Chapter 3

Understanding MDX

What's in this chapter?

- What is MDX?
- MDX concepts
- MDX queries
- MDX expressions
- MDX operators
- MDX functions
- MDX scripts
- Restricting cube space/slicing cube data
- Parameters and MDX queries
- MDX comments

In Chapter 2 you ran a simple MDX query to retrieve data from Analysis Services. Building on that, in this chapter you learn the fundamentals underlying the MDX language to manipulate and query Analysis Services objects. You also learn about some advanced concepts, such as MDX operators and functions. This forms the basis for many of the subsequent chapters in this book.

SQL Server 2012 provides a sample Analysis Services project that demonstrates the majority of the Analysis Services features available from www.wrox.com to learn MDX. In addition, this chapter along with Appendix A provides you with examples of MDX queries based on the sample Adventure Works DW Multidimensional project updated for Analysis Services 2012. Some of the MDX queries you need to write to solve business problems necessitate the use of cube space restriction, empty cell removal, and parameterized queries — all concepts covered in this chapter.

What Is MDX?

Just as Structured Query Language (SQL) is a query language used to retrieve data from relational databases, MultiDimensional eXpressions (MDX) is a query language used to retrieve data from Analysis Services databases. MDX supports two distinct modes:

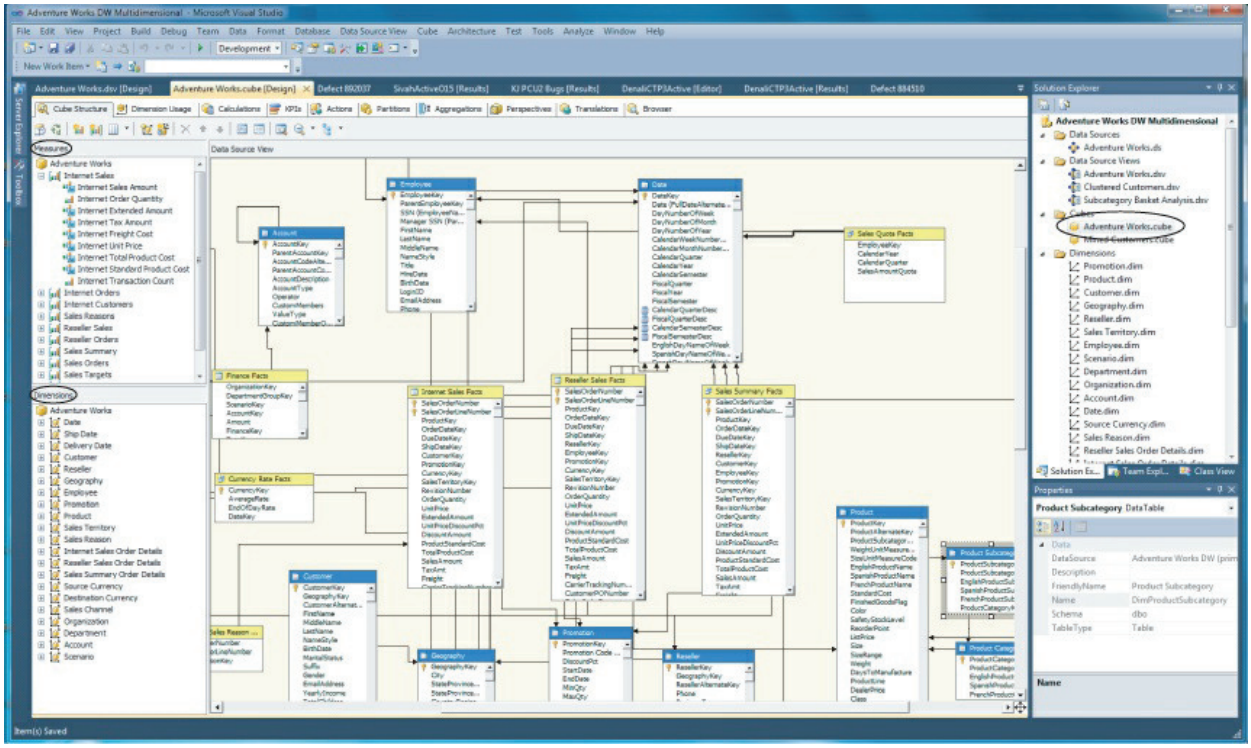
- **Expressions language:** Define and manipulate Analysis Services objects and data to calculate values.
- **Query language:** Retrieve data from Analysis Services.

MDX was originally designed by Microsoft and introduced in 1998 with SQL Server Analysis Services 7.0, but it is nevertheless a general, standards-based query language to retrieve data from OLAP databases. Many other OLAP providers support MDX, including Microstrategy's Intelligence Server, Hyperion's Essbase Server, and SAS's Enterprise BI Server. There are those who want to extend the standard for additional functionality, and MDX extensions have indeed been developed by individual vendors, but the constituent parts of any extension are expected to be consistent with the MDX standard. Analysis Services provides several extensions to the MDX standard defined by the OLE DB for OLAP specification. In this book you learn about the form of MDX supported by SQL Server Analysis Services 2012.

MDX Concepts

A multidimensional database in SQL Server Analysis Services contains one or more cubes. A *cube* is the object that is designed and used by the users for data analysis. Each cube must contain at least two dimensions (one of which is a special dimension called Measures), but typically will contain more than two dimensions. For example, the callouts in Figure 3.1 show the dimensions that are part of the Adventure Works cube.

Figure 3.1



You can download and install the AdventureWorksDW relational database and the SSAS Multidimensional model Projects of SQL Server 2012 from this book's page on www.wrox.com. Open the Adventure Works DW Multidimensional project in SQL Server Data Tools (SSDT) to see the measures and dimensions that make up the cube on the Cube Structure tab.

Measures and Measure Groups

The *Measures* object within a cube is a special cube dimension representing a collection of measures. Measures are quantitative entities that are used for analysis. You can see some measures in the sample project in Figure 3.1.

Each measure is part of an entity called a measure group. *Measure groups* are collections of related measures, and each measure can only be part of a single measure group. Measure groups are primarily used for navigational purposes for better readability or ease of use in client tools. Measure groups are never used in MDX queries when querying measures. However, they can be used in certain MDX functions discussed later in this chapter.

Hierarchies and Hierarchy Levels

You can see that the dimensions in the Adventure Works cube have one or more hierarchies, and each hierarchy contains one or more levels. Figure 3.2 shows a simplified cube with three hierarchies — Calendar, Product Line, and Country — analyzing the data for illustration in this chapter. In the Adventure Works DW sample, the Calendar hierarchy of the Date dimension contains five levels: Calendar Year, Calendar Semester, Calendar Quarter, Month, and Date, whereas the Product Line and Country hierarchies are attribute hierarchies with only two levels: the All level and the Product Line or Country level, respectively. You learn more about dimensions, hierarchies, and levels in Chapters 5 and 8.

Figure 3.2

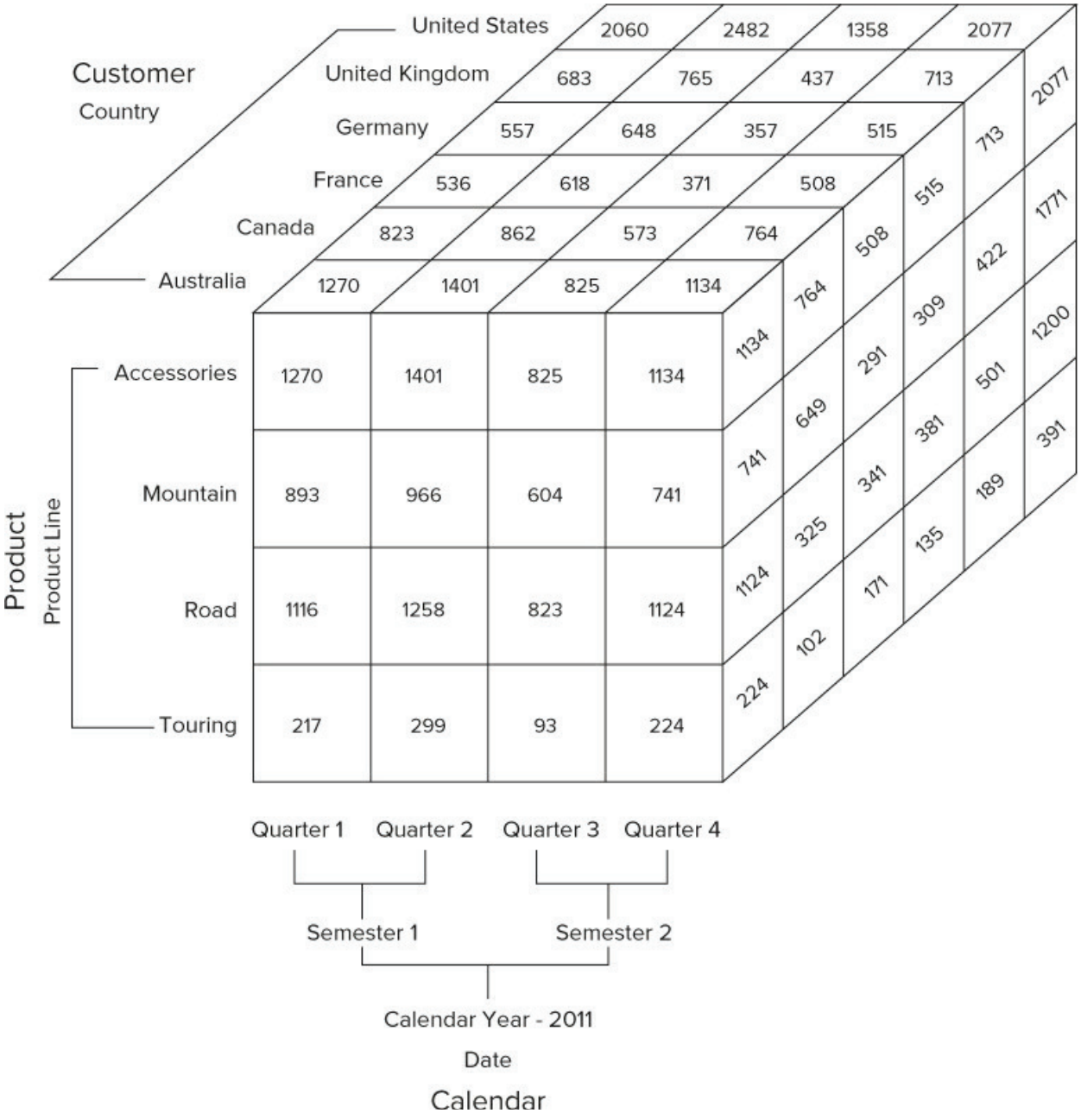


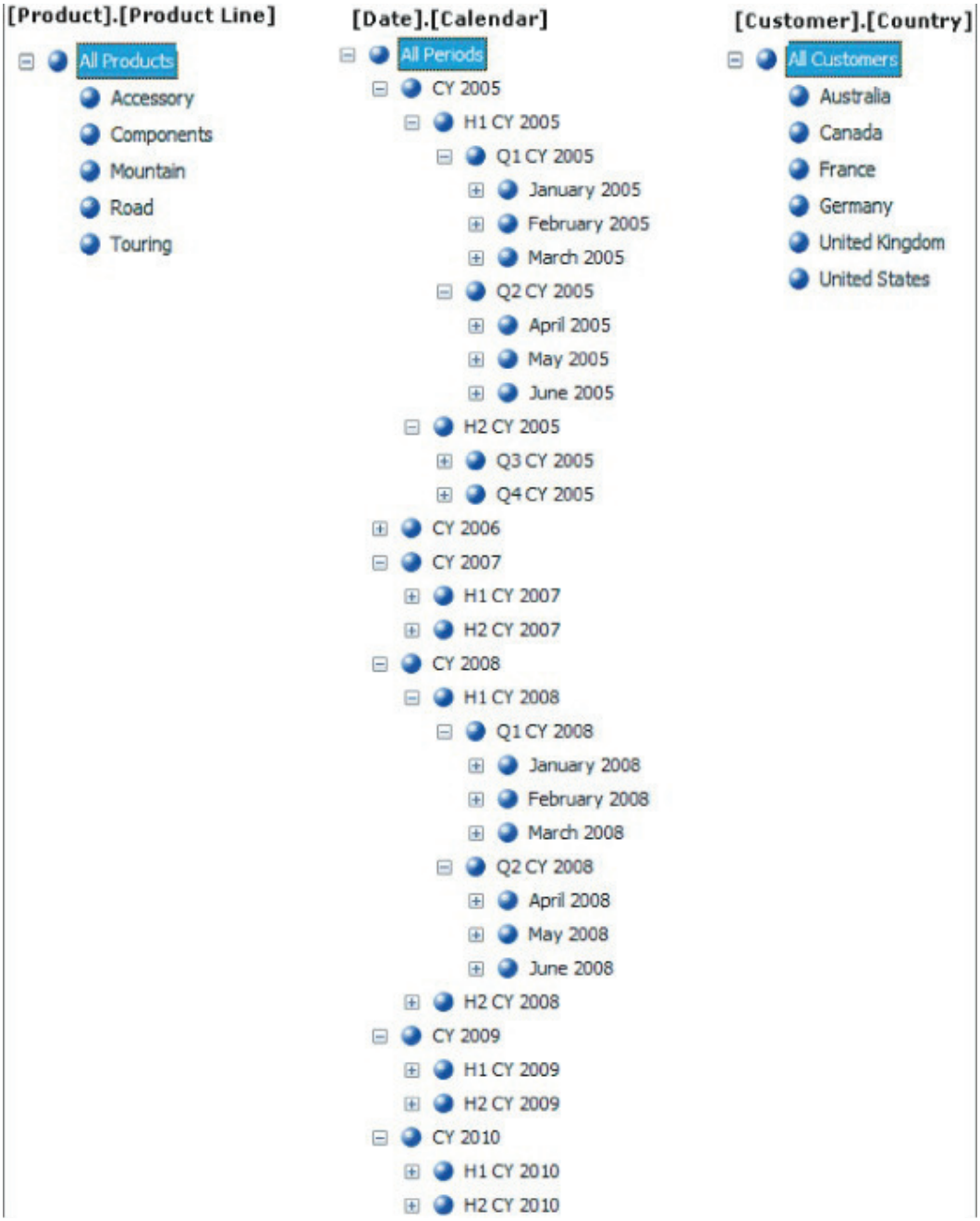
Figure 3.2 does not reflect the actual data in the sample cube. For illustration purposes, some members from the hierarchies are removed.

Members

Each hierarchy contains one or more items that are referred to as *members*, and each member corresponds to one or more occurrences of the referenced value in the underlying dimension table.

Figure 3.3 shows the members of the Calendar hierarchy in the Date dimension. In the Calendar hierarchy, the items CY 2005, H1 CY 2005, H2 CY 2005, Q1 CY 2005, Q2 CY 2005 Q3 CY 2005, and Q4 CY 2005 are the members. You can see that the items at each level together form the collection of the members of the hierarchy. You can also query the members of a specific level. For example, Q1 CY 2005, Q2 CY 2005, Q3 CY 2005, and Q3 CY 2005 are members of the Calendar Quarter level for the calendar year CY 2005.

Figure 3.3



In MDX, each specific member of a hierarchy is identified by a unique name. You can access a member in a dimension with its dimension name, hierarchy name, and level name using the name path (using the name of the member) or the key path (using the key of the member). For example, member Q1 CY 2006 in the Calendar hierarchy is represented as:

```
[Date].[Calendar].[Calendar Quarter].[Q1 CY 2006]
```

Use the square brackets to enclose the names whenever you have a name that contains a space, has a number in it, or is an MDX keyword. Otherwise, you can omit the square brackets. In the preceding expression the dimension name Date is an MDX keyword and hence must be enclosed within brackets. (Although the brackets are optional, it's good practice to use them.)

The following three representations are also valid for the member Q1 CY 2006:

```
[Date].[Calendar].[Q1 CY 2006]                              (1)
[Date].[Calendar].[CY 2006].[H1 CY 2006].[Q1 CY 2006]       (2)
[Date].[Calendar].[Calendar Quarter].&[2006]&[1]             (3)
```

In the first, the member is represented in the format *[DimensionName].[HierarchyName].[MemberName]*. You can use this format as long as there are no two members with the same name. For example, if quarter 1 in each year is called Q1, you cannot use this format; you would need to qualify using the level name in the MDX expression. If you do use the preceding format, it always retrieves Q1 for the first year in the hierarchy. In the second format, you can see the navigational path for the member clearly because you see all the members in the path. The final format uses the key path where the keys of the members in a path are represented as

`&[MemberName]`. When you use the key path, the members are always preceded with the ampersand (`&`) symbol. In general, you can use the following format for accessing a member:

```
[DimensionName].[HierarchyName].[LevelName].[MemberName]
```

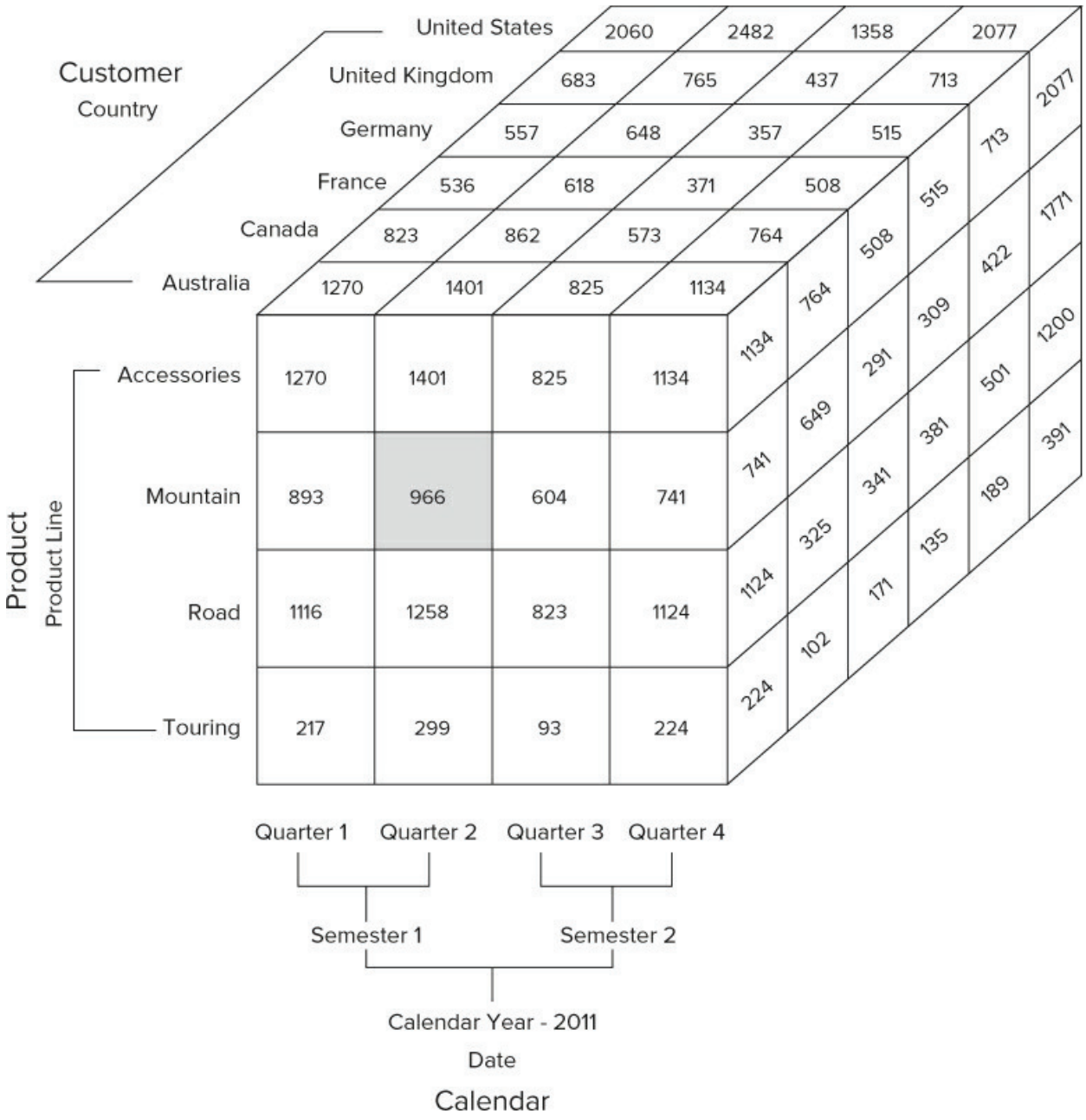
This format is predominantly used in this chapter as well as the rest of the book. The formats described in this section are called as `Uniquename` of a specific dimension member and are included as examples only. We recommend that you retrieve and use the `Uniquename` the server generates when you are querying a specific member.

Cells

Figure 3.2 shows three faces of a cube. The front face is divided into 16 squares, and each square holds a number. Assume the number within each square is the measure [Internet Sales Amount]. If you view the remaining visible faces of the cube, you can see that each square you analyzed in the front face of the cube is actually a small cube. The top-right-corner square of the front face contains the value 1134; the same number is represented on the other sides as well. This smaller cube is referred to as a *cell*. Cells hold the data values of all measures in the cube. If the data value for a measure within a cell is not available, the corresponding measure value is Null.

The number of cells within a cube depends on the number of hierarchies within each dimension and the number of members in each hierarchy. Similar to a three-dimensional coordinate space, where each point is represented by an X, Y, and Z coordinate value, each cell within a cube is represented by dimension members. In the illustration shown in Figure 3.4, you can see three dimensions: Product, Customer, and Date. Assume that each of these dimensions has exactly one hierarchy; then you can see that Product Line has four members, Calendar has four members (considering only quarters), and Country has six members. Therefore, the number of cells is equal to 4 *4* 6 = 96 cells.

Figure 3.4



Assume you want to retrieve the data shown by the shaded area in the cube. The Sales amount value in this cell is 966. This cell is located at the intersection of Product=Mountain, Date=Quarter2, and Customer=Australia. To get this data, your MDX query needs to uniquely identify the cell that contains the value 966. That MDX query is:

```
SELECT Measures.[Internet Sales Amount] ON COLUMNS
FROM [Adventure Works]
WHERE ( [Date].[Calendar].[Calendar Quarter].&[2011]&[2],
    [Product].[Product Line].[Mountain],
    [Customer].[Country].[Australia] )
```

You can see from this query that you are selecting the `Measures.[Internet Sales Amount]` value from the Adventure Works cube based on a specific condition mentioned in the query's `WHERE` clause, which uniquely identifies the cell.

Tuples

An MDX expression that uniquely identifies a cell or a section of a cube is called a *tuple*. A tuple is represented by one member from each dimension, separated by a comma, and enclosed within parentheses. A tuple does not necessarily need to explicitly contain members from all the dimensions. Following are some examples of tuples based on the Adventure Works:

- `([Customer].[Country].[Australia])`
- `([Date].[Calendar].[2011].[H1 CY 2011].[Q1 CY 2011], [Customer].[Country].[Australia])`

- `([Date].[Calendar].[2011].[H1 CY 2011].[Q1 CY 2011], [Product].[ProductLine].[Mountain], [Customer].[Country].[Australia])`

Tuples 1 and 2 do not contain members from all the dimensions in the cube. Therefore, they represent sections of the cube. A cube section represented by a tuple is called a *slice*.

A tuple represented by a single member is called a *simple tuple* and does not need to be enclosed within parentheses. For example, `([Customer].[Country].[Australia])` is a simple tuple and can be referred to as `[Customer].[Country].[Australia]` or simply `Customer.Country.Australia`. When there is more than one dimension in a tuple, it needs to be enclosed in parentheses however.

When you refer to the tuple `([Customer].[Country].[Australia])`, you actually refer to the 16 cells that correspond to the country Australia in Figure 3.4. Therefore when you retrieve the data held by the cells pointed to by this tuple, you are retrieving the Internet Sales Amount of all the customers in Australia. The Internet Sales Amount value for the tuple `[Customer].[Country].[Australia]` is an aggregate of the cells encompassed in the front face of the cube. The MDX query to retrieve data represented by this tuple is

```
SELECT Measures.[Internet Sales Amount] ON COLUMNS
FROM [Adventure Works]
WHERE ([Customer].[Country].[Australia])
```

The result of this query is $9,061,000.58.

The order of the members used to represent a tuple does not matter, and because a tuple uniquely identifies a cell, it cannot contain more than one member from each dimension.

Sets

A collection of tuples forms a new object called a *set*. This collection of tuples is defined using the exact same set of dimensions, both in type and number. Sets are frequently used in MDX queries and expressions.

A set is specified within curly brace characters (`{` and `}`). Set members are separated by commas. The following examples illustrate sets:

- **Example 1:** The tuples `(Customer.Country.Australia)` and `(Customer.Country.Canada)` are resolved to the exact same hierarchy `Customer.Country`. A collection of these two tuples is a valid set and is specified as:

```
{(Customer.Country.Australia), (Customer.Country.Canada)}
```

- **Example 2:** Each of the following tuples has the three dimensions: Date, Product, and Customer:

```
1.   ([Date].[Calendar].[2008].[H1 CY 2008].[Q1 CY 2008], [Product]          .[Product Line].
[Mountain], [Customer].[Country].[Australia]),
2.   ([Product].[Product Line].[Mountain], [Customer].[Country].[Australia],
         ([Date].[Calendar].[2006].[H1 CY 2006].[Q1 CY 2006])
3.   ([Customer].[Country].[Australia], [Date].[Calendar].[2007]          .[H1 CY 2007].[Q1
CY 2007], [Product].[Product Line].[Mountain] )
```

The members in the `[Date].[Calendar]` hierarchy of the three preceding tuples are different; therefore, these tuples refer to different cells. As per the definition of a set, a collection of these tuples is a valid set and is shown here:

```
{ ([Date].[Calendar].[2008].[H1 CY 2008].[Q1 CY 2008], [Product].[Product
Line].[Mountain], [Customer].[Country].[Australia]), ([Product].[Product
Line].[Mountain], [Customer].[Country].[Australia],
([Date].[Calendar].[2006].[H1 CY 2006].[Q1 CY
2006]),([Customer].[Country].[Australia], [Date].[Calendar].[2007].[H1 CY
2007].[Q1 CY 2007], [Product].[Product Line].[Mountain] )}
```

A set can contain zero, one, or more tuples. A set with zero tuples is referred to as an empty set. An empty set is represented as:

```
{    }
```

A set can contain duplicate tuples. An example of such a set is:

```
{Customer.Country.Australia, Customer.Country.Canada,
        Customer.Country.Australia}
```

This set contains two instances of the tuple `Customer.Country.Australia`. A few important points to understand about tuples and sets are:

- A member of a dimension (Example: `([Date].[Calendar].[2008].[H1 CY 2008].[Q1 CY 2008]`) by itself forms a tuple and a set. Hence, it can be used as such in MDX queries.
- If there is a tuple that is specified by only one hierarchy, you do not need the parentheses to specify it as a set. This tuple can be directly used in MDX queries.
- When there is a single tuple specified in a query, you do not need curly braces to indicate it should be treated as a set. When the query is executed, the tuple is implicitly converted to a set.

In general it is a good practice to use the parenthesis and curly braces while you write MDX queries to ensure the tuples or set

specified in your MDX queries are accurate.

MDX Queries

Chapter 2 introduced you to the MDX `SELECT` statement with its syntax:

```
[WITH <formula_expression> [, <formula_expression> ...]]
SELECT [<axis_expression>, [<axis_expression>...]]
FROM [<cube_expression>]
[WHERE [slicer_expression]]
```

You might be wondering whether the `SELECT`, `FROM`, and `WHERE` clauses are the same as those in Structured Query Language (SQL). Even though they look identical, the MDX language is different and supports more complex operations. The keywords `WITH`, `SELECT`, `FROM`, and `WHERE` along with the expressions following them are referred to as *clauses*. In the preceding MDX query template, anything specified within square brackets means it is optional.

Considering that the `WITH` and `WHERE` clauses are optional, the simplest possible MDX query should be the following:

```
SELECT
FROM [Adventure Works]
```

This MDX query returns a single value. Which value, you might ask? Recall that fact data is stored in a special dimension called Measures. When you send this query to Analysis Services, you get the value of the default member from the Measures dimension which, for the Adventure Works cube, is Reseller Sales Amount from the Reseller Sales measure group. The result of this query is the aggregated value of all the cells in the cube for this measure for the default values of each cube dimension.

SELECT Statement and Axis Specification

The MDX `SELECT` statement allows retrieving data with more than just two dimensions. Indeed, MDX provides you with the capability of retrieving data on one, two, or many axes. When referring to an axis dimension, this actually corresponds to a hierarchy for Analysis Services because you include hierarchies in the MDX statement.

The syntax for axis specification in a `SELECT` statement is:

```
SELECT [<axis_expression>, [<axis_expression>...]]
```

The *axis_expressions* refer to the dimension data you want to retrieve. The data from these dimensions are projected onto the corresponding axes. The syntax for an *axis_expression* is:

```
<axis_expression> := <set> ON (axis | AXIS(axis number) | axis number)
```

Axis dimensions are used to retrieve result sets. The set, a collection of tuples, is defined to form an axis dimension. MDX provides you with the ability to specify up to 128 axes in a `SELECT` statement. The first five axes have aliases: `COLUMNS`, `ROWS`, `PAGES`, `SECTIONS`, and `CHAPTERS`. Axes can also be specified as a number, which allows you to specify more than five dimensions in your `SELECT` statement. Take the following example:

```
SELECT   Measures.[Internet Sales Amount] ON COLUMNS,
         [Customer].[Country].MEMBERS ON ROWS,
         [Product].[Product Line].MEMBERS ON PAGES
FROM [Adventure Works]
```

Three axes are specified in the `SELECT` statement. Data from dimensions Measures, Customers, and Product are mapped on to the three axes to form the axis dimensions. This statement could equivalently be written as shown below. Please do note that most of the client tools including SQL Server Management Studio (SSMS) may not be able to retrieve data from this query since it contains three axes.

```
SELECT   Measures.[Internet Sales Amount] ON 0,
         [Customer].[Country].MEMBERS ON 1,
         [Product].[Product Line].MEMBERS ON 2
FROM [Adventure Works]
```

No Shortcuts! In MDX you cannot create a workable query that omits lower axes. If you want to specify a `PAGES` axis, you must also specify `COLUMNS` and `ROWS`.

FROM Clause and Cube Specification

The `FROM` clause determines the cube from which you retrieve and analyze data. It's a necessity for any MDX query. The syntax of the

FROM clause is

```
FROM <cube_expression>
```

The *cube_expression* denotes the name of a cube from which you want to retrieve data. You can define just one cube name, which is called the *cube context,* and the query is executed within this cube context. That is, every part of *axis_expressions* are retrieved from the cube context specified in the FROM clause:

```
SELECT [Measures].[Internet Sales Amount] ON COLUMNS
FROM [Adventure Works]
```

This is a valid MDX query that retrieves data from the [Internet Sales Amount] measure on the X-axis. The measure data is retrieved from the cube context [Adventure Works].

Subselect Clauses

MDX queries can also contain a FROM clause called *subselect*, which allows you to restrict your query to a subcube instead of the entire cube. The syntax of the subselect clause is:

```
[WITH <formula_expression> [, <formula_expression> ...]]
SELECT [<axis_expression>, <axis_expression>...]]

FROM [<cube_expression> | (<sub_select_statement>)]
[WHERE <expression>]
[[CELL] PROPERTIES <cellprop> [, <cellprop> ...]]

<sub_select_statement> =
SELECT [<axis_expression> [, <axis_expression> ...]]
FROM [<cube_expression> | (< sub_select_statement >)]
[WHERE <expression>]
```

The *cube_expression* in the SELECT statement can now be replaced by another SELECT statement called the *sub_select_statement*. You can have nested subselect statements up to any level. The subselect clause restricts the cube space to the subselect's specified dimension members. Outer queries can therefore see only the dimension members that are specified in the inner subselect clauses, as illustrated in the following MDX query that uses subselect syntax:

```
SELECT NON EMPTY { [Measures].[Internet Sales Amount] } ON COLUMNS,
NON EMPTY { ([Customer].[Customer Geography].[Country].ALLMEMBERS ) }
DIMENSION PROPERTIES MEMBER_CAPTION, MEMBER_UNIQUE_NAME ON ROWS
FROM (
   SELECT ( { [Date].[Fiscal].[Fiscal Year].&[2008],
              [Date].[Fiscal].[Fiscal Year].&[2009]
            }
          )
   ON COLUMNS
   FROM (
         SELECT ( { [Product].[Product Categories].[Subcategory].&[26],
                    [Product].[Product Categories].[Subcategory].&[27] } )
         ON COLUMNS
         FROM [Adventure Works]
         )
      )
WHERE ( [Product].[Product Categories].CurrentMember,
        [Date].[Fiscal].CurrentMember
      )
CELL PROPERTIES VALUE, BACK_COLOR, FORE_COLOR, FORMATTED_VALUE
```

WHERE Clause and Slicer Specification

The WHERE clause adds a whole new level of power to restricting queries to return only wanted data. Although the MDX SELECT statement identifies the dimensions and members a query returns, the WHERE statement limits the result set by some criteria. The concept is the same in SQL, but keep in mind that, in MDX, members are elements that make up a dimension's hierarchy.

A Product table, when modeled as a cube, might contain two measures, Sales and Weight, and a Product dimension with the hierarchies ProductID, ProductLine, and Color. In this example the Product table is used as a fact as well as a dimension table. The following MDX query restricts the results only to those products with a silver color:

```
SELECT Measures.[Sales] ON COLUMNS,
[Product].[Product Line].MEMBERS on ROWS
FROM  [ProductsCube]
```

```
WHERE ([Product].[Color].[Silver])
```

The axes `COLUMNS` and `ROWS` refer to the Sales amount per Product Line, and the MDX `WHERE` clause refers to a slice on the cube that contains those products that have silver color. As you can see, even though the SQL and MDX queries look similar, their semantics are quite different.

Slicer Dimension

The *slicer dimension* is what you build when you define the `WHERE` statement. It is a filter that removes unwanted members in a dimension. In Analysis Services, the dimensions are actually hierarchies. What makes things interesting is that the slicer dimension includes any axis in the cube including those that are not explicitly included in any of the queried axes. Each hierarchy in a dimension always has a default member. (You learn about default members in Chapters 8). The default members of hierarchies not included in the query axes are used in the slicer axis. Regardless of how it gets its data, the slicer dimension accepts only MDX expressions that evaluate to a single set. When there are tuples specified for the slicer axis, MDX evaluates those tuples as a set, and the results of the tuples are aggregated based on the measures included in the query and the aggregation function of that specific measure.

WITH Clause, Named Sets, and Calculated Members

The MDX `WITH` clause provides you with the ability to create calculations that must be formulated within the scope of a specific query. In addition, you can retrieve data from outside the context of the current cube by using the `LookupCube` MDX function.

Typical calculations that are created using the `WITH` clause are named sets and calculated members. In addition to these, the `WITH` clause enables you to define cell calculations, load a cube into an Analysis Services cache for improving query performance, alter the contents of cells by calling functions in external libraries, and additional advanced capabilities such as solve order and pass order.

The syntax of the `WITH` clause is:

```
[WITH <formula_expression> [, <formula_expression> ...]]
```

You can specify several calculations in one `WITH` clause. The *formula_expression* can vary depending on the type of calculation. Calculations are separated by commas.

Named Sets

A set expression, even though simple, can often be quite lengthy and might increase query complexity, but you can mitigate this issue by defining sets dynamically and assigning a name that is then used within the query. Think of it as an alias for the collection of tuples in the set that can be used anywhere within the query as an alternative to specifying the actual set expression. This is called a *named set*.

Suppose you want to retrieve the Sales information for selected customers in Europe. Your MDX query could look like this:

```
SELECT Measures.[Internet Sales Amount] ON COLUMNS,
{[Customer].[Country].[Country].&[France],
[Customer].[Country].[Country].&[Germany],
[Customer].[Country].[Country].&[United Kingdom]} ON ROWS
FROM [Adventure Works]
```

This query is not too lengthy, but you can imagine a query that would contain a lot of members and functions being applied to this specific set several times within the query. Instead of specifying the complete set every time, you can create a named set and then use it in the query as follows:

```
WITH SET [EUROPE] AS '{[Customer].[Country].[Country].&[France],
[Customer].[Country].[Country].&[Germany],[Customer].[Country].[Country].
        &[United Kingdom]}'

SELECT Measures.[Internet Sales Amount] ON COLUMNS,
[EUROPE] ON ROWS
FROM [Adventure Works]
```

The *formula_expression* for the `WITH` clause with a named set is:

```
formula_expression :=  [DYNAMIC] SET <set_alias_name> AS ['']<set>['']
```

The *set_alias_name* can be any alias name and is typically enclosed within square brackets. The keywords `SET` and `AS` are used in this expression to specify a named set. The keyword `DYNAMIC` (explained later in this chapter) is optional. The actual set of tuples does not need to be enclosed within single quotes. The single quotes in the MDX query are optional.

Global Named Sets

Named sets created within an MDX query can be accessed only within the scope of the query. Other queries within the same session or other users in different sessions cannot access these named sets in their queries. However, some of the named sets might be useful for others users.

Analysis Services provides ways to define named sets within a specific session where you can send multiple queries, or in a cube's

MDX script where they can be accessed by multiple users. This is done using the CREATE statement, as shown here:

```
CREATE SET [Adventure Works].[Europe]
AS { [Customer].[Country].[Country].&[France],
  [Customer].[Country].[Country].&[Germany],
  [Customer].[Country].[Country].&[United Kingdom] };
```

Instead of the WITH clause that you used in the MDX query for set creation, the CREATE statement allows you to create a set within the scope of a session or the entire cube. You need to specify the cube name as a prefix, but you do not specify the name of a dimension because the created set is not considered to be part of any single dimension.

After the set has been created, it can be accessed in any query. If it were created within a session, it would be valid only within that specific session and could not be used by users in other sessions. If named sets are to be used by several users, you should create them in the cube scope by defining them in an MDX script. Named sets can be in one of three scopes when an MDX query is being executed:

- They can be within the query scope where they are defined with the WITH clause in MDX.
- They can be within the session scope where they can be created within a specific session using the CREATE SET statement you just learned.
- They can be scoped as global and defined within an MDX script using the CREATE SET statement.

Static and Dynamic Named Sets

Analysis Services supports the useful ability to define dynamically evaluated sets, yet before speaking further about dynamic sets, it makes sense to discuss static sets briefly. Use the following CREATE SET statement to illustrate the point, which creates a named set of a global scope (in the MDX script of the cube):

```
CREATE SET CURRENTCUBE.[Static Top 10 Customers]
  AS TopCount
    (
      [Customer].[Customer].[Customer].MEMBERS,
      10,
      [Measures].[Internet Sales Amount]
    ) ;
```

Prior to Analysis Services 2008, a set would be evaluated once in the context it was created. Imagine this example defined in a query scope using WITH SET rather than CREATE SET. Evaluating the top ten customers once in the context of a query is just fine because the query sets the dimensional context and uses it once. Defining the set in the scope of a session or at the cube level (to be used across all future client sessions) is clearly problematic for this type of data-bound set. The set will be evaluated when the client connects or in the current session context so any subsequent filters (say by product or time period) will not result in the set membership being updated and will effectively return the wrong data.

Dynamic sets solve this issue. Hence, the keyword DYNAMIC, which is typically used within MDX scripts to evaluate the expression at query execution time. In practice, this means that the set will not be static and will be evaluated in the context of every query that directly or indirectly references them and with respect to the WHERE clause and subselects.

Here is the preceding set, defined as DYNAMIC:

```
CREATE DYNAMIC SET CURRENTCUBE.[ Dynamic Top 10 Customers]
  AS TopCount
    (
      [Customer].[Customer].[Customer].MEMBERS,
      10,
      [Measures].[Internet Sales Amount]
    ) ;
```

We recommend you create both sets to see the differences in the MDX query results. Using a DYNAMIC set can have a performance impact if the set is very large. Hence, you should evaluate whether you truly need the results to be evaluated dynamically at query time, and then create appropriate named sets for your business.

Caption and Display Folder for Named Sets

Named sets also support the following two properties:

- CAPTION: This property is primarily for scenarios involving session-scoped calculations where greater flexibility in naming (for example, when defining reports) may be wanted and as such is not exposed in the user interface.
- DISPLAY_FOLDER: This property allows for defining a customizable folder structure in which to display metadata. A hierarchical organization may be specified by separating display folders by backslashes (\).

Deleting Named Sets

Named sets are useful for querying because they are easy to read and allow multiple users to access them. However, be aware that there is a memory cost associated with holding them in Analysis Services. If you need to create a large number of named sets that are quite

large in terms of number of tuples, exercise caution. Drop any named sets whenever they are not used. Just as there is a CREATE SET statement, there is a DROP SET statement to delete named sets as well. The syntax is simple:

```
DROP SET <setname>
```

Calculated Members

Calculated members are calculations specified by MDX expressions. They are resolved as a result of MDX expression evaluation rather than just by the retrieval of the original fact data. A typical example of a calculated member is the calculation of year-to-date sales of products. Say the fact data contains only sales information of products for each month and you need to calculate the year-to-date sales. You can do this with an MDX expression using the WITH clause.

The *formula_expression* of the WITH clause for calculated members is:

```
Formula_expression := MEMBER <MemberName> AS ['']<MDX_Expression>[''],
                [ , SOLVE_ORDER = <integer>]
                [ , <CellProperty> = <PropertyExpression>]
```

MDX uses the keywords MEMBER and AS in the WITH clause for creating calculated members. The *MemberName* should be a fully qualified name that includes the dimension, hierarchy, and level under which the specific calculated member needs to be created. The *MDX_Expression* should return a value that calculates the value of the member.

The SOLVE_ORDER, which is an optional parameter, should be a positive integer value if specified. It determines the order in which the members are evaluated when multiple calculated members are defined. The *CellProperty* is also optional and is used to specify cell properties for the calculated member such as the text formatting of the cell contents including the background color.

All the measures in a cube are stored in a special dimension called Measures. Calculated members can also be created on the measures dimension, in which case they are referred to as calculated measures. The following is an example of a calculated member statement:

```
WITH MEMBER MEASURES.[Profit] AS [Measures].[Internet Sales Amount]-
[Measures].[Internet Standard Product Cost]
SELECT measures.profit ON COLUMNS,
 [Customer].[Country].MEMBERS ON ROWS
FROM [Adventure Works]
```

In this example, a calculated member, Profit, has been defined as the difference of the measures [Internet Sales Amount] and [Internet Standard Product Cost]. When the query is executed, the Profit value will be calculated for every country based on the MDX expression.

Global Calculated Members

As with named sets, you can create calculated members using the CREATE statement followed by the keyword MEMBER and the member name, as follows:

```
CREATE MEMBER [Adventure Works].[Measures].[Profit] AS
  '([Measures].[Internet Sales Amount] - [Measures].[Total Product Cost])';
```

You are expected to specify the cube and dimension name for calculated members. In an MDX script, you can use the CURRENTCUBE keyword instead of the cube name as shown here:

```
CREATE MEMBER CURRENTCUBE.[Measures].[Profit] AS
  '([Measures].[Internet Sales Amount] - [Measures].[Total Product Cost])';
```

However, all measures within a cube are always within a special dimension called Measures. Hence, in the preceding CREATE statement, the cube name and the dimension Measures are specified. The calculated members that are most often created by users are calculated measures, that is, calculated members on the Measures dimension. For convenience, Analysis Services assumes that a calculated member will be in the Measures dimension if it is not explicitly prefixed with Measures. Hence, the following statement is valid syntax in your cube's MDX script:

```
CREATE MEMBER [Profit]
AS '( [Measures].[Internet Sales Amount] - [Measures].[Total Product Cost] )';
```

After the calculated members have been created, you can use them as shown in the following query. The query scope, session scope, and global scope seen for named sets also apply to calculated members.

```
SELECT [Measures].[Profit] ON COLUMNS,
     [Customer].[Country].MEMBERS ON ROWS
FROM [Adventure Works]
```

Analysis Services provides another way to define calculated members at the global scope (in an MDX script). This involves declaring a member first without any definition and later defining the expression, as the following MDX statements demonstrate. Using this technique is a matter of convenience if, say, you want to create a calculated measure and are not sure about the actual expression which later is specified in the MDX script. You can define the calculated member, use it in statements, and finally create the actual expression.

```
CREATE MEMBER [Profit] AS NULL;
```

```
[Measures].[Profit] = [Measures].[Sales Amount] - [Measures].
        [Total Product Cost];
```

Properties of Calculated Members

Analysis Services further extends the calculated member syntax by enabling the definition of three properties:

- `CAPTION`: Changing the caption is most useful for session-scoped calculations, and for this reason it's not exposed in the development tools.
- `DISPLAY_FOLDER`: Simply controls the name of the display folder, as you have likely guessed.
- `ASSOCIATED_MEASURE_GROUP`: A direct reference to an existing measure group that is used to visually group calculated measures with the appropriate physical measures. This property is an XML property, not in the MDX language.

Deleting Calculated Members

Similar to named sets, calculated members can also be dropped using the `DROP MEMBER` statement. The syntax for `DROP MEMBER` is:

```
DROP MEMBER <member name>
```

Ranking and Sorting

Ranking and Sorting are pretty common features in most business analyses, and MDX provides several functions for this purpose, such as `TopCount`, `BottomCount`, `TopPercent`, `BottomPercent`, and `Rank`. The following are basic examples for `TopCount` and `BottomCount` based on Adventure Works DW.

If you are looking for the top *N* product categories based on the sales in all countries, the following query can provide you with the answer.

```
SELECT [Measures].[Internet Sales Amount] ON COLUMNS,
TopCount( [Product].[Product Categories].[Category].MEMBERS, 3,
    [Measures].[Internet Sales Amount] ) ON ROWS
FROM [Adventure Works]
```

On the other hand, to maximize your business you might want to discontinue products that are not providing your best sales. Now see the bottom 10% in terms of Internet sales:

```
//Total number of products contributing towards internet sales – 159 products

SELECT { [Measures].[Internet Sales Amount] } ON COLUMNS,
NON EMPTY [Product].[Product Categories].[Product].MEMBERS ON ROWS
FROM [Adventure Works]

//Bottom 10% (Sales) of the products sold through the internet – 95 products
SELECT { [Measures].[Internet Sales Amount] } ON COLUMNS,
NON EMPTY BottomPercent(
    [Product].[Product Categories].[Product].MEMBERS, 10,
    [Measures].[Internet Sales Amount] ) ON ROWS
FROM [Adventure Works]
```

You can see that there are 159 products that contribute toward Internet sales, and out of these, 95 products contribute to the bottom 10% of the overall sales. Now you can further drill down at each product and identify the cost of selling them over the Internet to see if it makes sense to keep selling these products.

MDX Expressions

MDX expressions are partial MDX statements that evaluate to a value. They are typically used in calculations or in defining values for objects such as default members and default measures, or for defining security expressions to allow or deny access. MDX expressions typically take a member, a tuple, or a set as a parameter and return a value. If the result of the MDX expression evaluation is no value, a Null value is returned. Following are some examples of MDX expressions:

- **Example 1:** This example returns the default member specified for the Customer Geography hierarchy of the Customer dimension.

```
Customer.[Customer Geography].DEFAULTMEMBER
```

- **Example 2:** This MDX expression compares the sales to customers of different countries with sales of customers in Australia.

```
(Customer.[Customer Geography].CURRENTMEMBER, Measures.[Sales Amount]) -
(Customer.[Customer Geography].Australia, Measures.[Sales Amount)
```

You typically use such an expression in a calculated measure. Complex MDX expressions can include various operators in the MDX language along with the combination of the functions available in MDX as follows.

- **Example 3:** This example is an MDX cell security expression that allows employees to see Sales information made by them or by the employees reporting to them and not other employees. This MDX expression uses several MDX functions. (You learn some of these in the next section.) You can see that this is not a simple MDX expression. The preceding MDX expression returns a value True or False based on the employee logged in. Analysis Services allows appropriate cells to be accessed by the employee based on the evaluation.

```
COUNT(INTERSECT( DESCENDANTS( IIF( HIERARCHIZE(EXISTS[Employee].
         [Employee].MEMBERS,
STRTOMEMBER("[Employee].[login].[login].&["+USERNAME+"]")),
        POST).ITEM(0).ITEM(0).PARENT.DATAMEMBER is
HIERARCHIZE(EXISTS([Employee].[Employee].MEMBERS,
        STRTOMEMBER("[Employee].[login].[login].&["+USERNAME+"]")),
        POST).ITEM(0).ITEM(0),
HIERARCHIZE(EXISTS([Employee].[Employee].MEMBERS,
        STRTOMEMBER("[Employee].[login].[login].&["+username+"]")),
        POST).ITEM(0).ITEM(0).PARENT,
HIERARCHIZE(EXISTS([Employee].[Employee].MEMBERS,
STRTOMEMBER("[Employee].[login].[login].&["+USERNAME+"]")),
        POST).ITEM(0).ITEM(0))
).ITEM(0) , Employee.Employee.CURRENTMEMBER))  > 0
```

MDX Operators

The MDX language has several types of operators including arithmetic operators, logical operators, and special MDX operators. An operator is a function that performs a specific action, takes arguments, and returns a result.

Arithmetic Operators

Regular arithmetic operators such as +, −, *, and / are available in MDX. Just as with other programming languages, you can apply these operators on two numbers. You can also use the + and − operators as unary operators on numbers. Use unary operators, as the name indicates, with a single operand (single number) in MDX expressions such as `+ 100` or `− 100`.

Set Operators

The +, −, and * operators, in addition to being arithmetic operators, can also perform operations on the MDX sets. The + operator returns the union of two sets: The − operator returns the difference of two sets, and the * operator returns the cross product of two sets. The cross product of two sets results in all possible combinations of the tuples in each set and helps to retrieve data in a matrix format. For example, if you have two sets, `{Male, Female}` and `{2007, 2008, 2009}`, the cross product, represented as `{Male, Female} * {2007, 2008, 2009}`, is `{(Male,2007), (Male,2008), (Male,2009),(Female,2007), (Female,2008),(Female,2009)}`. The following examples show MDX expressions that use the set operators:

- **Example 1:** The result of the MDX expression:

```
{[Customer].[Country].[Australia]} + {[Customer].[Country].[Canada]}
```

is the union of the two sets as shown here:

```
{[Customer].[Country].[Australia], [Customer].[Country].[Canada]}
```

- **Example 2:** The result of the MDX expression:

```
{[Customer].[Country].[Australia],[Customer].[Country].[Canada]}*
{[Product].[Product Line].[Mountain],[Product].[Product Line].[Road]}
```

is the cross product of the sets as shown here:

```
{([Customer].[Country].[Australia],[Product].[Product Line].[Mountain])
([Customer].[Country].[Australia],[Product].[Product Line].[Road])
([Customer].[Country].[Canada],[Product].[Product Line].[Mountain])
([Customer].[Country].[Canada],[Product].[Product Line].[Road])}
```

Comparison Operators

MDX supports the comparison operators <, <=, >, >=, =, and <>. These operators take two MDX expressions as arguments and return

TRUE or FALSE based on the result of comparing the values of each expression.

- **Example:** The following MDX expression uses the greater than comparison operator, >:

```
Count (Customer.[Country].members) > 3
```

In this example, Count is an MDX function that can count the number of members in the Country hierarchy of the Customer dimension. Because there are more than three members, the result of the MDX expression is TRUE.

Logical Operators

The logical operators that are part of MDX are AND, OR, XOR, NOT, and IS, which are used for logical conjunction, logical disjunction, logical exclusion, logical negation, and comparison, respectively. These operators take two MDX expressions as arguments and return TRUE or FALSE based on the logical operation. You typically use logical operators in MDX expressions for cell and dimension security, which you learn about in Chapter 14.

Special MDX Operators — Curly Braces, Commas, and Colons

You can use the curly braces, represented by the characters { and }, to enclose a tuple or a set of tuples to form an MDX set. Whenever you have a set with a single tuple, the curly brace is optional because Analysis Services implicitly converts a single tuple to a set when needed. When there is more than one tuple to be represented as a set or when there is an empty set, you need to use the curly braces.

You have already seen the comma character used in several earlier examples. The comma character can form a tuple that contains more than one member. By doing this you create a slice of data on the cube. In addition, the comma character can separate multiple tuples specified to define a set. In the set {(Male,2007), (Male,2008), (Male,2009),(Female,2007),(Female,2008), (Female,2009)} the comma character is not only used to form tuples, but also to form the set of tuples.

The colon character defines a range of members within a set. Use it between two nonconsecutive members in a set to indicate inclusion of all the members between them, based on the set ordering (key-based or name-based). For example, if you have the following set:

```
{[Customer].[Country].[Australia], [Customer].[Country].[Canada],
[Customer].[Country].[France], [Customer].[Country].[Germany],
[Customer].[Country].[United Kingdom], [Customer].[Country].[United States]}
```

the following MDX expression:

```
{[Customer].[Country].[Canada] : [Customer].[Country].[United Kingdom]}
```

results in the following set:

```
{[Customer].[Country].[Canada], [Customer].[Country].[France],
[Customer].[Country].[Germany], [Customer].[Country].[United Kingdom]}
```

MDX Functions

MDX functions help address some of the common operations needed in MDX expressions or queries including ordering tuples in a set, counting the number of members in a dimension, and string manipulation required to transform user input into corresponding MDX objects. Because MDX functions are so central to the successful use of Analysis Services, it is best if you jump in and learn some of them now.

MDX Function Categories

In this section, the MDX functions have been categorized in a specific way similar to the product documentations of MDX functions to help you understand them efficiently. You also see some details on select functions of interest, where interest level is defined by the probability you will use a given function in your future BI development work. You can see all the MDX functions in detail in Appendix A (available online at www.wrox.com) .

MDX functions can be called in several ways:

- .Function (read *dot* function)
- **Example:** *Dimension.Name* returns the name of the object being referenced (could be a hierarchy or level/member expression). Perhaps this reminds you of the dot operator in VB.NET or C# programming — that's fine. It's roughly the same idea.
- Function
- **Example:** Username acquires the username of the logged-in user. It returns a string in the following format: *domain-name\username*. Most often you use this in dimension-or cell security-related MDX expressions.
- Function ( )
- **Example:** The function CustomData ( ) requires parentheses, but takes no arguments. You can find more on CustomData ( ) in Appendix A (available online at www.wrox.com).
- Function (arguments)

- **Example: .** `( [Level_Expression [ , Member_Expression] ] )` is an MDX function that takes an argument that can specify both the `level_expression` with the `member_expression` or just the `member_expression` itself, and returns the first member at the level of the `member_expression`. For example, `.(Day, [April])` returns the first member of the Day level of the April member of the default time dimension. (Note the opening period in this example function.)

Set Functions

Set functions, as the category name suggests, operate on sets. They take sets as arguments and often return a set. Some of the widely used set functions are `Crossjoin` and `Filter`.

Crossjoin

`Crossjoin` returns all possible combinations of sets as specified by the arguments to the `Crossjoin` function. If there are N sets specified in the `Crossjoin` function, this results in a combination of all the possible members within that set on a single axis. You see this in the following example:

```
Crossjoin ( Set_Expression [ ,Set_Expression ...] )

SELECT Measures.[Internet Sales Amount] ON COLUMNS,
CROSSJOIN( {Product.[Product Line].[Product Line].MEMBERS},
{[Customer].[Country].MEMBERS}) on ROWS
FROM [Adventure Works]
```

NONEMPTYCROSSJOIN and NONEMPTY

The previous `Crossjoin` example produces the cross product of each member in the Product dimension with each member of the Customer dimension along the sales amount measure. Sometimes such a cross join results in a large number of values being null, but instead of retrieving all the results and then checking for null values, you can restrict these on the server side and optimize the query so that only the appropriate result is retrieved and sent. To eliminate the null values, you can use the keyword `NON EMPTY` on the axis or the MDX functions `NONEMPTYCROSSJOIN`, `NONEMPTY`, and `FILTER`.

Use the `NON EMPTY` operator on an axis to remove the members that result in rows with empty (null) cell values. When you apply `NON EMPTY`, cells with null values are eliminated in the context of members on other axes. You can see this in the following query:

```
SELECT [Measures].[Internet Sales Amount] ON 0,
NON EMPTY [Customer].[Customer Geography].[Country].MEMBERS *
[Product].[Product Categories].MEMBERS ON 1
FROM [Adventure Works]
```

The `NonEmptyCrossjoin` and `NonEmpty` functions remove rows with empty cells in these query results. The syntax is:

```
NonEmpty(Set_Expression [ ,FilterSet_Expression])
NonEmptyCrossjoin(
      Set_Expression [ ,Set_Expression ...][ ,Crossjoin_Set_Count ] )
```

When using the `NonEmptyCrossjoin` function, it uses the default measure unless you specify the wanted measure directly, such as `[Internet Sales Amount]` in the following example. Most users and client tools use the `NonEmptyCrossjoin` function extensively. When using the `NonEmpty` function, you first do the `Crossjoin` and then filter out the tuples that have null values for the Internet Sales amount, as shown in the second query in the following code.

```
SELECT Measures.[Internet Sales Amount] ON COLUMNS,
NONEMPTYCROSSJOIN( {Product.[Product Line].[Product Line].MEMBERS},
{[Customer].[Country].MEMBERS},Measures.[Internet Sales Amount],2 )  ON ROWS
FROM [Adventure Works]
SELECT Measures.[Internet Sales Amount] ON COLUMNS,
NONEMPTY(CROSSJOIN ( {Product.[Product Line].[Product Line].MEMBERS},
{[Customer].[Country].MEMBERS}),Measures.[Internet Sales Amount]) ON ROWS
FROM [Adventure Works]
```

Filter and Having

The `Filter` function helps to restrict the query results based on one or more conditions. The `Filter` function takes two arguments: a set expression and a logical expression. The logical expression is applied on each item of the set and returns a set of items that satisfy the logical condition. The function arguments for the `Filter` function are:

```
Filter( Set_Expression , { Logical_Expression | [ CAPTION | KEY | NAME ]
        =String_Expression } )
```

If you are interested only in the products for which the sales amount is greater than a specific value and are still interested in finding out amounts by countries, you can use the `Filter` function as shown here:

```
SELECT Measures.[Internet Sales Amount] ON COLUMNS,
FILTER(CROSSJOIN( {Product.[Product Line].[Product Line].MEMBERS},
{[Customer].[Country].MEMBERS}),[Internet Sales Amount] >2000000) on ROWS
FROM [Adventure Works]
```

This query filters out all the products for which the sales amount is less than 2,000,000 and returns only the products that have the sales amount greater than 2,000,000.

Another option to eliminate cells that have null values and then apply the filter condition on the resulting set is to use the `HAVING` clause. The syntax for the `HAVING` clause is:

```
SELECT <axis_specification> ON 0,
NON EMPTY <axis_specification> HAVING <filter condition> ON 1
FROM <cube identifier>
```

The following MDX query uses the `HAVING` clause to analyze the gross profit of all the products that have sales amounts greater than $100,000 for a product in a given year:

```
SELECT { [Measures].[Gross Profit] } ON 0,
NON EMPTY [Product].[Product Categories].[Product] *[Date].[Calendar].[Calendar Year]
    HAVING [Sales Amount] > 100000 ON 1
FROM [Adventure Works]
```

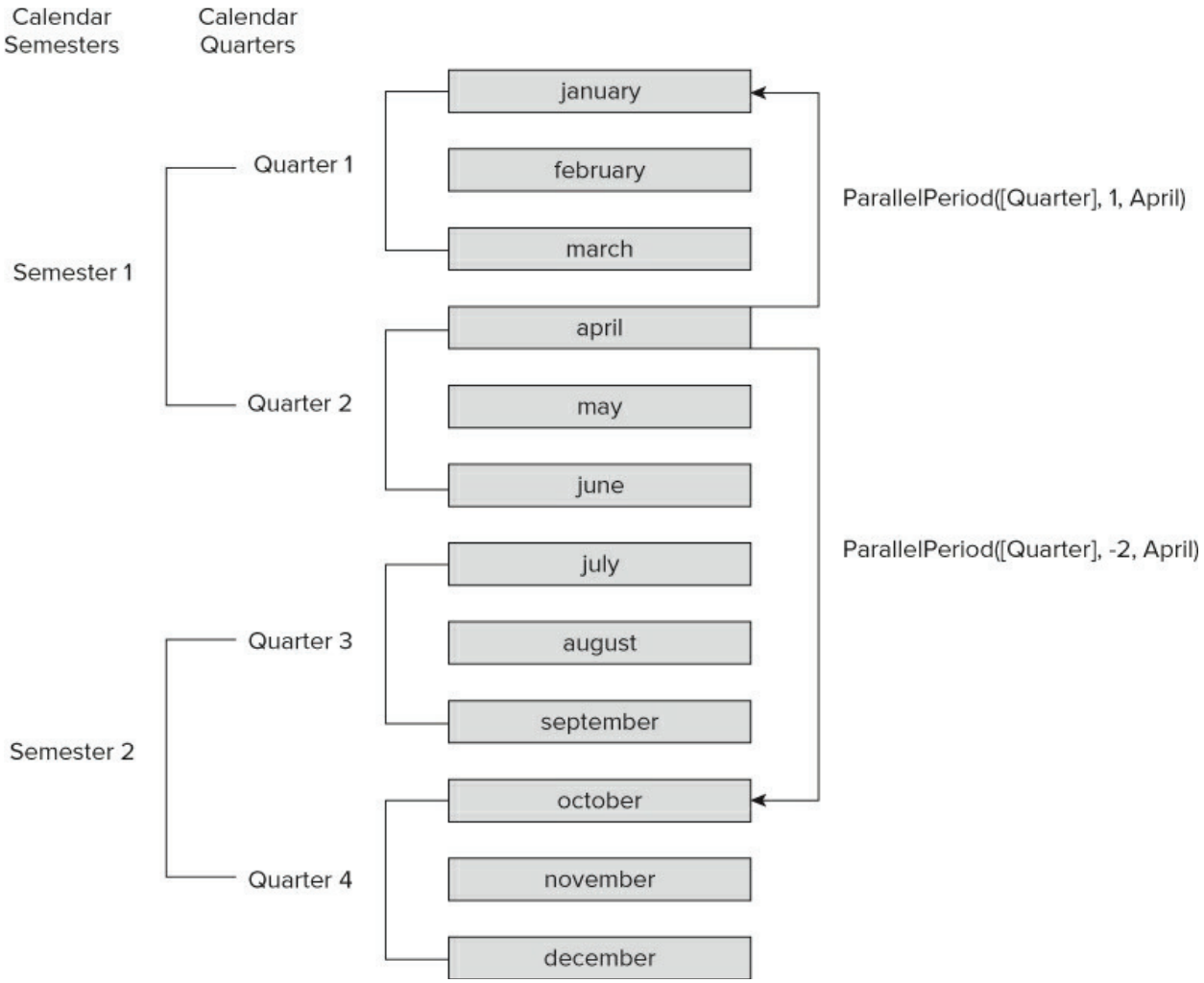
Member Functions

You can use member functions for operations on members such as retrieving the current member, ancestor, parent, children, sibling, next member, and so on. All the member functions return a member. One of the most widely used member functions is `ParallelPeriod`, which helps you to retrieve a member in the Time dimension based on a given member and certain conditions. The function definition for `ParallelPeriod` is:

```
ParallelPeriod( [ Level_Expression [ ,Numeric_Expression [ , Member_Expression ] ] ] )
```

Figure 3.5 shows an illustration of the `ParallelPeriod` function, which returns a member from a Time dimension relative to a given member for a specific time period. The *`Numeric_Expression`* is the number of time periods to move to. You learn more about time dimensions in Chapter 5.

Figure 3.5



Numeric Functions

Numeric functions are handy when you define the parameters for an MDX query or create any calculated measure. Plenty of statistical functions are in this group, including standard deviation, sample variance, and correlation. The most common is a simple one, `Count`, along with its close cousin, `DistinctCount`. You can use the `Count` function to count the number of items in the collection of a specific object such as a dimension, a tuple, a set, or a level. The `DistinctCount` function, on the other hand, takes a *`Set_Expression`* as an argument and returns a number that indicates the number of distinct items in the *`Set_Expression`*, not the total count of all items. Following are the function definitions for each:

```
Count ( Dimension | Tuples | Set| Level)
DistinctCount ( Set_Expression )
```

Take a look at the following query:

```
WITH MEMBER Measures.CustomerCount AS DistinctCount(
Exists([Customer].[Customer].MEMBERS,[Product].[Product Line].Mountain,
"Internet Sales"))
SELECT Measures.CustomerCount ON COLUMNS
FROM [Adventure Works]
```

The `DistinctCount` function counts the number of distinct members in the Customer dimension who have purchased products in the Mountain product line. If a customer has purchased multiple products from the specified product line, the `DistinctCount` function counts the customer just once. The MDX function `Exists` filters customers who have purchased only product line Mountain through the Internet.

Dimension Functions, Level Functions, and Hierarchy Functions

Functions in these groups are typically used for navigation and manipulation. Following is an example of just such a function, the `Level` function from the Level group:

```
SELECT [Date].[Calendar].[Calendar Quarter].[Q1 CY 2008].LEVEL ON COLUMNS
FROM [Adventure Works]
```

This query results in a list of all the quarters displayed in the results. The reason is that `[Date].[Calendar].[Calendar Quarter].[Q1 CY 2008].LEVEL` evaluates to `[Date].[Calendar Year].[Calendar Semester].[Calendar Quarter]`. When a level is specified in an MDX expression, it is implicitly converted to a set and hence all the members at that level are implied in MDX. Due to this, you get the list of all quarters for the specified calendar year and semester.

String Manipulation Functions

To extract the names of sets, tuples, and members in the form of a string, you can use functions such as `MemberToStr ( <Member_Expression> )`. To do the inverse — take a string and create a member expression — you can use `StrToMember ( <String> )`. String manipulation functions are useful when accepting parameters from users and transforming them to corresponding MDX objects. However, there is a significant performance cost. Hence, you use these functions only if necessary.

```
SELECT STRTOMEMBER ('[Customer].[Country].[Australia]' ) ON COLUMNS
FROM [Adventure Works]
```

Other Functions

Four other function categories exist: `Subcube` and `Array` both have one function each. The final two categories are logical functions, which allow you to do Boolean evaluations on MDX objects, and tuple functions, which you can use to access tuples from an MDX set or convert a string to a tuple. You have seen some of them in this chapter, such as `NonEmpty` and `Exists`. Detailed information about these functions is provided in Appendix A (available online at www.wrox.com).

MDX Scripts

MDX scripts can contain complex calculations and consist of various types of MDX statements and commands, each separated by semicolons. `CALCULATE`, `SCOPE`, `IF-THEN-ELSE`, and `CASE` are just a few of the MDX statements that you can use within an MDX script. MDX scripts are meant to be structured in a way that the flow of the statements is simple and readable. The scripting language is based on a procedural programming model, and although it may sound complex, it is actually simpler to use than certain predecessor technologies.

When you create a cube using the Cube Wizard within the SSDT, a default MDX script is created for you. You can see the script definitions on the Calculations tabs of the Cube Designer, as shown in Chapters 6 and 9.

Analysis Services also includes tools to help you debug MDX scripts interactively, more like debugging a program, to identify any semantic errors in calculations defined in the script. Syntactic errors are automatically flagged when the cube is deployed to an Analysis Services instance. The real value of an MDX script is to define calculations that assign values to cells in the cube space based upon potentially complex business logic.

MDX Script Execution

MDX script typically contain the calculations that need to be applied to a cube including creation of calculated members, named sets, and calculations for the cells. However, users of the cube can have different security permissions defined for dimensions and cubes within a database. Therefore, when a user connects to a cube, Analysis Services evaluates the security permissions for the user and assigns the user a cube context. This ensures that the data populated within the cube for the user is based on the security permissions of that user. If a cube context with that specific set of permissions already exists, the user is automatically assigned to that cube context. You learn more about securing data in Chapters 9 and 14.

CALCULATE Statement

The `CALCULATE` statement is added to a cube's MDX script by the Cube Wizard to indicate that Analysis Services should aggregate the data from the lowest level of attributes and hierarchies to higher levels. Aggregation of data to various levels of a hierarchy is illustrated in Figures3.3, 3.4, 3.5, 3.6 through 3-8.

When a user accesses the cube, the fact data first loads into the cube (refer to Figure 3.6), which represents data for a specific year. This is referred to as PASS 0 within Analysis Services. After the fact data has been loaded into the cube, Analysis Services applies calculations for the cells based on an MDX script or dimension attributes. When the `CALCULATE` statement is encountered in the script, a new `PASS`, `PASS 1`, is created where the fact data, aggregated for appropriate levels of the dimension hierarchies, is accessible to end users. For hierarchies with only one level, there is no need to aggregate the data, but the Date hierarchy in Figure 3.7 and Figure 3.8 has the levels Semester and Year for which the data needs to be aggregated. Analysis Services aggregates the data for the Semester and Year levels as shown so that you can query the aggregated data.

Figure 3.6

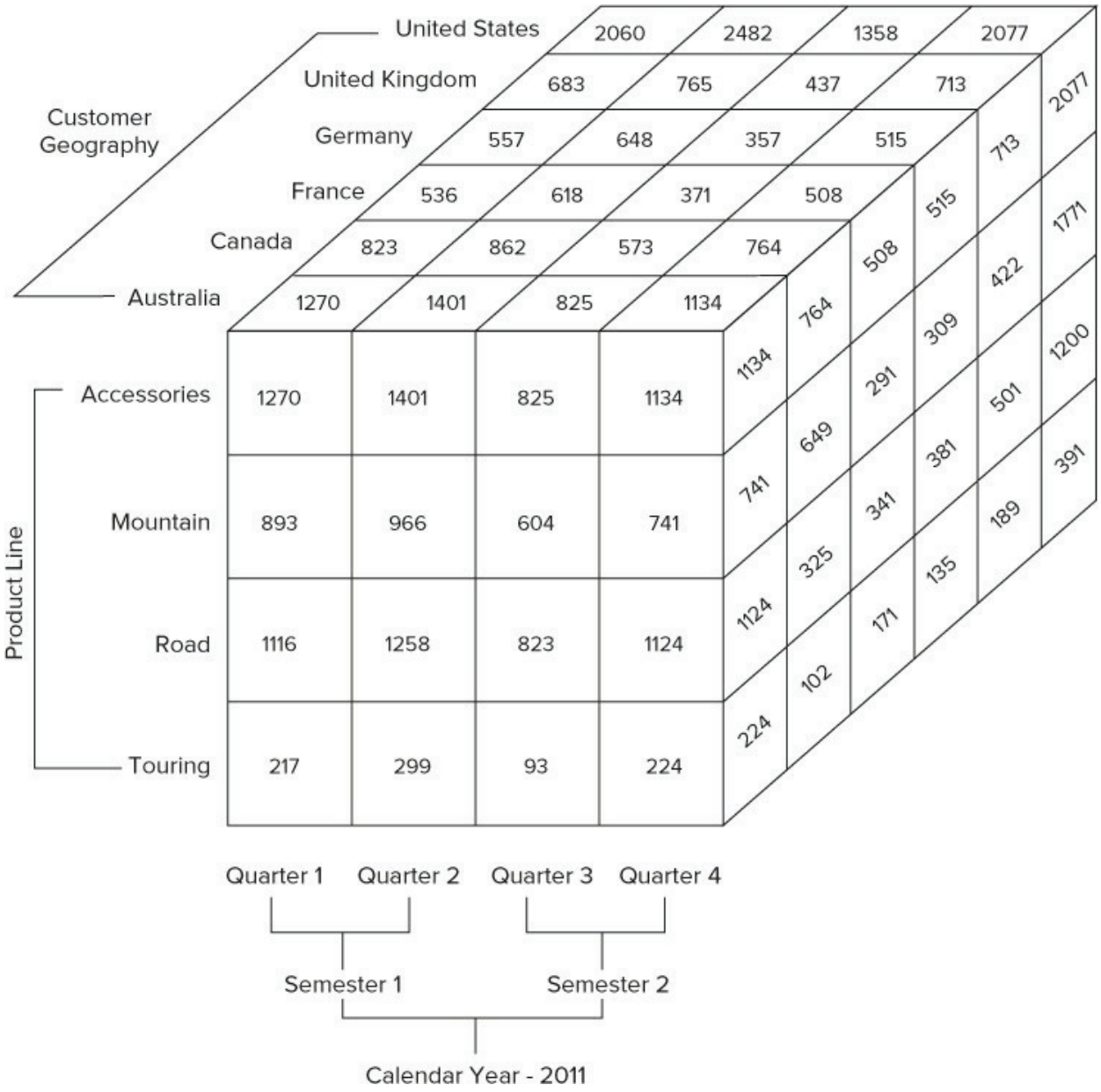


Figure 3.7

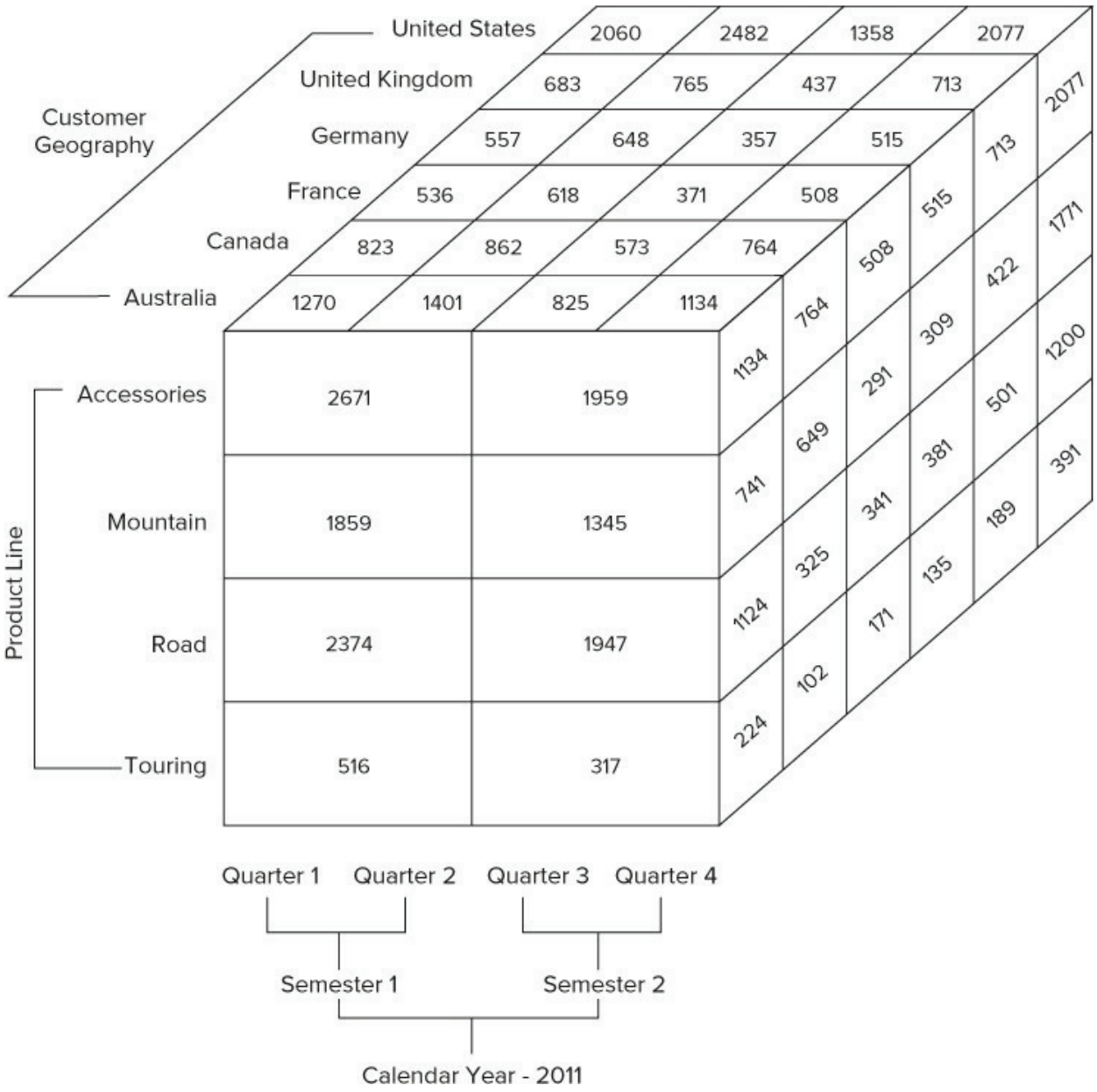
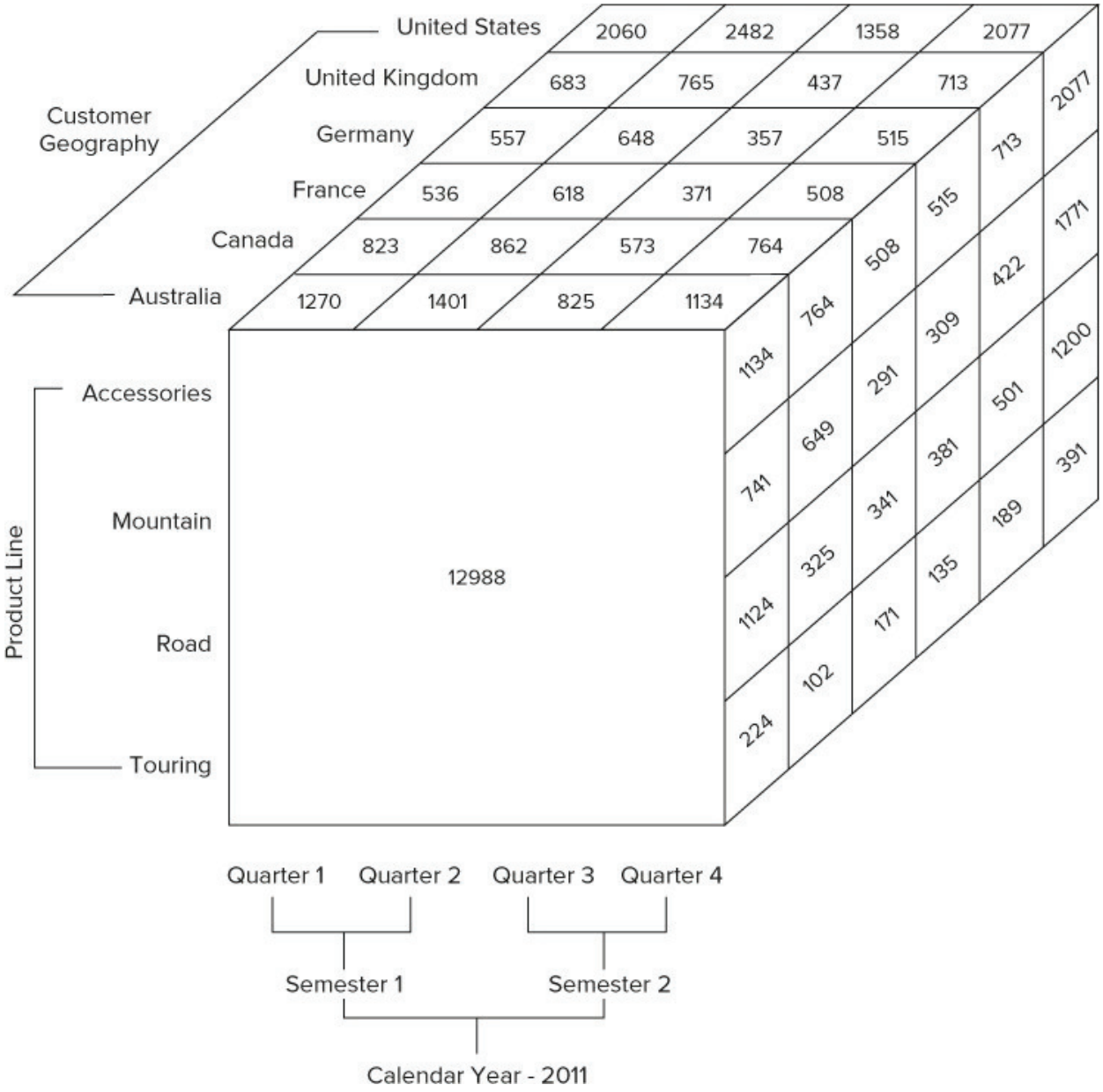

Customer Geography
United States
United Kingdom
Germany
France
Canada
Australia
2060
2482
1358
2077
683
765
437
713
557
648
357
515
536
618
371
508
823
862
573
764
1270
1401
825
1134
2077
1771
1200
391
713
422
501
189
515
309
381
135
508
291
341
171
764
649
325
102
1134
741
1124
224
Product Line
Accessories
Mountain
Road
Touring
2671
1959
1859
1345
2374
1947
516
317
Quarter 1
Quarter 2
Quarter 3
Quarter 4
Semester 1
Semester 2
Calendar Year - 2011

Figure 3.8



If the `CALCULATE` statement is not specified in the MDX script, you cannot query the aggregated data for Semester and Year. If you attempt such a query you, get null values for those levels. If you are missing a `CALCULATE` statement in the MDX script, you can retrieve the fact data for aggregated levels only when you include all hierarchies of all dimensions in your query.

If a cube does not have an MDX script defined, Analysis Services uses a default MDX script with a `CALCULATE` statement. It is not expected that users have MDX scripts without the `CALCULATE` statement other than by mistake. If you do not have any calculations defined and your queries return null values for various hierarchies, you should check if the `CALCULATE` statement is included in the MDX script.

Cube Space

The cube space (cells) can be quite large, even for a small Analysis Services cube that contains less than ten dimensions, with each dimension containing approximately ten attributes. When referring to the data in the cube space, we do not just refer to the data in the fact table. Some cells in the cube space retrieve data through calculations of the data in the fact table or are aggregated up across dimensions due to the cube modeling scheme. Still, the cells that have data are quite sparse compared to the entire cube space.

Referring to the cube space accessible to the users and that can be manipulated through calculations as the real cube space, you can access certain cells in the cube space through MDX queries that are actually not part of the real cube space. For example, assume a Customer dimension that has attributes Name, Gender, and Marital Status. There is a customer named Aaron Flores who is Male in the Adventure Works DW Multidimensional sample database, but the cell corresponding to `Customer.Customer.[Aaron Flores]` and `Customer.Gender.Female` does not exist in the cube space. If you request the cell corresponding to this coordinate, you get a null value.

AUTO EXISTS

If you do a cross-join of multiple attributes, you get the entire cross product of all the members of the attributes involved in the cross-join. However, if you do a cross-join of the attributes within the same dimension, Analysis Services eliminates the cells corresponding to the attributes' members that do not exist with one another. This specific behavior is called `AUTO EXISTS`, which can be interpreted as an `EXISTS` MDX function automatically being applied to attributes within the same dimension. For example, if you query `[Internet Sales Amount]` along with customers across states and countries, your MDX query will be:

```
SELECT [Measures].[Internet Sales Amount] ON COLUMNS,
[Customer].[Country].[Country].MEMBERS *
[Customer].[State-Province].[State-Province].MEMBERS
ON ROWS
FROM [Adventure Works]
```

The results of this MDX query have only the states within a specific country instead of a regular cross-join of the members of both hierarchies. Alberta, which is a state in Canada, does not exist in Australia, and hence you do not have a tuple containing Australia and Alberta in your result.

Cell Calculations and Assignments

MDX provides several ways to specify calculations, such as calculated members, calculated measures, custom rollups (discussed in Chapter 8), and unary operators (also discussed in Chapter 8). Using these features to affect a group of cell values or even a single cell value is not easy.

Query, Session, and Cube Scope

The `CREATE CELL CALCULATION` statement, similar to calculated members and named sets, can be specified at a query, session, or cube scope. The syntax for the `CREATE CELL CALCULATION` statement is

```
CREATE CELL CALCULATION  <CubeName>.<formula name>
FOR <SetExpression> AS <MDX Expression>, <cell property list>
CONDITION = <Logical Expression>
```

In the preceding syntax, the *<formula name>* is an identifier for the cell calculation statement. The *SetExpression* resolves to a set of tuples for which the cell values will be changed. The cell property list is an optional set of properties for the cell such as `DISABLED`, `DESCRIPTION`, `CALCULATION_PASS_NUMBER`, and `CALCULATION_PASS_DEPTH`, which can be applied to the cells being evaluated separated by commas.

For global scope you need to define the cell calculation statements within MDX scripts. If you want to use this within the query scope, you need to use the `CREATE CELL CALCULATION` statement with the `WITH` clause, like this:

```
WITH CELL CALCULATION [SalesQuota2009]
FOR '( [Date].[Fiscal Year].&[2009],
    [Date].[Fiscal].[Month].MEMBERS,
     [Measures].[Sales Amount Quota] )'
AS '( ParallelPeriod( [Date].[Fiscal].[Fiscal Year], 1,
     [Date].[Fiscal].CurrentMember), [Measures].[Sales Amount] ) * 2'
SELECT { [Measures].[Sales Amount Quota],
     [Measures].[Sales Amount] } ON 0,
DESCENDANTS( { [Date].[Fiscal].[Fiscal Year].&[2008],
     [Date].[Fiscal].[Fiscal Year].&[2009] }, 3, SELF ) ON 1
FROM [Adventure Works]
```

Cell calculations not only help you evaluate specific cell values, but also avoid the addition of members in the cube space. The properties `CALCULATION_PASS_NUMBER` and `CALCULATION_PASS_DEPTH` provide the functionality to specify complex recursive calculations such as goal-seeking equations. The `CALCULATION_PASS_NUMBER` specifies the `PASS` number at which the calculation is to be performed. Cell calculation is provided here for overall MDX understanding. This is deprecated by the product in favor of the assignments which you will learn in the next section.

Simple Assignments

The cell calculation syntax also allows you to model complex business logic through the `SCOPE` statement, the `CASE` statement, and MDX functions such as `Root` and `Leaves`. You must be familiar with the `SCOPE` statement along with assignments using the `THIS` keyword. Each assignment statement in MDX scripts results in a new `PASS` value. The cell calculation example with `SCOPE` is as follows:

```
SCOPE([Date].[Fiscal Year].&[2009],
     [Date].[Fiscal].[Month].MEMBERS,
     [Measures].[Sales Amount Quota]);
THIS = (ParallelPeriod( [Date].[Fiscal].[Fiscal Year],
```

```
1,[Date].[Fiscal].CurrentMember),[Measures].[Sales Amount])*2;
END SCOPE;
```

The preceding cell calculation is simple in the sense that it does not require special conditions. It is referred to as a simple assignment because the cell value for the current coordinate indicated by `THIS` is assigned a value, which is evaluated from the MDX expression on the right side.

Complex Assignments with IF Statements

If you have a complex expression with several conditions to apply to the cell calculation, a simple assignment will not be sufficient. You can use the `IF` statement to check for conditions before applying the cell calculation. For example, if you want the Sales Amount Quota to be two times the previous year's Sales Amount just for the first quarter, your calculation using the `IF` statement is:

```
SCOPE( [Date].[Fiscal Year].&[2008],
    [Date].[Fiscal].[Month].MEMBERS,
    [Measures].[Sales Amount Quota] );

    THIS = ( ParallelPeriod( [Date].[Fiscal].[Fiscal Year], 1,
        [Date].[Fiscal].CurrentMember ),
        [Measures].[Sales Amount] )  1.3;
    IF ( [Date].[Fiscal].CurrentMember.Parent IS
        [Date].[Fiscal].[Fiscal Quarter].&[2008]&[1] )
    THEN THIS = ( ParallelPeriod( [Date].[Fiscal].[Fiscal Year], 1,
        [Date].[Fiscal].CurrentMember ),
        [Measures].[Sales Amount] )  2.0
    END IF;

END SCOPE;
```

In the preceding example, you first assign values to all the cells corresponding to the subcube to be 1.3 times the value of the `Sales Amount` in the previous year. Then you use the conditional `IF` statement to update the first quarter's cell values to be two times the `Sales Amount`. As you can see, the statement is straightforward. It is quite easy to debug statements in MDX scripts with the help of the MDX debugger within the Cube Designer. (You learn how to debug MDX scripts in Chapter 9.)

Complex Assignments with CASE Statements

You can get into more complex expressions that may require multiple `IF` statements, which can lead to updating cells multiple times. To support efficient assignments, Analysis Services provides the `CASE` statement. The syntax for the `CASE` statement is:

```
CASE <value_expression>
WHEN <value_expression> THEN <statement>
ELSE  <statement>
END;
```

Here an MDX statement is assigned one of the values specified by the `CASE` statement. You can specify the conditions easily, as in the following example:

```
SCOPE ( [Date].[Fiscal Year].&[2008],
    [Date].[Fiscal].[Month].MEMBERS,
    [Measures].[Sales Amount Quota] );

    THIS = CASE

    WHEN ( [Date].[Fiscal].CurrentMember.Parent IS
        [Date].[Fiscal].[Fiscal Quarter].&[2008]&[1] )
    THEN ( ParallelPeriod( [Date].[Fiscal].[Fiscal Year], 1,
        [Date].[Fiscal].CurrentMember ),
        [Measures].[Sales Amount] )  1.3

    WHEN ( [Date].[Fiscal].CurrentMember.Parent IS
        [Date].[Fiscal].[Fiscal Quarter].&[2008]&[2] )
    THEN ( ParallelPeriod( [Date].[Fiscal].[Fiscal Year], 1,
        [Date].[Fiscal].CurrentMember ),
        [Measures].[Sales Amount] )  2.0

    ELSE ( ParallelPeriod( [Date].[Fiscal].[Fiscal Year], 1,
        [Date].[Fiscal].CurrentMember ),
        [Measures].[Sales Amount] ) * 1.75
```

```
    END;

END SCOPE;
```

Direct Assignments to a Subcube

In all the previous examples, you have seen `SCOPE – END SCOPE` being used. Use `SCOPE` when you have multiple calculations that need to be applied within the `SCOPE`. However, if it is a single MDX expression, you can write the cell calculation by direct assignment to the subcube, as shown here:

```
( [Date].[Fiscal Year].&[2009],
    [Date].[Fiscal].[Month].MEMBERS,
    [Measures].[Sales Amount Quota] ) =
( ParallelPeriod( [Date].[Fiscal].[Fiscal Year], 1,
    [Date].[Fiscal].CurrentMember ),
    [Measures].[Sales Amount] ) * 2;
```

Root and Leaves Assignments

Two additional MDX functions can help you write cell calculations with ease. These functions are `Root` and `Leaves`, which are great if you want to apply cell calculation to the leaf-level members or the root of a hierarchy. These functions appropriately position a coordinate so that the cell calculations can be applied to that coordinate. The following example uses the `Root` and `Leaves` MDX functions:

```
CREATE MEMBER CURRENTCUBE.[Measures].[Ratio To All Products] AS
    [Measures].[Sales Amount] /
        ( Root( [Product] ), [Measures].[Sales Amount] ),
    FORMAT_STRING = "Percent",
    NON_EMPTY_BEHAVIOR = [Sales Amount];

SCOPE( Leaves( [Date] ), [Measures].[Sales Amount Quota] );
    THIS = THIS * 1.2;
END SCOPE;
```

In the first part of the example, you see a calculated measure that computes the contribution of Sales of a product as a portion of total product sales. This is accomplished through the use of `Root(Product)`, which provides the sales information for all the products. `Root(Product)` is often used to calculate ratios of a measure for a single member against all the members in the dimension. In the second part of the example, the `Sales Amount Quota` is being applied to Leaf members of the Date dimension so that the `Sales Amount Quota` is increased by 20%. The `Leaves` MDX function would help in budgeting and financial calculations where you want the calculations applied only to the leaf-level members and then rolled up to the members at other levels. The `Leaves` MDX function (like the `Root` function) takes a dimension as its argument and returns the leaf-level members of the dimension that exist with the granularity attribute of the dimension. (You learn about the granularity attribute in Chapter 9.)

Recursion

Recursive calculations occur when a calculated member references itself for calculations. For example, if you want to calculate the cumulative sales over time, you can apply an MDX expression that calculates the sales of the current time member and the cumulative sales for the previous time member. This leads to recursion because the cumulative sales of the previous time member needs to be evaluated with the same MDX expression.

You can avoid infinite recursions by the use of a `PASS` value. Consider the following MDX statements in your MDX script:

```
SCOPE ([Date].[Fiscal Year].&[2004]);
    [Sales Amount Quota] = [Sales Amount Quota] * 1.2;
END SCOPE;
```

The evaluation of `[Sales Amount Quota]` would result in an infinite recursion, but Analysis Services can automatically handle these types of scenarios by virtue of internally assigning a value from the previous calculation pass.

Freeze Statement

Use the `Freeze` statement in circumstances in which you might want to change the cell value that was used in an MDX expression to determine results for another cell value without changing the cell value from an earlier calculation. This `Freeze` statement is used only within MDX scripts. The syntax is:

```
Freeze <subcube expression>
```

For more information on the `Freeze` statement including detailed examples and how to use it effectively, we recommend *MDX Solutions*: *With SQL Server Analysis Services 2005 and Hyperion Essbase*, *Second Edition*, by George Spofford *et al.* (Wiley Publishing,

Inc., 2006).

Restricting Cube Space/Slicing Cube Data

Considering that a typical cube contains several dimensions with possibly hundreds or even thousands of members, you should want to restrict the cube space for your queries by slicing the data or drilling down into specific sections. You can do this in several ways using MDX, the choice of which would depend on the context of your problem and what you want to accomplish.

SCOPE Statement

The SCOPE statement is used within MDX scripts to restrict the cube space so that all MDX statements and expressions specified within the SCOPE statement are evaluated exactly once against the restricted cube space. The syntax of the SCOPE statement is:

```
SCOPE <SubCubeExpression>
 <MDX Statement>
 <MDX Statement> ...
END SCOPE
```

You can have one or more MDX statements within the SCOPE statement, and you can have nested SCOPE statements. Nested SCOPE statements can often be simplified as a single SCOPE statement as long as all the MDX statements are within the innermost SCOPE statement. MDX statements expressed within SCOPE statements are actually cell calculations, which you learned earlier in this chapter. Named sets in the MDX script are not affected by the SCOPE statement.

CREATE and DROP SUBCUBE

The CREATE SUBCUBE statement enables you to restrict the cube space for subsequent queries. This statement is typically used within the scope of a query session. The syntax of this statement is:

```
CREATE SUBCUBE <SubCubeName> AS <SELECT Statement>
```

where the SELECT statement is an MDX SELECT clause that returns the results for the restricted cube space based on specific criteria. The cube name specified in the SELECT statement should have the same *SubCubeName* specified in the CREATE SUBCUBE statement.

Assume you are analyzing Internet sales in the Adventure Works DW Multidimensional database for various quarters by using the following MDX query:

```
SELECT [Measures].[Internet Sales Amount] ON 0,
[Date].[Fiscal].[Fiscal Quarter].MEMBERS ON 1
FROM [Adventure Works]
```

If you want to restrict the cube space and analyze only the Internet sales data for the fiscal year 2008, you can use a CREATE SUBCUBE statement, as follows:

```
CREATE SUBCUBE [Adventure Works] AS
SELECT { [Date].[Fiscal].[Fiscal Year].&[2008] } ON 0
FROM [Adventure Works]

SELECT [Measures].[Internet Sales Amount] ON 0,
[Date].[Fiscal].[Fiscal Quarter].MEMBERS ON 1
FROM [Adventure Works]

DROP SUBCUBE [Adventure Works]
```

Notice the DROP SUBCUBE statement followed by the name of the subcube after the query to revert back to the original cube space.

Using EXISTS

As mentioned earlier, the cube space is typically quite large and sparse. Remember AUTO EXISTS? Well, EXISTS is a function that allows you to explicitly do the same operation of returning a set of members that exists with one or more tuples of one or more sets. The EXISTS function can take two or three arguments according to the following syntax:

```
EXISTS( Set, <FilterSet>, [MeasureGroupName])
```

The first two arguments are sets that get evaluated to identify the members that exist with each other. EXISTS identifies all the members in the first set that exist with the members in the *FilterSet* and returns those members as results. The third optional parameter is the Measure group name, so EXISTS can be applied across the measure group.

Using EXISTING

By now, you should be quite familiar with the WHERE clause in the MDX SELECT statement. The WHERE clause changes the default members of the dimensions for the current subcube but does not restrict the cube space. It does not change the default for the outer query and gets a lower precedence as compared to the calculations specified within the query scope.

To restrict the cube space so that calculations are performed within the scope of the conditions specified in the WHERE clause, you can use the keyword EXISTING, by which you force the calculations to be done on a subcube under consideration by the query rather than the entire cube. Following is an MDX query using EXISTING:

```
WITH MEMBER [Measures].[X] AS
COUNT ( EXISTING [Customer].[Customer Geography].[State-Province].MEMBERS)
SELECT [Measures].[X] ON 0
FROM [Adventure Works]
WHERE ( [Customer].[Customer Geography].[Country].&[United States] )
```

If you execute the above query against the sample Adventure Works DW Multidimensional database you will get a result of 36. If you remove EXISTING and execute the query you will get the result 71. The first MDX query returns the states associated with the country United States and the later returns the total number states in the databases.

Using SUBSELECT

MDX queries can contain a clause called subselect, which allows you to restrict your query to a subcube instead of the entire cube. The syntax of the subselect clause along with SELECT is:

```
[WITH <formula_expression> [, <formula_expression> ...]]
SELECT [<axis_expression>, <axis_expression>...]]

FROM [<cube_expression> | (<sub_select_statement>)]
[WHERE <expression>]
[[CELL] PROPERTIES <cellprop> [, <cellprop> ...]]

<sub_select_statement> =
SELECT [<axis_expression> [, <axis_expression> ...]]
FROM [<cube_expression> | (< sub_select_statement >)]
[WHERE <expression>]
```

The *cube_expression* in the MDX SELECT statement can now be replaced by another SELECT statement called the sub_select_statement, which queries a part of the cube. You can have nested sub_select_statements up to any level. The subselect clause in the SELECT statement restricts the cube space to the specified dimension members in the subselect clause. Outer queries can therefore see only the dimension members specified in the inner subselect clauses. Look at the following MDX query that uses subselect syntax:

```
SELECT NON EMPTY { [Measures].[Internet Sales Amount] } ON COLUMNS,
NON EMPTY { ([Customer].[Customer Geography].[Country].ALLMEMBERS ) }
DIMENSION PROPERTIES MEMBER_CAPTION, MEMBER_UNIQUE_NAME ON ROWS
FROM (
   SELECT ( { [Date].[Fiscal].[Fiscal Year].&[2008],
              [Date].[Fiscal].[Fiscal Year].&[2009]
            }
          )
   ON COLUMNS
   FROM (
         SELECT ( { [Product].[Product Categories].[Subcategory].&[26],
                    [Product].[Product Categories].[Subcategory].&[27] } )
         ON COLUMNS
         FROM [Adventure Works]
         )
      )
WHERE ( [Product].[Product Categories].CurrentMember,
        [Date].[Fiscal].CurrentMember
      )
CELL PROPERTIES VALUE, BACK_COLOR, FORE_COLOR, FORMATTED_VALUE
```

The query contains two subselect clauses. The innermost clause returns a subcube that contains only Products whose SubCategory ids are 26 or 27. Assume that this subcube is named subcube A. The second subselect uses subcube A and returns another subcube with the restriction of Fiscal Years 2008 and 2009. Finally, the outermost SELECT statement retrieves the Internet Sales Amount for Customers in various countries. Here, subselect clauses restrict the cube space to certain members on Product and Date dimensions and

thereby the outermost `SELECT` statement queries data from a subcube rather than the entire cube space. If you execute the preceding query in SSMS, you can see the results. You can rewrite most queries using subselect clauses with the `WHERE` clause in Analysis Services 2012, which accept sets as valid MDX expressions. There are instances in which subselects and `WHERE` clauses can return different results. More information is provided in the book *MDX Solutions*: *With SQL Server Analysis Services 2005 and Hyperion Essbase*, *Second Edition*, by George Spofford *et al.* (Wiley Publishing, Inc., 2006). Analysis Services 2012 supports calculated members within subselects.

Parameterized MDX Queries

Parameterized queries in MDX, as the name suggests, help in passing parameters to a query where the values for the parameters are substituted before query execution. Why are parameterized queries important? You might have heard about injection attacks on web sites in which an attacker hacks the sites by entering their own raw inputs that are executed along with the full query. One of the main reasons why such attacks are possible is because user input is not validated, but Analysis Services overcomes MDX injection threats by allowing parameters to be passed along with queries. Analysis Services validates these parameters, replaces the parameters in the query with the actual values, and then executes the query.

The parameters to a query are represented within the query prefixed with the `@` symbol. The following is an example that uses a parameter for filtering the country.

```
select [Measures].members on 0,
       Filter(Customer.[Customer Geography].Country.members,
              Customer.[Customer Geography].CurrentMember.Name =
              @CountryName) on 1
from [Adventure Works]
```

Your client application would send the preceding query along with the list of parameters and values. The following XMLA script shows how this is sent to Analysis Services. You have a name and value pair specified for each parameter under the Parameters section of the XMLA script.

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <Execute xmlns="urn:schemas-microsoft-com:xml-analysis">
      <Command>
        <Statement>
select [Measures].members on 0,
       Filter(Customer.[Customer Geography].Country.members,
              Customer.[Customer Geography].CurrentMember.Name =
              @CountryName) on 1
from [Adventure Works]
</Statement>
      </Command>
      <Properties >
      <Parameters>
        <Parameter>
          <Name>CountryName<Name>
          <Value>'United Kingdom'</Value>
        </Parameter>
      </Parameters>
    </Execute>
  </Body>
</Envelope>
```

MDX Comments

MDX can quickly end up with a complex query or expression. Therefore, you should add comments to your MDX expressions and queries so that you can look back at a later point in time and interpret or understand what you were implementing with a specific MDX expression or query.

You can comment your MDX queries and expressions in three different ways:

```
// (two forward slashes) comment goes here
-- (two hyphens) comment goes here
/* comment goes here */ (slash-asterisk pairs)
```

Summary

Congratulations, you have made it through the first three chapters! Ostensibly, you should now feel ready to take on the rest of the chapters in no particular order. But you got this far, so why not go immediately to Chapter 4 and jump right in? Now you know the fundamental elements of MDX: cells, members, tuples, and sets. Further, you learned that MDX has two forms: queries and expressions. You also learned about calculation fundamentals and the use of MDX scripts to apply global scope calculations.

You saw that MDX queries retrieve data from Analysis Services databases. MDX expressions, on the other hand, are simple yet powerful constructs that are partial statements; by themselves they do not return results like queries. The expressions are what enable you to define and manipulate multidimensional objects and data through calculations, like specifying a default member's contents, for example.

To solidify your understanding of MDX, you learned the common query statements `WITH`, `SELECT`, `FROM`, and `WHERE`, as well as the MDX operators such as addition, subtraction, multiplication, division, set and the logical operators AND and OR. These details are crucial for effective use of the language. You got a good look at the MDX function categories, saw the four forms MDX functions can take, and even saw detailed examples of some commonly used functions. All the MDX functions supported in Analysis Services are provided with examples in Appendix A, available online at www.wrox.com.

Don't think you are done learning about MDX with this chapter. You learn additional MDX in subsequent chapters through illustrations and examples wherever applicable. Your journey through the Analysis Services landscape will become even more interesting and exciting as you work your way through the book. Coming up in Chapter 4 are the details to create a data source and a Data Source View, and how to deal with multiple data source views in a single project.