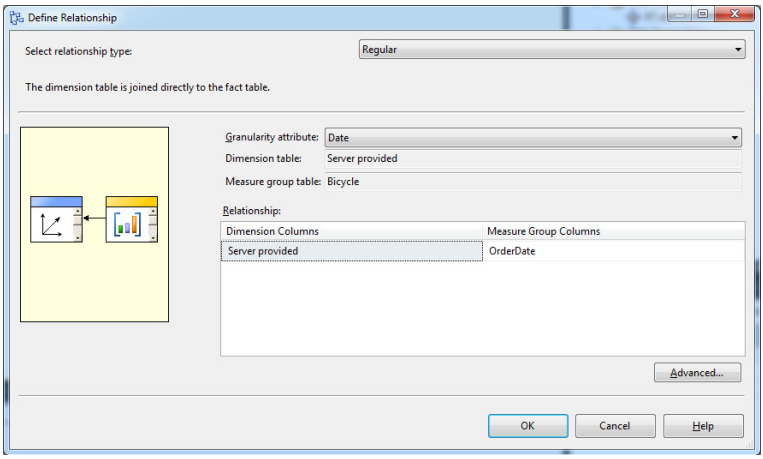
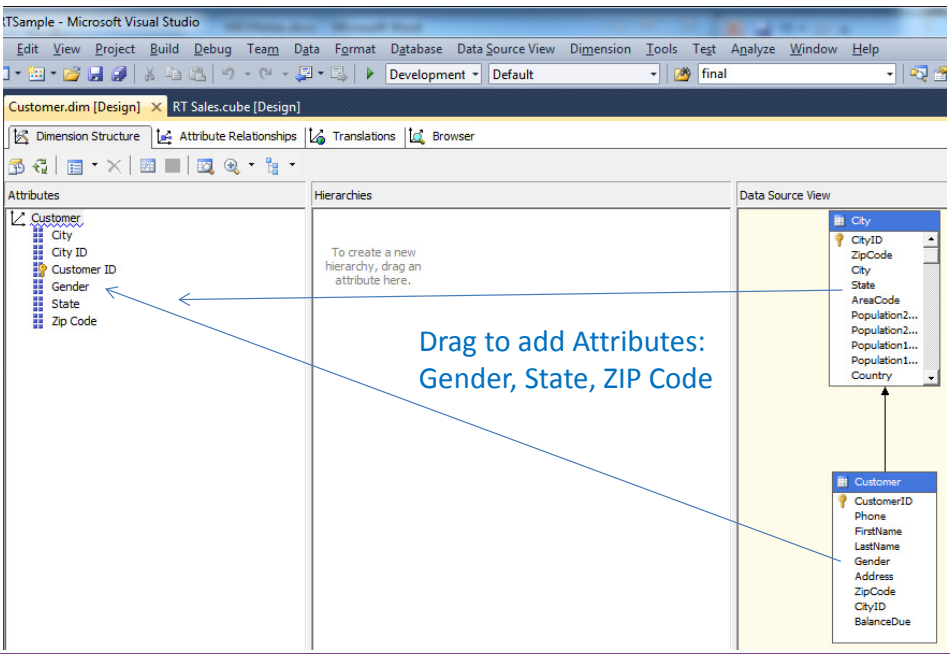


Figure 10.5

Assign Date Hierarchy to the OrderDate column. Set the relationship to Regular, Granularity to Date, and pick the OrderDate column in the Measure Group list.

Figure 10.6

Add elements to the Customer dimension. By default the Customer dimension includes only the ID values. Edit the dimension and drag useful columns into the attribute list: City, State, Gender, and ZIP Code.



set that contains a list of variables that can be used as totals. Within MDX, a specific measure variable is referenced by its full name, such as [Measures].[Sale Price]. The square brackets are used to support names that might contain spaces, key words, or special characters. Essentially, the brackets provide a way to delimit a name that could use characters that might be misinterpreted. In the example, the dot (.) indicates that the [Sale Price] attribute is a **member** of the [Measures] set.

A similar syntax is used to identify items within other dimensions. For example, several model types exist (race, road, mountain, and so on). To indicate a specific value of a model, use a term of the form: [Model Type].[Race]. For hierarchies, use the names from the top down, such as [Date Hierarchy].[Year – Quarter – Month – Date].[Date].members. Note that the last item here is members, which returns a list of all the values within the context (Date in this case). The members (or sometimes allmembers) keyword returns all item values at the specified **level** in the descriptor. It is similar to the children keyword, but technically the term children refers to the entries in a hierarchy tree; whereas members applies to any values—even if no tree is involved. For instance, to get a list of all bicycle model types, use the notation: [Model Type].members. Yes, the choice between "children" and "members" is confusing. When in doubt try "members" first and switch to "children" to see if the difference matters.

MDX notation also relies heavily on the concept of a **set**. Similar to a mathematical set, MDX uses collections of items. These collections can be a simple list of entries—such as a list of the model types; or a complex collection of dimensions—such as combining all model types with a list of states. Sets are defined inside of curly braces { }. They are most commonly used to specify the row and column dimensions for the cube. The terms **rows** and **columns** represent two common **axes** of a cube. The column axis is number 0 and the rows axis is number 1 so they can be referred to as axis(0) or axis(1); or even just 0 and 1. Sometimes the numbers are more convenient, such as when building a cube that contains more than two axes.

MDX borrows the concept of a **tuple** from SQL, which is somewhat of a strange term. In the context of MDX, think of a tuple as a specification of dimension values. For example, a tuple (Race, March 2007, CA) provides specific values for the Model Type, Month, and State dimensions. MDX uses this concept in formulas to define specific values. For instance, a formula with the term [Measures].[Sale Price] refers to the total sale price across all dimensions. To refer to the Sale Price value at a specific set of values or points, use the parentheses notation. For example, ([Model Type].[Race], [Measures].[Sale Price]) computes the total sale price just for the Race model type. Multiple dimensions are supported simply by separating them by commas.

Main Syntax

What is the primary structure of an MDX query? An MDX query can become relatively complex, but as shown in Figure 10.7, the basic structure is fairly simple. A query has four basic elements: (1) calculated values defined at the top using the **WITH MEMBER** command, (2) the sets of dimensions used for each axis specified using the **SELECT** command, (3) the name of the OLAP cube holding the data using the **FROM** command, and (4) **WHERE** conditions that restrict the display to specified dimension values. These keywords might look familiar, but they have nothing to do with SQL; and the syntax and results are completely different.

```
WITH MEMBER          create calculated values
SELECT               define the axes by selecting dimensions
 {   …   } On Columns,
 {   …   } On Rows
FROM [cube name]     name of the cube
WHERE (   …   )      restrict results to specified dimension values
```

Figure 10.7

Basic MDX syntax. The four primary keywords are used to define calculated values, choose the sets of dimensions for each axis, specify the cube name (only one cube), and limit the display to specified attribute values.

Note that the WITH MEMBER and WHERE phrases are optional, so the initial examples will ignore those elements to focus on the underlying goals and syntax first. These two elements along with several useful functions are covered in later sections. At first glance, the main structure of MDX focuses on defining the axes for the OLAP cube and the dimensions to assign to each axis (in the SELECT section). The WHERE section simply provides a way to slice the cube and control the data. However, the optional WITH section has considerable power to perform complex calculations. The results are still computed as sums and displayed in the form of a cube, but the computations can be used to answer some relatively complex business questions. A key aspect of MDX is that all of the elements rely on sets of data and almost all of the computations involve subtotals.

Basic Examples

How are MDX queries written and what basic data do they provide? The flexibility and power of MDX is easiest to see through examples. This section covers the basics—to highlight the syntax and the results. Two additional sections cover computed values and some more advanced tricks that can be useful in business analyses.

SQL Server has two main ways to run MDX queries: (1) Within a Visual Studio Analysis Project Cube Browser, and (2) Connecting SQL Server Management Studio to the Analysis database. Both of them use similar steps and produce similar results. The Management Studio supports cut-and-paste on some of the results, but the Visual Studio Cube Browser supports a Designer mode where basic cubes can be built by dragging and dropping dimensions. The examples in this chapter can be run with either method; but some examples found on the Web will run only in the Management Studio.

Creating and running MDX queries is a little tricky in the recent versions of Visual Studio. If necessary, open the project used to define the cube. Open the RT Sales cube. If necessary, process the cube so all dimensions, hierarchies, and subtotals are built. Click the cube's Browser tab. By default, the browser is in designer mode—where you can drag and drop dimensions and measures to create a cube. This process automatically generates MDX queries—which can provide useful examples. But for now, turn off the designer by deselecting the Design Mode icon, which switches the display to a window to type MDX queries and a window to display results.

```
SELECT
  { [Model Type].members } on rows,
  { [Measures].[Sale Price] } on columns
FROM [RT Sales]
```

Model Type	Sale Price
(null)	208438543
Hybrid	3399366.21
Mountain	25793699.63
Mountain full	61436230
Race	61700574.07
Road	46617603.73
Tour	9268120.59
Track	222948.77
Unknown	(null)

Figure 10.8

Initial MDX query. Total sales (Sale Price) by Model Type. The query needs to specify only the rows dimension (Model Type), and the column measure (Sale Price). MDX automatically totals across all other dimensions. The (null) row in the result shows the grand total. The Unknown row is automatically included in case some values are not specified. Changing [Model Type].members to [Model Type].children removes the (null) total.

To create MDX queries in SQL Server Management Studio, start the Management Studio from Windows and change the Server type to: Analysis Services. Choose the correct server name and login information. In the Object Explorer, expand the Databases entry to find the project you created and saved within Visual Studio. Right-click that entry (RT Sample) and choose the option for New Query/MDX.

A First Example

Begin with a query that uses the simplest syntax possible. The business question is to display total sales by model type. Figure 10.8 shows the query and the results. The FROM command is the simplest because it just lists the name of the data cube. Only one cube can be used in any query. The SELECT element specifies two sets—one for each axis of rows and columns. The braces indicating a set are required—even if only a single dimension is needed. The first set { [Model Type].members } is assigned to the rows. The .members notation tells the system to look up all entries for model type. The second set { [Measures].[Sale Price] } specifies the values to be displayed. Every query must have at least one measure in the SELECT statement. Note that the order of the sets in the SELECT statement does not matter (columns can be defined before rows). Check the results to see that all of the model types have been retrieved. Notice the *(null)* and *Unknown* entries. The *Unknown* value is automatically included to handle cases where the model type might not be given. The *(null)* row shows the grand total. Out of curiosity, change the .members to .children and rerun the results. The values will be the same, but the .children approach does not include the grand total row.

```
SELECT
  { [Model Type].members } on rows,
  { [Measures].[Sale Price] } on columns
FROM [RT Sales]
WHERE ( [Year].[Calendar 2010] )
```

Model Type	Sale Price
(null)	14501690
Hybrid	(null)
Mountain	1018340
Mountain full	4277740
Race	5171940
Road	3417090
Tour	616580
Track	(null)
Unknown	(null)

Figure 10.9

Adding a WHERE condition. Conditions added to WHERE limit the data to the specified values. The new table results are structurally the same as before but the values are computed only for 2010. Note the need to specify [Calendar 2010] because that is the way the values are entered in the Date Hierarchy dimension.

Adding a WHERE Condition

What if the analyst wants to limit the results? For example, the analyst wants to see the sales of model type but only for 2010. Note that the date is not currently included in the display—the existing values are computed for all possible years. Restricting the results to a single year can be thought of as looking at a **slice** of the cube. As shown in Figure 10.9, this basic condition is straightforward to add to the WHERE clause. The one catch is that the condition has to be written as [Calendar 2010] because that is the way the data was generated for the Date Hierarchy dimension. In many cases it helps to build a sample query using the designer first—just to see the exact specifications of the data for these generated dimensions.

Note that a WHERE condition specifies values that will limit the subtotal computations. To specify additional conditions, separate them by commas. The totals will then be computed where all of the specified conditions are true. For example, keep the year 2010 condition and add another restriction to the state of California (CA). Figure 10.10 shows the syntax and the results. Note that the values in the WHERE clause are separated by commas and data must match both conditions to be included in the result totals. More complex conditions require the use of functions; which are described in a later section.

Notice that Track and Hybrid bikes were only sold in some years—and not in 2010—so the results include missing (null) values. Because this effect occurs on only two rows it might not matter. But if a data result contains many empty rows, the analyst might want to remove them to focus on the values that do exist. MDX uses the **NON EMPTY** keyword to remove rows that are completely empty. Figure 10.11 shows the keyword and the result.

```
SELECT
  { [Model Type].members } on rows,
  { [Measures].[Sale Price] } on columns
FROM [RT Sales]
WHERE ( [Year].[Calendar 2010], [State].[CA] )
```

Model Type	Sale Price
(null)	895730
Hybrid	(null)
Mountain	73530
Mountain full	226640
Race	290470
Road	262160
Tour	42930
Track	(null)
Unknown	(null)

Figure 10.10

Multiple WHERE conditions. The WHERE clause specifies values that will be included. Separate them by commas and only data matching all conditions will be used for the totals. Here, the state of California for the year 2010.

Figure 10.11

Removing empty rows. The NON EMPTY keyword drops the rows from the results that are completely empty (null).

```
SELECT NON EMPTY
  { [Model Type].members } on rows,
  { [Measures].[Sale Price] } on columns
FROM [RT Sales]
WHERE ( [Year].[Calendar 2010], [State].[CA] )
```

Model Type	Sale Price
(null)	895730
Mountain	73530
Mountain full	226640
Race	290470
Road	262160
Tour	42930

```
SELECT NON EMPTY
  { [Model Type].[Mountain], [Model Type].[Mountain full] } on rows,
  { [Measures].[Sale Price] } on columns
FROM [RT Sales]
WHERE ( [Year].[Calendar 2010], [State].[CA] )
```

Model Type	Sale Price
Mountain	73530
Mountain full	226640

Figure 10.12

Selecting only some rows. Selection values are sets can contain lists and combinations of dimensions. Remove the .members value that retrieved all model types and specify just the two values for mountain and mountain full model types.

Displaying Specific Dimension Values

What if the analyst wants to focus on just the mountain and mountain full bikes? Because the Model Type dimension is used in the SELECT clause, it should not (or cannot) be used as a WHERE condition. The answer is to not select all members but just specify the desired entries on the SELECT statement. Remember that selection of dimensions for the axes use a set so many different values can be entered. Figure 10.12 shows the change in the MDX query that removes the .members keyword and replaces it with the two model types (mountain and mountain full). The computed values are the same as the earlier query but now display just the values for the two selected model types.

It is possible to add other types of dimensions to the same set. However, it might be more useful to put different dimensions on different axes. On the other hand, the browser with SSAS 2012 really only supports rows and columns. It has no interface to show different combinations of dimensions. The Microsoft PivotTable in Excel does a better job of handling multiple dimensions at the same time. So, a cube could be constructed and saved in SSAS and then browsed with a PivotTable. Some of these issues arise in the next section on cross joins.

Cross Join

The whole point of an OLAP cube is to provide the ability to explore data and browse through different dimensional subtotals and different hierarchical levels. MDX has some useful tools to create these types of cubes. Unfortunately, the cube browser shipped with SSAS 2012 is weak and does not support interactive rollup and drill down. Hopefully, the browser will be improved in future editions. In the meantime, it is still important to understand how MDX handles multiple dimensions.

The cross join is an important tool in building a cube. A **cross join** takes all the values of one dimension and combines them with every value from a second dimension. For example, say the state dimension has 50 states and the model type dimension has 7 entries. Crossing every state with every model type leads to 7 * 50 = 350 entries. Think of the results in terms of a matrix—using 7 columns against 50 rows. In fact, this is the way most cube browsers would display this cross join.

```
SELECT NON EMPTY
  { [Measures].[Sale Price] } ON COLUMNS,
NON EMPTY
  { ([Date Hierarchy].[Year -  Quarter -  Month -  Date].[Date].ALLMEMBERS
                          * [Model Type].[Model Type].[Model Type].ALLMEMBERS ) }
ON ROWS
FROM [RT Sales]
```

Note that the hyphens in the [Year – Quarter …] term are picky. You might have to copy and paste the notation from the designer.

Year	Quarter	Month	Date	Model Type	Sale Price
Calendar 1994	Quarter 1	January	01-JAN-1994	Race	2990
Calendar 1994	Quarter 1	January	01-JAN-1994	Road	3490
Calendar 1994	Quarter 1	January	02-JAN-1994	Race	2478.95
Calendar 1994	Quarter 1	January	03-JAN-1994	Mountain	5020
Calendar 1994	Quarter 1	January	03-JAN-1994	Race	10410
…					

Figure 10.13

Selecting only some rows. Selection values are sets can contain lists and combinations of dimensions. Remove the .members value that retrieved all model types and specify just the two values for mountain and mountain full model types.

To demonstrate the cross join in MDX, start with a new problem. The goal is to cross the Date Hierarchy with the Model Type dimension to examine Sale Price over time and model. Figure 10.13 shows the MDX command and a small portion of the results. The key to creating the cross join is the asterisk (*). MDX syntax includes a *cross join* key word but the asterisk is actually easier to read. The basic syntax is dimension1.members * dimension2.members. The SSAS 2012 browser display is not interactive so most people would transfer the cube to an Excel PivotTable for browsing instead. The Visual Studio editor does have a button to transfer the data to Excel but the cube has to be reconfigured. Of course, filters can still be used to reduce the number of rows—such as selecting individual states or the gender of customers.

Notice that the SSAS 2012 browser does not display the data as a cube. Instead, it generates a new row for each entry and then lists the corresponding dimension value. This approach makes it harder to see patterns, but easier to display multiple measure values. Also, the data can be passed cleanly to other analysis tools such as statistical packages.

Calculated Measures

How are computations and new measures defined? Many business problems require manipulating the measure values to compute new variables. Simple computations include profit margin (revenue – costs) and simple percentages such as Costs / Revenue. These computations are straightforward because they operate on data within the same cell.

```
WITH MEMBER [Measures].[Margin]
  AS '[Measures].[Sale Price] - [Measures].[Component List] - [Measures].[Frame Price]'
SELECT
  NON EMPTY { [Measures].[Margin], [Measures].[Sale Price] } On columns,
  NON EMPTY   { [Year].members * [Model Type].members } ON rows
FROM [RT Sales]
```

Year	Model Type	Margin	Sale Price
(null)	(null)	47211482.32	208438543
(null)	Hybrid	884842.47	3399366.21
…			
Calendar 1994	(null)	261651.05	2555013
Calendar 1994	Hybrid	14310.22	199006.21
…			

Figure 10.14

Calculated Measures. Computations are defined in the WITH MEMBER section which contains the name of a new measure (margin) and is definition delimited in single quotes. The new variable can then be used in the SELECT statement.

In MDX, calculated measures are defined in a separate section at the top of the query. A new variable name is assigned and the formula is delimited with single quotes. Figure 10.14 shows the formula for computing the simple profit margin for Rolling Thunder Bicycles (ignoring labor costs). The measure is defined in the WITH MEMBER section added to the top of the query. It must be given a unique name (Measures.Margin) and the calculation is delimited in single quotes. The new variable can then be used within a SELECT statement.

The syntax for defining a new measure is straightforward—just remember to place single quotes around the expression that defines the computation. Also, be sure that the name of the new variable is unique. Similar computations involving data from the same source at the same level are equally straightforward to create. However, MDX has several built-in functions that make it possible to answer more complex questions.

Complex Computations

How does MDX handle complex computations that cross levels or rows of data? The computations in the previous section were deliberately kept simple to illustrate the syntax for calculating new values. MDX has several powerful functions that can be used to answer more complex questions. Most of these types of questions involve the use of data beyond that found in a single row. For example, calculating percentages requires dividing the current value by a subtotal which must be computed across all the members within the same level. Similarly, many time-based problems involve looking at consecutive values to compute changes over time. Special functions are often needed to handle these jumps across different levels, but MDX has several of these capabilities.

```
WITH
  MEMBER Measures.PctSales AS '([Model Type].CurrentMember, [Sale Price] ) / ([Model
Type].CurrentMember.Parent, [Sale Price])' , FORMAT_STRING="0.00%"
SELECT
{ [Measures].[Sale Price], [Measures].[PctSales] } ON COLUMNS,
{ [Model Type].Members } ON ROWS
FROM [RT Sales]
```

Model Type	Sale Price	PctSales
(null)	208438543	Infinity
Hybrid	3399366.21	0.01631
Mountain	25793699.63	0.12375
Mountain full	61436230	0.29475
Race	61700574.07	0.29601
Road	46617603.73	0.22365
Tour	9268120.59	0.04446
Track	222948.77	0.00107

Figure 10.15

Computing percentages using Parent function. The computation relies on the fact that MDX always computes totals. The key term is: ([Model Type].CurrentMember. Parent, [Sale Price]) which computes the overall total to use as the divisor. Note that the Format_String does not work in Visual Studio 2012.

Percentages

A common business situation is the need to display subtotals along with their percentage of the overall total. Consider the sales for Rolling Thunder by model type. A listing of the model types followed by the total sales value is interesting. But, the numbers can be easier to understand if they are computed as percentages. For instance, rather than just listing: Race 2000, Mountain 4000, and so on; the display could list: Race 15%, Mountain 35%, and so on. These percentages would be even more useful to examine changes over time. The computation of the percentages in MDX is handled by defining a new measure and using the parent property.

Figure 10.15 shows the MDX command and the results with the percentages. The key trick is in the formula for the divisor. Look closely at the formula: ([Model Type].CurrentMember.Parent, [Sale Price]). This formula uses the Parent function to move up one level in the hierarchy; and one level up means to look at all model types. MDX then sums the [Sale Price] at that parent level—which provides the total sales for all model types.

What would happen if you combine the cross-join with percentages? That is, the business analyst wants to examine the total sales by model type by year; and wants to see the percentages computed within each year. The answer is that you simply need to add the cross join to the MDX row axis. Everything else stays the same. Figure 10.16 shows the MDX query and a portion of the results. Again, the results would be easier to browse if the model type were displayed across the top as columns as in an Excel PivotTable. But the main point is that MDX computed the percentages using the correct totals. The totals are correct because the Parent

```
WITH
  MEMBER Measures.PctSales AS '([Model Type].CurrentMember, [Sale Price] ) /
        ([Model Type].CurrentMember.Parent, [Sale Price])'
SELECT NON EMPTY
  { [Measures].[Sale Price], [Measures].[PctSales] } ON COLUMNS,
NON EMPTY
  { ([Year].AllMembers
          * [Model Type].[Model Type].[Model Type].ALLMEMBERS ) }  ON ROWS
FROM [RT Sales]
```

Model Type	Sale Price	PctSales
Hybrid	199006.21	0.07789
Mountain	559019.63	0.21879
Race	928984.07	0.36359
Road	486773.73	0.19052
Tour	346200.59	0.13550
Track	35028.77	0.01371
Hybrid	124240	0.04200
Mountain	567940	0.19199
Race	294390	0.99516
Road	1074490	0.36322
Tour	709230	0.23975
Track	187920	0.06352

Figure 10.16

Cross-join with percentages. Compute sales by model type by year and the corresponding percentages within each year. The only change is to add the cross-join. The parent function automatically computes the total across model types within the same year.

function simply moves up one level from the current item (model type within a given year) and then computes the total at that level. So, regardless of how the cross join is defined, the totals and percentages will be computed across the model types for the level.

Yes, it is possible to navigate further up the data tree using multiple parent commands (.parent.parent). Such a command might be used to compute a grand-grand total. But, be careful to ensure that the data tree has enough levels or trying to move to a non-existent parent will trigger an error.

Compute Changes

Another common analytical issue is the need to examine changes in data—particularly over time. As shown in the earlier examples, computing totals at any point in time is straightforward in MDX. Picture a table listing the Year and Sale Price for each year. The challenge now is to take the value for one row and subtract the value from the previous row. That means MDX needs a function to retrieve the value on the previous row. The function **PrevMember** does exactly that task. There is also a **NextMember** function that looks forward instead of back. For

```
WITH
  MEMBER [Measures].[Change] AS '[Measures].[Sale Price]
                                - ([Measures].[Sale Price], [Year].PrevMember)'
SELECT
{ [Measures].[Sale Price], [Measures].[Change] } ON COLUMNS,
{ [Date Hierarchy].[Year].Children } ON ROWS
FROM [RT Sales]
```

Year	Sale Price	Change
Calendar 1994	2555013	2555013
Calendar 1995	2958210	403197
Calendar 1996	4050860	1092650
Calendar 1997	5358120	1307260
…		

Figure 10.17

Computing change values with PrevMember. Change values require looking at values in prior (or next) rows. The PrevMember function looks back one item. NextMember looks forward by one, and Lag and Lead functions can jump multiple rows at one time.

more extreme situations, the **Lag** and **Lead** functions accept a number to go back or forward more than one item at a time. These functions overlap somewhat. For example, data on the previous row could be referenced by any of the commands: PrevMember, Lag(1), or Lead(-1).

Figure 10.17 shows the Change variable defined with the PrevMember function. The basic syntax is to use parentheses to specify the variable to look up (Sale Price) and then the PrevMember function of the year variable: ([Measures].[Sale Price], [Year].PrevMember). Check the change values in the partial results table. Notice that the first row does have an entry—it is equal to the first value of sales (for 1994). Specifying this value is always a question when computing changes. MDX assumes that the system starts at zero, so any data in the first year must be the full change amount from zero. Other systems might leave the first row empty, so it is worth remembering this approach in MDX.

ParallelPeriod Function

Many business questions require comparing numbers at different points in time. Comparing data to the prior period is common, but more complex comparisons are also common. For example, consider a listing of monthly sales using the time hierarchy. The hierarchy shows Year, Quarter, and Month. The challenge is that the business managers want to compare sales in a given month to the same month in the prior quarter. For example, sales in April (second quarter, first month) should be compared to sales in January (first quarter, first month). The **ParallelPeriod** function was designed to handle these types of question.

Figure 10.18 shows the basic objective. Looking at the month of April, the matching value is found by moving up the tree (parent) to the quarter; then moving back one quarter. From that point, the three months are matched in parallel to the months in the original quarter.

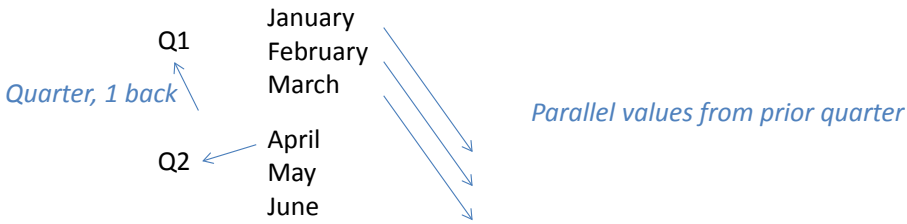


Figure 10.18

Comparing monthly values against the corresponding month in the prior quarter. The comparison uses the hierarchy to move up one level (to quarter from month) and then back one quarter. Then it tracks the months in parallel to the original data.

Figure 10.19 shows the MDX command to compare the corresponding months from a quarter to those in the prior quarter. The heart of the query is function: ParallelPeriod([Date Hierarchy].[Year – Quarter – Month – Date].[Quarter], 1). The first parameter tells the function to use the quarters (parent of the month level). The second parameter specifies to go back a single quarter. A third parameter can be added, but it defaults to the current member (month) being displayed and is easier to leave blank. Actually, for this specific query, all three parameters could be blank because the default values match the desired elements. Looking at the

Figure 10.19

ParallelPeriod function. The totals are monthly so the function uses the parent level (Quarter) to move back one period. It then automatically parallels or matches the corresponding months.

```
WITH
  MEMBER [Measures].[ParQtr] AS
    '(  [Measures].[Sale Price],
       ParallelPeriod ( [Date Hierarchy].[Year -  Quarter -  Month -  Date].[Quarter], 1    ) )'
 SELECT NON EMPTY
  { [Measures].[Sale Price], [Measures].[ParQtr] } ON COLUMNS,
NON EMPTY
  { ([Date Hierarchy].[Year -  Quarter -  Month -  Date].[Month].ALLMEMBERS ) }  ON
ROWS
FROM [RT Sales]
```

Year	Quarter	Month	Sale Price	ParQtr
Calendar 1994	Quarter 1	January	215836.97	(null)
Calendar 1994	Quarter 1	February	205952.48	(null)
Calendar 1994	Quarter 1	March	211128.77	(null)
Calendar 1994	Quarter 2	April	188159.83	215836.97
Calendar 1994	Quarter 2	May	225387.63	205952.48
Calendar 1994	Quarter 2	June	210278.66	211128.77
…				

partial results, it is clear that the function has picked up the matching months from the prior quarter—except for the first quarter of null values because nothing exists before that time.

In a similar manner, companies often want to compare monthly sales to the value in the prior year. The ParallelPeriod function in this example could be modified to handle a full year simply by changing the second parameter from 1 to 4, because 4 quarters make up a year. This function simplifies many common business comparisons over time. Note that the function itself returns the time value not the actual data. Hence the new member measure is defined using the tuple notation with parentheses: (data item, member/level) where the member value is obtained using the ParallelPeriod function.

Some MDX Functions

What other MDX functions are commonly used in business problems? MDX has many functions and they can be used to solve tricky problems. For example, check the Web for examples of problems such as computing year-to-date totals for current and prior years. The purpose of this section is to briefly describe some of the commonly-used functions in MDX. Only a partial list is covered here—the online reference documents for MDX provide complete lists and more examples. Figure 10.20 lists some of the main MDX functions by category. Detailed lists and descriptions of the functions can be found online in the reference documents. Some of the functions are straightforward; others are complex and require detailed explanations and examples. Some of the functions have already been mentioned (Lag, Lead, ParallelPeriod, Parent, PrevMember). A couple of additional functions are described in this section, but a comprehensive discussion would require an entire textbook just to cover the functions.

EXCEPT: Taking Values Out of Totals

Recall the WHERE conditions given in the initial examples. Expressions entered in the WHERE clause act as a cube slicer or filter and limit the totals to just that data. The example computed total sales by Model Type for the state of California (CA). The syntax was straight forward:

```
SELECT NON EMPTY
  { [Model Type].members } on rows,
  { [Measures].[Sale Price] } on columns
FROM [RT Sales]
WHERE ([State].[CA])
```

But, what if the business manager wanted the opposite information: Sales in all states except California? So far, all of the functions and tools of MDX have focused on selecting data to be included and displayed. The solution is to use the **EXCEPT** function, which returns all of the values from a set minus the ones specified. The EXCEPT function takes two parameters: (1) The full set of items, and (2) The item to be excluded.

Figure 10.21 shows the MDX query to display the sales by model type for all states except the state of California. The results show the query executed three times with different WHERE conditions. The data can be tested in Excel to verify that the EXCEPT column contains the total for all states except California. The EXCEPT function can also be used within the SELECT clause to choose which model types should be displayed. In most cases, it will be easier to read the query

Meta	Axis Count Hierarchy	Level Name Ordinal	
Navigation	Ancestor Ascendants Children Cousin Current CurrentMember	DataMember FirstChild FirstSibling Lag LastChild LastSibling	Lead LinkMember NextMember Parent PrevMember Siblings
Logical	IIF IsAncestor IsEmpty	IsGeneration IsLeaf IsSibling	
Sets	AllMembers BottomCount BottomPercent BottomSum Crossjoin Descendants Distinct Except	Exists Extract Filter Generate Head Hierarchize Intersect Members	NonEmpty Order Subset Tail TopCount TopPercent TopSum Union
Statistical	Avg Correlation Count Covariance DistinctCount	Max Median Min Rank RollupChildren	Stdev Sum Var VisualTotals LinReg…
Time	ClosingPeriod LastPeriods OpeningPeriod ParallelPeriod	Mtd Qtd Wtd Ytd	PeriodsToDate

Figure 10.20

Some MDX functions. Some are more useful than others and the full list and details can be found in the online MDX reference documents.

if the desired items are entered. However, for long lists, the EXCEPT function could make it easier to create a query. The syntax remains the same; simply enter the EXCEPT function inside a SELECT brace and enter the full list and the exception item.

Conditions with IIF and CoalesceEmpty

Much like SQL, MDX is designed to operate on sets of data and it is not designed to function as a sequential programming language. Conditions are a key aspect of sequential languages (IF … THEN … ELSE), but are less important in queries. Instead, the SELECT and WHERE statements control which items are included or excluded. Nonetheless, sometimes queries require different treatment for certain situations. A common example is to handle problems arising from division by zero. A calculation might be performed differently, or skipped, if a divisor is zero. MDX, as with many other tools such as Excel, handle conditions with the **IIF** function. IIF stands for "immediate if," and consists of three parameters:

```
IIF ( condition, true, false)
```

```
SELECT NON EMPTY
  { [Model Type].members } on rows,
  { [Measures].[Sale Price] } on columns
FROM [RT Sales]
WHERE ( EXCEPT ( [State].Children, [State].[CA]) )
```

Model Type	All	CA	EXCEPT CA
All	208438543	16274614.3	192163928.7
Hybrid	3399366.21	219987.38	3179378.83
Mountain	25793699.63	1742601.88	24051097.75
Mountain full	61436230	4362440	57073790
Race	61700574.07	5879280.04	55821294.03
Road	46617603.73	3380330	43237273.73
Tour	9268120.59	654335	8613785.59
Track	222948.77	35640	187308.77

Figure 10.21

Except function. The query was run with three different WHERE conditions to verify the results. The EXCEPT function requires two parameters: (1) The full list, (2) The item to be excluded from the list.

Most of the data in the Rolling Thunder Bicycles case is relatively clean, with few chances of dividing by zero. However, several examples do generate missing or null data which can also be used to illustrate the IIF function. Figure 10.22 shows an example of the IIF function. Full suspension mountain bikes were not introduced by the company until 1997, so model type computations before that year contain missing (null) values. The IIF function is used here to check for missing (IsEmpty) values before dividing by sale price. If the value is missing, the function returns a zero, otherwise it performs the division. Technically, the IIF function is not needed for null data because any computation with null value always results in a null value so the system does not bother to attempt the actual computation. But the format and use of the function is the same if testing for zero values—simply replace the IsEmpty function with a test to see if the item equals zero (a = 0).

The IIF function can also be used to recode data—but it only codes two items at a time. For instance, the condition could test to see if total sales are above some limit and then return an positive indicator, otherwise, it returns a zero or some other value.

Dealing with null values is common enough that MDX has a separate function just to handle this situation: CoalesceEmpty. Essentially it is a simplified test to see if a value is empty. Remember that any computation with null values creates a null result. Consider a calculation that adds two numbers: [Measures].[Frame Price] + [Measures].[Component List]. What happens if a customer purchases a frame but no components? Then [Component List] will be empty and the total will also be defined as null, and essentially discarded from the analysis. To treat

```
WITH
  MEMBER [Measures].[ComponentPct] AS
 'IIF(IsEmpty([Measures].[Sale Price]),0, [Measures].[Component List]/[Measures].[Sale
Price])'
SELECT
  { [Year].members * [Model Type].members } on rows,
  { [Measures].[Sale Price], [Measures].[ComponentPct] } on columns
FROM [RT Sales]
```

Year	Model Type	Sale Price	ComponentPct
Calendar 1994	All	2555013	0.68167
Calendar 1994	Hybrid	199006.21	0.68211
Calendar 1994	Mountain	559019.63	0.64696
Calendar 1994	Mountain full	(null)	0
Calendar 1994	Race	928984.07	0.69753
Calendar 1994	Road	486773.73	0.70422
Calendar 1994	Tour	346200.59	0.64459
Calendar 1994	Track	35028.77	0.86555
Calendar 1994	Unknown	(null)	0

Figure 10.22

IIF function. Commonly used to handle division by zero or missing data, the IIF function takes three parameters: (1) logical condition, (2) value if condition is true, and (3) value if condition is false.

the two items as optional, they can be coalesced to a zero value before trying to add them:

```
CoalesceEmpty([Measures].[Frame Price], 0)
+ CoalesceEmpty( [Measures].[Component List], 0)
```

Now if either value is missing, it will first be converted to zero before attempting to add the values together. As a result, the calculation will not generate null values even if one of the two items is missing.

TopCount Function

With SQL, Microsoft SQL Server introduced the TOP n option which is often used to cut off a display list to show just the top number of items. This same concept shows up in the MDX **TopCount** (and **BottomCount**) function. The purpose of the TopCount function is to compute the desired totals, sort the totals, and return just the top n rows. The function requires three parameters:

```
TopCount ( row list members, n, measure to total)
```

The Bottom count function works the same way but counts up from the bottom of the list.

Figure 10.23 shows a sample use of the TopCount function. It computes the total sales by state and returns to five highest values.

Initially, the **TopSum** and **TopPercent** functions appear to be similar to the TopCount function; however, the behavior is different. The TopCount function

```
SELECT
  { [Measures].[Sale Price] } ON Columns,
  TOPCOUNT ( [State].Children, 5, [Measures].[Sale Price]) ON ROWS
FROM [RT Sales]
```

State	Sale Price
CA	16274614.3
NY	12437726.13
TX	11914093.18
IL	11408168.16
PA	11294417.46

Figure 10.23

TopCount function. The function sorts the data and returns the top rows from the list. The function needs to know the dimension for the rows, the number of rows to return, and the measure to be summed.

computes the totals and then cuts off the display list by counting the number of rows. The TopSum function has a similar syntax:

```
TopSum( [State].Children, 50000000, [Measures].[Sale
Price] )
```

But the control number (50,000,000) represents a total sum. The query starts at the top of the list and returns rows until the sum of the values exceeds the specified control value. So the sample query will list the top states needed to accumulate a total sales level of at least 50 million. The TopPercent function behaves similarly to the TopSum function. It lists states from the top down until the sales total from those states exceeds the specified percentage of the aggregate sum. So the business query would be of the form: List the states that generated 20 percent of the total sales. The BottomX functions operate the same but work from the bottom up instead of top down.

The TopCount function also introduces some issues in how sets are organized. What happens when a new list is created from Year cross-joined to the States? Should the top 5 count apply within each year or across the states? Figure 10.24 shows two ways to write the function: (1) Year cross-joined to TopCount, and (2) TopCount of Year cross-joined to State. Of course, another possibility would be to cross join State to TopCount by Year. The point of the examples is to highlight the importance of understanding when to apply a function versus when to perform a cross-join. The choice depends on the specific business question to be answered; but it is critical to match the method to the question—and then test the query by carefully examining the results.

Year to Date

Business managers are often interested in the progress of sales throughout the year. In particular, they want to see year-to-date totals, so they can get a feel for the total sales for the year. Year-to-date totals are often displayed next to monthly (or quarterly) values so the display essentially provides the monthly values and a

```
SELECT
  { [Measures].[Sale Price] } ON Columns,
  { [Year].Children *
  TopCount ( [State].Children, 5, [Measures].[Sale Price]) } ON ROWS
FROM [RT Sales]
```

Year	State	Sale Price
1994	CA	257734.3
1994	NY	188326.13
1994	TX	151563.18
1994	IL	158728.16
1994	PA	163677.46
1995	CA	286920
1995	NY	186600

```
SELECT
  { [Measures].[Sale Price] } ON Columns,
  { TopCount( [Year].Children * [State].Children, 5, [Measures].[Sale Price]) } ON ROWS
FROM [RT Sales]
```

Year	State	Sale Price
2014	CA	1160690
2012	CA	1157050
2007	CA	1150560
2014	TX	1142330

Figure 10.24

TopCount function in a cross-join. The first version cross joins year to the top count, which results in the top 5 states for each year. The second version performs the cross join first and finds the top 5 overall totals.

running total. Because of the popularity of these totals, MDX has a specific **YTD** function.

Technically, the YTD function returns a set of time periods within a hierarchy from the start of the year to the current time. So, time has to be in a hierarchy that specifies a calendar year and the MDX query has to use the Sum or Aggregate function to total the desired measure. Figure 10.25 shows the simplest version of the YTD function. The function automatically evaluates the time dimension hierarchy and finds the start of the year for the current time variable—month in this case. Hence, the function can be called with no parameters. The Aggregate (or Sum) function is used to add up the values for the Sale Price measure over the time periods returned from the YTD function.

Of course, business managers are never happy with the data provided to them. So, the next question asked is going to be: Can MDX compare year-to-date values for the current year with the values from the previous year? The solution is to

```
WITH
  MEMBER [Measures].[MonthYTD] AS
   'Aggregate ( YTD(), [Measures].[Sale Price] ) '
SELECT
  { [Measures].[Sale Price], [Measures].[MonthYTD] } on 0,
  { [Month].Children } on 1
FROM [RT Sales]
```

Month	Sale Price	MonthYTD
Jan-94	215836.97	215836.97
Feb-94	205952.48	421789.45
Mar-94	211128.77	632918.22
Apr-94	188159.83	821078.05
May-94	225387.63	1046465.68
Jun-94	210278.66	1256744.34
Jul-94	244788.67	1501533.01
Aug-94	197913.77	1699446.78
Sep-94	200731.22	1900178
Oct-94	219020.42	2119198.42
Nov-94	216574.56	2335772.98
Dec-94	219240.02	2555013
Jan-95	240050	240050

Figure 10.25

YTD function. The YTD function knows about time hierarchies and calendar years automatically so it can be called with no parameters to use the current time variable (month).

combine the YTD function with the ParallelPeriod function. Define a new measure (PriorYTD) as:

```
Aggregate (
   YTD (    ParallelPeriod ( [Date Hierarchy].[Year -
Quarter -  Month -  Date].[Year], 1  )      )
   ,[Measures].[Sale Price] )
```

Moving Averages

One more common business problem involves moving averages—which is a statistical concept that applies to time series data. Moving averages are commonly used in finance—they are useful for smoothing out short-term variations in prices. But they can be used on any time series data, including sales. A moving average is exactly what it says: an average that moves over time. The average moves because it is defined for a set number of periods. For instance, an MA(3) refers to an average of three data points and MA(12) averages data across 12 consecutive periods. Consider sales by month for Rolling Thunder Bicycles. An MA(3) would begin by averaging the values for the first three months: 1994-01, 1994-02, and 1994-03. This value would be entered for 1994-03. For the next month (1994-04), a new average of three values would be computed from 1994-02, 1994-03, and 1994-04.

```
WITH
  MEMBER [Measures].[MA03] AS
    'Avg([Month].CurrentMember.Lag(2):[Month], [Measures].[Sale Price] )'
  MEMBER [Measures].[MA12] AS
    'Avg([Month].CurrentMember.Lag(11):[Month], [Measures].[Sale Price] )'
SELECT
  { [Measures].[Sale Price], [Measures].[MA03], [Measures].[MA12] } ON 0,
  { [Month].Children } ON 1
FROM [RT Sales]
```

Month	Sale Price	MA03	MA12
1994-01	215837	215837	215837
1994-02	205952.5	210894.7	210894.7
1994-03	211128.8	210972.7	210972.7
1994-04	188159.8	201747	205269.5
1994-05	225387.6	208225.4	209293.1
1994-06	210278.7	207942	209457.4
1994-07	244788.7	226818.3	214504.7

Figure 10.26

Moving average examples. The Avg function does the work but the Lag function and : operator set the data range.

Because the MA(3) average always applies to three consecutive months, the calculation continually moves forward one month at a time.

With moving averages, a question arises over how to handle the first data points. For MA(3) over months, what should be done with the first two months? At that point, not enough data exists to compute an average for 3 months. Should the first two be skipped (null), or should the average simply use the data available? One point for the first month, and two points for the second month. With MDX, this latter option is the easiest to create because the Avg function automatically uses the number of data points available. It is possible to create null values for the first months if desired—by using the IIF function.

The Visual Studio Designer has a template under the calculations tab that generates the syntax for creating a moving average. However, the syntax is easier to understand using an actual example. Figure 10.26 shows two moving average measures for monthly sales—one for 3 months and one for 12 months. The **Avg** function does the actual work of computing the average. The trick is to get the correct data range so that it moves with the month being displayed. The solution is to use the Lag function to go back two months and then use the range operator (**:**) to specify all of the months from that lagged value to and including the current month. The 12-month moving average is included to show that the Lag function goes back 11 months because 11 plus the current month equals the 12 months needed.

The question of how to handle the first months—before reaching the number requested—is automatically solved by the Avg and Lag functions. The Lag function returns null values when trying to reach earlier than the starting month; and the Avg function ignores null values. So the MA(3) for the first month includes

only one value and the second month includes two but the averages are computed based only on the count available.

Several other useful functions exist in MDX, including year-to-date computations. They generally follow principles similar to the functions covered in this section. A few more exotic functions exist, including Generate, and Sets. Documentation and examples of these functions can be found on the Web. A key concept for many of them is that they create and modify sets of data. A powerful aspect of sets is that a cube itself is a set, and many functions operate on sets. Hence, it is possible to use functions to create a cube, essentially save (name) it as a set, and then apply MDX functions on those results. This iterative process is similar to saving a query and then writing another query using those results. Some problems are easier to solve by creating intermediate steps and applying calculations to those initial results.

Summary

Multidimensional expression (MDX) queries are a powerful way to retrieve data from OLAP cubes. MDX commands rely on four basic clauses: WITH to define calculations, SELECT to specify cube axes and dimensions, FROM to indicate the cube data source, and WHERE to set filters for the data. A key element of MDX queries is that sums are computed automatically—virtually all data returned consists of subtotals across selected dimensions. Dimensions, particularly hierarchies of dimensions, play an important role in MDX and in the terminology.

The SELECT statement contains sets of data for the various axes of a cube. Typically a cube consists of at least two axes: Columns and Rows, which can also be numbered 0 and 1. The Columns axis generally must contain values from the Measures set—specifically numbers that can be added to compute totals and subtotals. Sets in the SELECT statement are generally enclosed within curly braces. The cross join (*) is an important function in MDX because it combines all elements from one set of data with all elements from a second set. For example, a cross join would be used to create a matrix or table of sales for each month against model type in the Rolling Thunder Bicycles case.

New measures or columns of data can be calculated in a WITH statement by assigning a new name to the variable and writing a calculation. Simple computations use basic arithmetic on data at the same level. More complex calculations can reach across levels or dimensions. For example, percentages can be computed by dividing by the total value derived from the parent of the current member. Changes from one period to the next can be computed using the PrevMember, NextMember, Lag, or Lead functions. The ParallelPeriod function is a powerful way of comparing data to similar levels at different points in the hierarchy tree. For example, it is useful for comparing monthly sales to sales in the similar month in the previous quarter.

MDX includes several functions to handle tree navigation, logical conditions, sets of data, statistical computations, and operations with respect to time. Additional tools, such as string and value functions also exist. Some of the more common functions include Except, IIF, CoalesceEmpty, TopCount, and Avg. The main syntax of MDX is designed to return values that match specified dimensions, so the Except function is a negation method that returns data that matches items in a list except the ones specified. The immediate if (IIF) function provides inline conditional tests—returning different values if the condition is true instead of false. The CoalesceEmpty function provides a convenient way to handle missing data

by converting null values to almost any other value. Several "Top" and "Bottom" functions exist to return a smaller set of results. TopCount returns rows up to the count specified, while TopSum and TopPercent are used to limit results based on a running total. They are used to address business questions of the form: How many states does it take to reach 50 million in sales? Moving averages are quickly computed using the Avg and Lag function. The Lag function and range operator (:) are used to specify the data points relative to the current location, and the Avg function computes the average of those values.

Many other functions exist to solve complex business questions. Because MDX is designed to work with sets, and OLAP cubes are also sets, difficult problems can be solved by creating intermediate cubes and writing new queries against those named sets. Several Web sites provide detailed descriptions and examples of MDX functions and capabilities.

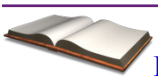
Key Words

* (cross join operator)
: (range operator)
Avg
axes
BottomCount
Children
columns
cross join
cube
dimensions
EXCEPT
FROM
hierarchical
IIF
Lag
Lead
Leaves
level
measure
Member
Multi-dimension expression or MDX
NextMember
NON EMPTY
ParallelPeriod
Parent
PrevMember
root
rows
SELECT
set
slice
TopCount
TopPercent
TopSum
tuple
WHERE
WITH MEMBER
YTD

Review Questions

1. What is a dimension hierarchy?
2. How are OLAP cube measures different from dimensions?
3. Compared to SQL, how is MDX better for dealing with OLAP cubes?
4. What is the main structure of an MDX query?
5. What would be the basic MDX query structure to list total sales by state for half of the states?
6. What is the difference between using several dimension axes and a cross join?
7. How do calculations involving percentages rely on the automatic subtotals in MDX?
8. Briefly explain how the following functions work and give an example situation: Parent, PrevMember, Lag, Lead.
9. How is the ParallelPeriod function useful for common business accounting reports that compare data to earlier time periods?
10. How does MDX handle reverse condition where an element is to be excluded from a total?
11. Which MDX function would be best used for each of the following problems?
    a. How many production days are required in each month to reach 50 percent of the total output?
    b. Classify customers as “whales” or “minnows” based on whether their total purchases for a month exceed some fixed value.
    c. Compute the percentage change in sales from year-to-year.
    d. List all of the months in a year within a dimension hierarchy.
    e. Treat missing values as 1 instead of null.
    f. When listing sales by month keep a running total of year-to-date sums.
    g. Compute a moving average.
    h. List the ten states with the lowest sales for each year.

Exercises



Book

1. Create and save the OLAP cube for Rolling Thunder Bicycles as explained in the chapter.

2. Use the MDX references to find two functions not covered in the chapter. Explain the purpose of each function, and create a sample query that uses the function.

3. Run the following query in both SQL Server Management Studio and the Visual Studio Cube Browser. Explain any differences and rewrite the query so that it runs the same in both tools. SELECT ([Measures].[Sale Price], [State].[CA]) ON 0 FROM [RT Sales].

4. Modify the query in Figure 10.17 to compute the percentage change from one year to the next.

5. Modify the query in Figure 10.19 to obtain the values from a year ago for each month.

6. Modify the query in Figure 10.21 to list the sales by model type by state but exclude models Hybrid and Track, and exclude Alaska and Hawaii (AK, HI).

7. Create a query that shows the ten states with the lowest sales in 2010.

8. Modify the query in Figure 10.24 do perform the cross join as: State * TopCount(Year) and explain what the results mean in the context of the other two versions from that figure.

Rolling Thunder Database

9. Create a new table (Region) that assigns states to regions (Northeast, Midwest, Northwest, Southeast, South, Southwest, and West). Create a new dimension hierarchy based on customer location using region, state, and city. Write the MDX query to examine sales by location over time.

10. Write the MDX query to compute the percentage change in sales over time (year) for each employee/sales person.

11. Create three-month moving average columns for the model types but exclude Hybrid and Track because of limited sales of those models.

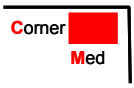
12. Using the data from the previous exercise, transfer the results to an Excel PivotTable (copy/paste), reformat the cube with months as rows and the model types as columns. Plot the moving averages.

13. Using quarterly totals instead of monthly values, create the MDX query to compute quarterly, YTD, and prior YTD totals for Sale Price. Copy at least a couple of years' worth of data to Excel and test the computations.

Diner

14. By week and gender, compute the average bill per person (diner).

15. Similar to the previous exercise, but edit the data source view and add a named calculation to the table: BillTotal/Number and add that new column to the cube measures (drag it in the designer). First, write a similar query in SQL and find the average values for week 1. (Hint: Use DatePart(wk, DineDate)). Now, write the MDX query that generates the same averages as the SQL query. Explain why these values are different from those in the previous exercise.



Corner Med

16. Create an OLAP cube for the financial aspects of Corner Med—using the Visit and Visit Procedures tables. Create the MDX statement to compute the monthly totals paid by the insurance company and the patient separately. Then include the year-to-date total that combines both values.

17. Because two Measures tables exist, the Time hierarchy has to be connected to both separately. The link from Time to Visit is handled as a "regular" relationship as usual. The link from Time to Visit Procedures is handled as a many-to-many relationship through the Visit table. Create this second relationship and create the MDX query to show the Procedure Amount charged per month.

18. Create the MDX query to list the number of procedures performed each month by each category of employee.

19. Create a three-month moving average of the total amount charged for procedures.

20. Compare the number of patient visits in a month for patients who use tobacco and those who do not. List the count of each type by month and show the corresponding percentage of the number of visits.

Basketball

21. Create a new OLAP cube based on the view for Team Game Totals. You might have to build a named query using the saved view in order to set the primary key correctly. Create a time dimension hierarchy that at least includes year and week; use a fiscal year for the season instead of a calendar year. Choose a team and examine the total points per game scored by week.

22. Choose a team and examine items that might be different for wins and losses [Won Loss]. Consider at least points per game, three points per game, and total rebounds [TRB].

23. List the teams by conference and compute the total points scored by each team per year. Within each year, compute the percentage of points scored by each team for its conference. That is, find the highest-scoring teams per conference. Use the Order function to sort the rows.

Bakery

24. Build an OLAP cube for sales that includes at least week and product category as dimensions. Note that because the SaleDate includes time, it will not join with a standard Time table. Hence, it is necessary to create a new named calculation SaleDateAlone: Convert(date, SaleDate). Also, remember to create SaleAmount=SalePrice*Quantity in the Sale Item table. Use MDX to create a four-week moving average for total sales.

25. Use MDX to create a cube that shows total sales quantity (not value) by year and category. Show the percentage of sales by category within each year.

Cars

26. Create a new OLAP cube with just the Cars table (not sales). Use MDX to create a cube that shows the average highway MPG by manufacturer (Make).

27. Use MDX to create a cube that lists the Drive type within each car Category and shows the average weight and average highway MPG. As a separate MDX query, compute the correlation coefficient between weight and highway MPG across all cars [Car ID].

Teamwork

28. Each team member should choose an MDX function not covered in this chapter and trade it with a team member. Each person then uses the received function in an MDX query for one of the databases.

29. Each team member should choose one database and write a business question to be answered with an MDX query. Trade problems with another team member and solve the received problem.

30. Each person should research a different software tool that supports MDX. Briefly summarize any major differences or features in the implementation. Combine the comments and draw a table comparing the tools.

Additional Reading

IBM, *MDX Language Reference*, http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=%2Fcom.ibm.dwe.cubemdx.doc%2Fmdx_concepts.html. [IBM's reference documentation for MDX.]

Carl Nolan, 1999, "Manipulate and Query OLAP Data Using ADOMD and Multidimensional Expressions," *Microsoft Systems Journal*, August. http://www.microsoft.com/msj/0899/mdx/mdx.aspx. [Introduction to MDX with simple examples and some discussion.]

Microsoft, *Multidimensional Expressions (MDX) Reference*, http://msdn.microsoft.com/en-us/library/ms145506.aspx. [Microsoft's main reference site for MDX, best for function reference.]

Microsoft, *MDX (SQL Server 2000)*, http://msdn.microsoft.com/en-us/library/aa216767(v=sql.80). [Microsoft overview and examples for MDX.]

SAS, *SAS 9.2 OLAP Server MDX Guide*, http://support.sas.com/documentation/cdl/en/mdxag/59575/HTML/default/viewer.htm#titlepage.htm. [MDX introduction and examples from SAS.]

Russ Whitney, 2001, "MDX by Example," *SQL Magazine*, http://www.sqlmag.com/article/quering/mdx-by-example. [Interesting business problems with MDX samples. Some sample code might not work with current versions.]