This first chapter introduces the syntax and semantics of the MDX (MultiDimensional eXpressions) language, looking at basic queries and the language’s modular nature. We will assume that you have a basic understanding of the multidimensional structures and metadata supported by the server(s) that you work with, but we won’t assume that you’ve ever seen MDX before. This chapter introduces most of the major aspects of an MDX query and builds an important foundation for the subsequent chapters. It also introduces you to many important parts of the language. This material may bear reading slowly and perhaps more than once if you are new to the language.

**What Is MDX?**

MDX is a language that expresses selections, calculations, and some metadata definitions against an Online Analytical Processing (OLAP) database, and provides some capabilities for specifying how query results are to be represented. Unlike some other OLAP languages, it is not a full report-formatting language, though. The results of an MDX query come back to a client program as data structures that must be processed in some way to look like a spreadsheet, a
chart, or some other form of output. This is quite similar to how SQL works with relational databases and how its related application programming interfaces (APIs) behave. As of this writing, there are several different APIs that support MDX, including Object Linking and Embedding Data Base for Online Analytical Processing (OLE DB for OLAP), ADO MD, ADOMD.Net, XMLA (XML for Analysis), the Hyperion Essbase C and Java APIs, and the Hyperion ADM API.

The specification for OLE DB for OLAP describes the full relationship between MDX queries and the data structures that convey the queried information back to the client program. This chapter focuses more on the more logic-related side—what queries are asking for—rather than the programming-oriented aspect of how the results are returned.

**Query Basics**

We will start by looking at simple MDX queries and build on them in small steps. Throughout this chapter, we will mix descriptions of the abstract properties of a query in with concrete examples to build up a more comprehensive picture of MDX.

Imagine a very simple cube, with dimensions of time, sales geography, and measures. The cube is called Sales. Let’s say we want to look at a grid of numbers that has the Massachusetts sales and costs for the first two quarters of 2005. MDX queries result in grids of cells holding data values. The grid can have two dimensions, like a spreadsheet or table, but it can also have one, three, or more. (It can also have zero; we’ll talk about that in the section “Queries.”) The grid we want to see is shown in Figure 1-1.

The MDX query in the following example would specify the cells we want to see in Figure 1-1:

```mdx
SELECT
  { [Measures].[Dollar Sales], [Measures].[Unit Sales] }
on columns,
  { [Time].[Q1, 2005], [Time].[Q2, 2005] }
on rows
FROM [Sales]
WHERE ([Customer].[MA])
```

<table>
<thead>
<tr>
<th></th>
<th>Dollar Sales</th>
<th>Unit Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1, 2005</td>
<td>96,949.10</td>
<td>3,866</td>
</tr>
<tr>
<td>Q2, 2005</td>
<td>104,510.20</td>
<td>4,125</td>
</tr>
</tbody>
</table>

*Figure 1-1*  A very simple data grid to retrieve.
In the query, `SELECT`, `FROM`, and `WHERE` are keywords that denote different parts of the query. The result of an MDX query is itself a grid, essentially another cube. You lay out the dimensions of the cube you are querying onto the axes of the result; this query refers to two of them by the names `rows` and `columns`. In MDX terminology, an `axis` is an edge or dimension of the query result. Referring to “axis” rather than “dimension” makes it simpler to distinguish the dimensions of the cube being queried from the dimensions of the results. Furthermore, each axis can hold a combination of multiple cube dimensions.

You may guess some generalizations immediately from this example. Let’s break this simple query down into pieces:

1. The `SELECT` keyword starts the clause that specifies what you want to retrieve.
2. The `ON` keyword is used with an axis name to specify where dimensions from your database are displayed. This example query puts measures on the columns axis and time periods on the row axis.
3. MDX uses curly braces, `{ and }`, to enclose a set of elements from a particular dimension or set of dimensions. Our simple query has only one dimension on each of the two axes of the query (the measures dimension and the time dimension). We can separate different elements with commas (,). Element names may be enclosed by `[]` characters, and may have multiple parts separated by dot (.) characters.
4. In an MDX query, you specify how dimensions from your database lay out onto axes of your result. This query lays out measures on columns and time periods on rows. Each query may have a different number of result axes. The first three axes have the names “columns,” “rows,” and “pages” to conceptually match a typical printed report. (You can refer to them in other ways, as you will see in the “Axis Numbering and Ordering” section.) Although this simple query does not show more than one dimension on a result axis, when more than one dimension maps to a result axis, each cell slot on the axis is related to a combination of one member from each of the mapped dimensions.
5. The `FROM` clause in an MDX query names the cube from which the data is being queried. This is similar to the `FROM` clause of Structured Query Language (SQL) that specifies the tables from which data is being queried.
6. The `WHERE` clause provides a place to specify members for other cube dimensions that don’t appear in the columns or rows (or other axes). If you don’t specify a member for some dimension, then MDX will make use of some default. (We will ignore the parentheses in this query until later on when we discuss tuples.) The use of `WHERE` is optional.
Once the database has determined the cells of the query result, it fills them with data from the cube being queried.

MDX shares the keywords SELECT, FROM, and WHERE with SQL. Based on substantial experience teaching MDX, please take this advice: If you are familiar with SQL and its use of SELECT, FROM, and WHERE, do your best to forget about SQL when learning MDX. The meanings and semantics are quite different, and trying to apply SQL concepts to MDX will most likely create confusion.

Let’s look at another example. As far as MDX is concerned, every dimension is like every other dimension, even the measures dimension. To generate the result grid shown in Figure 1-2, with sales shown over all three quarters across the columns and for two states down the rows, we can use the following query:

```
SELECT
  { [Time].[Q1, 2005], [Time].[Q2, 2005], [Time].[Q3, 2005] }
on columns,
  { [Customer].[MA], [Customer].[CT] }
on rows
FROM Sales
WHERE ( [Measures].[Dollar Sales] )
```

As you can see, the orientation of time, location, and measures has been switched just by listing time on columns, customers on rows, and measures in the WHERE section.

MDX has a number of other relevant aspects that the remainder of this chapter will describe:

- The MDX data model
- Simple MDX query construction
- Parts of MDX queries

You can skip ahead to the section “Simple MDX Construction,” later in this chapter, if you want to jump right into learning more MDX vocabulary. In the discussion of the first two topics—the MDX data model and how metadata entities are named—we will present a lot of detail that may seem mundane or boring. However, the details are important for almost every query, and gaining a deep understanding of MDX demands mastery of the basics. If you do jump ahead and later find yourself getting lost in the syntax, come back to this chapter and get a refresher on the basics before moving on to the next level.

<table>
<thead>
<tr>
<th></th>
<th>Q1, 2005</th>
<th>Q2, 2005</th>
<th>Q3, 2005</th>
</tr>
</thead>
<tbody>
<tr>
<td>MA</td>
<td>96,949.10</td>
<td>104,510.20</td>
<td>91,025.00</td>
</tr>
<tr>
<td>CT</td>
<td>12,688.40</td>
<td>24,660.70</td>
<td>16,643.90</td>
</tr>
</tbody>
</table>

**Figure 1-2** A second simple result grid.
Axis Framework: Names and Numbering

You signify that you are putting members onto columns or rows or other axes of a query result by using the “on columns/rows/pages/etc” syntax. The axis designations can come in any order; the last query would have been just as valid and would mean exactly the same if it had been phrased as follows:

```sql
SELECT
 { [Customer].[MA], [Customer].[CT] }
on rows,
{ [Time].[Q1, 2005], [Time].[Q2, 2005], [Time].[Q3, 2005] }
on columns
FROM Sales
WHERE ( [Measures].[Dollar Sales] )
```

You can also use axis numbers to specify the axis in the query, by stating, for example:

```sql
{ [Time].[Q1, 2005], [Time].[Q2, 2005], [Time].[Q3, 2005] }
on axis(0),
{ [Customer].[MA], [Customer].[CT] }
on axis(1)
```

The phrase `on axis(n)` indicates that those members should be put on the axis numbered `n`. The names used so far are synonyms for these numbers:

<table>
<thead>
<tr>
<th></th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Columns</td>
</tr>
<tr>
<td>1</td>
<td>Rows</td>
</tr>
<tr>
<td>2</td>
<td>Pages</td>
</tr>
<tr>
<td>3</td>
<td>Chapters</td>
</tr>
<tr>
<td>4</td>
<td>Sections</td>
</tr>
</tbody>
</table>

For axes beyond `axis(4)`, you must use the numbers, because there are no names for them. You can also mix and match names and numbers in a query:

```sql
SELECT
 { [Customer].[MA], [Customer].[CT] }
on rows,
{ [Time].[Q1, 2005], [Time].[Q2, 2005], [Time].[Q3, 2005] }
on axis(0)
FROM Sales
WHERE ( [Measures].[Dollar Sales] )
```

However, a query that uses axis 1 must also use axis 0, and a query that uses axis 2 must also use axes 1 and 0. You cannot skip an axis in a query—you’ll get an error. The following won’t work because it skips axis 1:
SELECT
  {{[Customer].[MA], [Customer].[CT] }
on axis(2),
  {{ [Time].[Q1, 2005], [Time].[Q2, 2005], [Time].[Q3, 2005] }
on axis(0)
FROM Sales
WHERE ([Measures].[Dollar Sales])

If you think of the result of an MDX query as a hypercube (or subcube), and each axis as a dimension of that hypercube, then you can see that skipping an axis would be problematic.

**Case Sensitivity and Layout**

MDX is neither case-sensitive nor line-oriented. For example, both of the following fragments are equally valid:

```mdx
SELECT
  {{ [Time].[Q1, 2005], [Time].[Q2, 2005], [Time].[Q3, 2005] }
on COLUMNS, ...
```

```mdx
select
  {{ [Time].[Q1, 2005], [Time].[Q2, 2005], [Time].[Q3, 2005] }
on columns, ...
```

You can mix capitalization of letters as you want. For language keywords, capitalization doesn’t matter. For names, capitalization may or may not matter to the MDX provider—it usually doesn’t for Analysis Services or Essbase, although there are some cases that we will cover later where Essbase does care about capitalization. It also doesn’t matter where you put spaces and line ends. You can rewrite the prior fragments as the following:

```mdx
SELECT {
  [Time].[Q1, 2005]
, [Time].[Q2, 2005]
, [Time].[Q3, 2005]
} 
on COLUMNS,
```

This gives you some flexibility when you are generating MDX by code as well as when you want to arrange it for readability. We strongly recommend using plenty of indenting when writing MDX queries, for both you and anyone else who has to understand your work.
In this book, we will frequently use white space and an outline-like style in MDX examples. Our experience is that this helps make code more readable and much easier to edit, debug, and evolve. This follows the same principles as development in any other programming language. In order to conserve page space, however, we will sometimes compress the examples a bit more than we would prefer.

Additionally, MDX makes heavy use of parentheses, brackets, and braces. These can look fairly similar in common fonts like Arial. In whatever environment you write MDX, if you can use a monospace font such as Lucida Console or Courier, you will have an easier time reviewing the MDX you have written.

Simple MDX Construction

Let’s build up a little vocabulary and look at some of the very simplest and most frequently used operators and functions. We will introduce them here and describe how they are often used. (More complete and detailed descriptions can be found in Appendix A, which is a reference on MDX functions and operators.) The ones we will look at here are

- Comma (,) and colon (:)
- .Members
- .Children and Descendants()

We will use the term set when discussing these functions and operators. A set is a rather important concept in MDX, and will be described in detail in the section “The MDX Data Model: Tuples and Sets.” Until then, we will be a little informal with our use.

Comma (,) and Colon (:)

We have already seen the comma operator used to construct sets; let’s talk about this more here. We can assemble a set by enumerating its components and separating them with commas, as in:

```mdx
{ [Time].[January 2005], [Time].[February 2005],
  [Time].[March 2005] }
```

This expression results in a set that holds the first three months of 2005.
At every level in every dimension, the members of that level are arranged in a particular order (usually by member key or by name). When it makes sense, you can specify a set as a range of members in that order by listing two members from the same level as endpoints and putting a colon between them to mean “These members and every member between them.” (This is similar to the syntax used to specify ranges of cells in spreadsheets like Excel.) For example, the following query, whose results are shown in Figure 1-3, selects the months from September 2004 through March 2005 and the product categories from Tools through Home Audio for customers in Lubbock, and Unit Sales:

```sql
SELECT
  { [Time].[Sep,2004] : [Time].[Mar,2005] } on columns,
  { [Product].[Tools] : [Product].[Home Audio] } on rows
FROM [Sales]
WHERE ([Customer].[Lubbock, TX], [Measures].[Unit Sales])
```

Sets are usually built like this when the database-defined ordering corresponds to a useful real-world ordering, such as with time. The colon takes two members on the same level as its endpoints. It is not an error to put the same member on both sides of the colon; you will just get a range of one member.

The comma operator can be used anywhere within a set specification to add subsets to an overall set. For example, the following creates a set of the first three and last three months of 2001:

```sql
```

whereas the following creates a set of 2001 and its first three months:

```sql
```
When subsets are concatenated by commas, the order in which the commas join them is the order in which they are returned.

**PRODUCT DIFFERENCES**

**ANALYSIS SERVICES**

In Analysis Services, regardless of whether the member on the left of the colon is before or after the member on the right in terms of database ordering, the set that results will have all members between the two in database ordering. That is, the following two return exactly the same set:

```
```

**ESSBASE**

In Essbase, the set starts with the left member and runs through the right member. The following returns the ordering April, March, February, January:

```
```

The members must also be at the same level; you can use the `MemberRange()` function to get a range from a common generation.

**.Members**

Getting the set of members for a dimension, hierarchy, or level is very common both for retrievals and as a starting point for further operations. The `.Members` operator takes a dimension, hierarchy, or level on its left-hand side, and returns a set of all members associated with that metadata scope. For example, `[Customer].Members` results in the set of all customers, whereas `[Product].[Product Category].Members` returns all members of the Product Category level in the Product dimension. For example, the query
SELECT
  { [Scenario].Members } on columns,
  { [Store].Members } on rows
FROM Budgeting

lays out all members of the scenario dimension across the columns and all members of the store dimension down the rows.

When a client uses .Members (or other metadata functions that return the set of members associated with some metadata element), neither Analysis Services nor Essbase will include any calculated members in the returned set. (Calculated members are a means for introducing calculations onto other data in the database, and are discussed starting in Chapter 2.) This means that the preceding request for [Scenario].Members, as written, will not return any calculated members in the scenario dimension. We can always ask for them by name, however, and Analysis Services provides the AddCalculatedMembers() and .AllMembers functions to add them into a set. See Appendix A for details on these functions.

**Getting the Children of a Member with .Children**

Another kind of selection that is very frequent is to get the children of a member. We may want to do this to implement a drill-down operation, or simply to conveniently obtain a range of members based on a common parent. MDX provides a .Children function that will do this for us. The following query selects both the [Product].[Tools] member and its children on the rows of a report, which is shown in Figure 1-4:

```
SELECT
  ( [Time].[Q3, 2005].Children )
on columns,
  ( [Product].[Tools], [Product].[Tools].Children )
on rows
FROM Sales
WHERE ([Customer].[TX], [Measures].[Unit Sales])
```

You can get the children of any member that has children using this function. If you request the children of a leaf-level member, you’ll get an empty set of members back. We’ll talk more about what this means later on.
PRODUCT-SPECIFIC ASPECTS OF .MEMBERS

ANALYSIS SERVICES 2000 AND 2005
In Analysis Services, hierarchies behave like dimensions, and multiple dimensions have different sets of members. As a result, in an Analysis Services dimension that contains multiple hierarchies, you cannot simply request [Dimension].Members. If you do, it will complain of an unknown dimension. For example, given a logical [Time] dimension that contains two hierarchies, [Time].[Fiscal] and [Time].[Calendar], a client taking metadata from OLE DB for OLAP will see one time dimension. However, the expression [Time].Members will result in an error rather than result in all members on all time hierarchies. To obtain a set of members, the client must request either [Time].[Fiscal].Members or [Time].[Calendar].Members. If the dimension has only one hierarchy and that hierarchy does not have an explicit name, then [Dimension].Members will work. For example, if time has only one hierarchy, then [Time].Members will work.

ESSBASE
In Essbase 9, when you request all members of some scope with .Members, all copies of shared members are returned in the result set. There is no way to strip shared members from a set. Therefore, you may need to build awareness of where shared members exist into your MDX, for example using Descendants() (described later) instead of .Members.

<table>
<thead>
<tr>
<th>Tools</th>
<th>July, 2005</th>
<th>Aug, 2005</th>
<th>Sep, 2005</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bench Power Tools</td>
<td>42</td>
<td>133</td>
<td>94</td>
</tr>
<tr>
<td>Compressors, Air Tools</td>
<td></td>
<td></td>
<td>51</td>
</tr>
<tr>
<td>Electrical Shop</td>
<td>107</td>
<td>118</td>
<td>33</td>
</tr>
<tr>
<td>Garage Door Openers</td>
<td>57</td>
<td>46</td>
<td>53</td>
</tr>
<tr>
<td>Hand Tools, Carpentry</td>
<td></td>
<td></td>
<td>55</td>
</tr>
<tr>
<td>Hand Tools, General Purpo</td>
<td>138</td>
<td>164</td>
<td>31</td>
</tr>
<tr>
<td>Mechanics Tools</td>
<td>88</td>
<td>38</td>
<td>57</td>
</tr>
<tr>
<td>Portable Power Tools</td>
<td>52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power Tool Accessories</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1-4 Result of query using .Children.

Getting the Descendants of a Member with Descendants()
To request members further away than immediate children, or for a more generalized search below in the hierarchy, you use Descendants(). We will skim the surface of this function because it has too many options to go into here; you can refer to a complete reference for it in Appendix A.
Syntax:

Descendants (member [, [ level ] [, flag]] )

Descendants() returns the members below member related to the level or generation, with a number of selection options based on the optional flag. The choices for flag are

SELF
BEFORE
AFTER
SELF_AND_BEFORE
SELF_AND_AFTER
SELF_BEFORE_AFTER
LEAVES

SELF refers just to the members at level, and this is the most frequent option. For example, the following selects the months of 2005, which is shown in Figure 1-5:

SELECT
( [Product].[Tools], [Product].[Toys] ) ON COLUMNS,
Descendants ( [Time].[2005],
[Time].[Month],
SELF
)
ON ROWS
FROM Sales
WHERE [Measures].[Dollar Sales]

Since SELF is so frequently used, it is the default option if you omit the flag from your code. For example, Descendants ([Time].[2005], [Time].[Month]) will also return the list of months in 2005 even though SELF is not explicitly included in the query.

The other flags that include SELF refer to levels around the level you reference. SELF_AND_BEFORE means to return all members between the SELF and “before” that level, which means all of them up to and including the starting member. For example, the following picks out [2005] and all quarters and months within it, and the results are shown in Figure 1-6:
SELECT
{ [Product].[Tools], [Product].[Toys] } ON COLUMNS,
Descendants {
    [Time].[2005],
    [Time].[Month],
    SELF_AND_BEFORE
}
ON ROWS
FROM Sales
WHERE [Measures].[Dollar Sales]

<table>
<thead>
<tr>
<th></th>
<th>Tools</th>
<th>Toys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan, 2005</td>
<td>59,722.40</td>
<td>49,948.20</td>
</tr>
<tr>
<td>Feb, 2005</td>
<td>65,604.10</td>
<td>42,885.40</td>
</tr>
<tr>
<td>Mar, 2005</td>
<td>57,715.50</td>
<td>56,601.70</td>
</tr>
<tr>
<td>Apr, 2005</td>
<td>64,179.90</td>
<td>51,794.40</td>
</tr>
<tr>
<td>May, 2005</td>
<td>68,152.60</td>
<td>62,135.70</td>
</tr>
<tr>
<td>Jun, 2005</td>
<td>67,476.70</td>
<td>55,582.90</td>
</tr>
<tr>
<td>Jul, 2005</td>
<td>71,997.90</td>
<td>50,111.80</td>
</tr>
<tr>
<td>Aug, 2005</td>
<td>71,411.90</td>
<td>48,965.30</td>
</tr>
<tr>
<td>Sep, 2005</td>
<td>58,979.60</td>
<td>52,532.90</td>
</tr>
<tr>
<td>Oct, 2005</td>
<td>77,720.10</td>
<td>58,969.60</td>
</tr>
<tr>
<td>Nov, 2005</td>
<td>196,946.70</td>
<td>147,854.50</td>
</tr>
<tr>
<td>Dec, 2005</td>
<td>223,948.60</td>
<td>171,600.20</td>
</tr>
</tbody>
</table>

**Figure 1-5** Result of using Descendants(, , SELF).

<table>
<thead>
<tr>
<th></th>
<th>Tools</th>
<th>Toys</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>1,083,855.90</td>
<td>848,982.70</td>
</tr>
<tr>
<td>Q1, 2005</td>
<td>183,042.90</td>
<td>149,435.30</td>
</tr>
<tr>
<td>Jan, 2005</td>
<td>59,722.40</td>
<td>49,948.20</td>
</tr>
<tr>
<td>Feb, 2005</td>
<td>65,604.10</td>
<td>42,885.40</td>
</tr>
<tr>
<td>Mar, 2005</td>
<td>57,715.50</td>
<td>56,601.70</td>
</tr>
<tr>
<td>Q2, 2005</td>
<td>199,809.20</td>
<td>169,513.10</td>
</tr>
<tr>
<td>Apr, 2005</td>
<td>64,179.90</td>
<td>51,794.40</td>
</tr>
<tr>
<td>May, 2005</td>
<td>68,152.60</td>
<td>62,135.70</td>
</tr>
<tr>
<td>Jun, 2005</td>
<td>67,476.70</td>
<td>55,582.90</td>
</tr>
<tr>
<td>Q3, 2005</td>
<td>202,389.30</td>
<td>151,610.00</td>
</tr>
<tr>
<td>Jul, 2005</td>
<td>71,997.90</td>
<td>50,111.80</td>
</tr>
<tr>
<td>Aug, 2005</td>
<td>71,411.90</td>
<td>48,965.30</td>
</tr>
<tr>
<td>Sep, 2005</td>
<td>58,979.60</td>
<td>52,632.90</td>
</tr>
</tbody>
</table>

**Figure 1-6** Result of using Descendants (, , SELF_AND_BEFORE).
SELF_AND_AFTER means to return all members from the level referenced down to the leaf level. SELF_Before_After means to return the entire sub-tree of the hierarchy down to the leaf level, starting with the given member. Note that this is the same as just saying Descendants ([Time].[2005]), since leaving off both the flag and the level means to take all descendants.

Note also that you can refer to descendants relative to a specific level, or you can refer to descendants relative to some depth away from the given member by using a number instead of a reference to a level. There are several more useful options; once again, you might be well served by looking at the reference in Appendix A.

**NOTE** In Essbase, you can refer to either generations or levels in the level argument.

### Removing Empty Slices from Query Results

In a multidimensional space, data is frequently sparse. For example, not all products are sold in all stores at all times to all customers, so a query for some particular set of products, time periods, and customers may return some slices along one or more axes that are entirely empty. You can be pretty sure that any large retailer won’t be selling snow shovels in Hawaii in December but will have a hard time keeping them in stock in Chicago. A user may want to see the empty slices, because the empty slice may tell him or her about a complete absence of activity, but the user may also just want the empty slices treated as irrelevant and have them removed from the report. Let’s take as an example the following query, the results of which are shown in Figure 1-7:

```plaintext
SELECT
{ [Time].[Jan,2005], [Time].[Feb,2005] } ON COLUMNS,
{ [Product].[Toys],
  [Product].[Toys].Children }
ON ROWS
FROM Sales
WHERE ([Measures].[Dollar Sales], [Customer].[TX])
```

For some reason, we haven’t sold any Dolls, Electronics, Games, or Musical toys during these two months anywhere in Texas. If we only want to see the products that we did sell within the report’s time frame, MDX provides a way to remove the entirely empty slices from the query result, by using the **NON EMPTY** keywords. You can see it in use in the following query, whose results are shown in Figure 1-8:
SELECT
{ [Time].[Jan, 2005], [Time].[Feb, 2005] }
ON COLUMNS,
NON EMPTY
{ [Product].[Toys],
   [Product].[Toys].Children
}
ON ROWS
FROM Sales
WHERE ([Measures].[Dollar Sales], [Customer].[TX])

All we need to do is to add the NON EMPTY keywords at the beginning of our request for the axis. In this case, it is after the comma that comes after ON COLUMNS, and before the { that starts our request for products. The query is evaluated and after all the data for the columns is known, the rows where at least one column has data in it are returned.

<table>
<thead>
<tr>
<th>Product</th>
<th>Jan, 2005</th>
<th>Feb, 2005</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toys</td>
<td>6,950.00</td>
<td>7,666.20</td>
</tr>
<tr>
<td>Action Figures</td>
<td>747.20</td>
<td>421.50</td>
</tr>
<tr>
<td>Arts &amp; Crafts</td>
<td>2,499.80</td>
<td>2,135.90</td>
</tr>
<tr>
<td>Cars &amp; Trucks</td>
<td></td>
<td>1,078.40</td>
</tr>
<tr>
<td>Construction</td>
<td>3,002.90</td>
<td>982.00</td>
</tr>
<tr>
<td>Dolls</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Educational</td>
<td>700.10</td>
<td>597.10</td>
</tr>
<tr>
<td>Electronics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Games</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Musical</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radio Controlled</td>
<td></td>
<td>2,451.30</td>
</tr>
</tbody>
</table>

**Figure 1-7**  Query with empty slices on rows.

<table>
<thead>
<tr>
<th>Product</th>
<th>Jan, 2005</th>
<th>Feb, 2005</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toys</td>
<td>6,950.00</td>
<td>7,666.20</td>
</tr>
<tr>
<td>Action Figures</td>
<td>747.20</td>
<td>421.50</td>
</tr>
<tr>
<td>Arts &amp; Crafts</td>
<td>2,499.80</td>
<td>2,135.90</td>
</tr>
<tr>
<td>Cars &amp; Trucks</td>
<td></td>
<td>1,078.40</td>
</tr>
<tr>
<td>Construction</td>
<td>3,002.90</td>
<td>982.00</td>
</tr>
<tr>
<td>Educational</td>
<td>700.10</td>
<td>597.10</td>
</tr>
<tr>
<td>Radio Controlled</td>
<td></td>
<td>2,451.30</td>
</tr>
</tbody>
</table>

**Figure 1-8**  Empty slices removed with NON EMPTY.
NON EMPTY works on any axis, and with any dimensions and tuples. The query can be flipped around so that you can see this:

```sql
SELECT
  { [Time].[Jan, 2005], [Time].[Feb, 2005] }
ON ROWS,
  NON EMPTY
  { [Product].[Toys],
    [Product].[Toys].Children
  } ON COLUMNS
FROM Sales
WHERE [Customer].[TX]
```

There are other functions that you can make use of to remove members with empty data from your results, but NON EMPTY is perfectly suited for this.

**NOTE**  Analysis Services 2005 introduces a HAVING clause to queries, which is similar in effect to NON EMPTY but is more flexible. It is discussed in Chapter 4.

**Comments in MDX**

To facilitate documentation, three variations of comment syntax are allowed in MDX, which should suit a variety of styles. (As with other programming, comments can also be used to exclude sections of code for debugging, too.)

The first style uses the symbols `/*` and `*/` to delimit a comment. All characters between the `/*` and the `*/` are ignored by the MDX parsing machinery. This enables comments to be placed within a line and for the comment to span lines. The following is an example of this style of comment:

```sql
SELECT /* Put products
    on columns */ [Product].Members
ON COLUMNS FROM Cube
```

The second style uses the `//` symbol to begin a comment, and the comment will extend to the end of the line. The following is an example:

```sql
SELECT // Put products on columns
  [Product].Members
ON COLUMNS FROM Cube
```
The third style is like the second style but uses a pair of dashes (—) to begin a comment. The following is an example:

```plaintext
SELECT — Put products on columns
    [Product].Members
on columns FROM Cube
```

Comments can be placed anywhere white space could be used. For example, `[Product]./* whole dimension */ Members` will work just fine. Comments can be used in queries or expressions. Make sure that you don’t use them directly inside a member name, though!

If you use comments to selectively include and exclude portions of a query or expression (for example, while debugging it), keep in mind that the `/* */` comments cannot be nested. That is,

```plaintext
/* /* comment */ */
```

is not a valid comment, whereas

```plaintext
/* /* comment */
```

is a valid comment. In the first of these two examples, the first `*/` ends the overall comment, so the second `*/` is a token that the MDX parser will try to parse (and fail on).

## The MDX Data Model: Tuples and Sets

MDX uses a data model that starts with cubes, dimensions, and members, but it is richer and somewhat more complex. Understanding this data model is the key to moving beyond the basics and unlocking powerful analyses. In this section, we will explore the key elements of this model: **tuples** and **sets**. Tuples and sets are very similar to members and dimensions, but are more generalized and more flexible.

**Tip** Understanding tuples and how to use them in the various components of MDX is sometimes the most difficult part of learning MDX. You may want to revisit this section again before proceeding with the more advanced content later in this book.
Tuples

A *tuple* is a combination of members from one or more dimensions. It is essentially a multidimensional member. A single member is a simple tuple (for example, `[Time].[Jun, 2005]`). When a tuple has more than one dimension, it has only one member from each dimension. To compose a tuple with more than one dimension, you must wrap the members in parentheses, for example:

```
([Customer].[Chicago, IL], [Time].[Jan, 2005])
```

A tuple can stand for a slice of a cube, where the cube is sliced by each member in the tuple. A tuple can also be a data object that you can manipulate on its own in MDX.

We can involve tuples that we build using this syntax directly in queries. For example, consider the following, the results of which are shown in Figure 1-9:

```
SELECT
  {  ([Time].[2005], [Measures].[Dollar Sales] ),
      ([Time].[Feb, 2005], [Measures].[Unit Sales] )
} ON COLUMNS ,
  { [Product].[Tools], [Product].[Toys] } ON ROWS
FROM [Sales]
```

Our result was asymmetric as far as the combinations of time and measures that get returned. This gives us a great deal of fine control over the combinations of members and cells that will come back from a query.

You can always put a single member within parentheses, but it’s not required if the tuple is defined by just that member. However, the following is not valid because it has two time members in it:

```
([Customer].[Chicago, IL], [Time].[Jan, 2005], [Time].[Feb, 2005])
```

Also, you can’t compose an empty tuple. () is not a valid tuple. (Analysis Services 2005 has the concept of a null member that you can specify using null, so you can create a tuple by writing (null) or (null, null), but this appears to be mostly useful when using stored procedures as discussed in Chapter 10.)

In addition to composing tuples by explicitly creating them in code, a number of functions will return tuples.

The “dimensionality” of a tuple refers to the set of dimensions whose members compose it. The order in which dimensions appear in a tuple is an important part of a tuple’s dimensionality. Any and all dimensions can be part of a tuple, including members of the measures dimension.
Although you can compose a tuple from members wrapped by ( ), you cannot use syntax to compose a tuple from tuples. That is, you can build up a tuple by saying:

```

(Time).[2004],
[Customer].[Chicago, IL],
[Product].[Tools]

```

but not:

```

(Time).[2004],

[Customer].[Chicago, IL],

[Product].[Tools]

```

(Using functions that let you compose tuples in other ways, you can build tuples from other tuples. We’ll talk about that later. But not by putting their names together like this.)

In calculations and queries, MDX identifies cells based on tuples. Conceptually, each cell value is identified by a tuple composed of one member from each dimension in the cube. (This is sort of like a spreadsheet, in which Sheet 1, Column B, Row 22 identifies a cell.) In a query, some of these members’ dimensions may be placed on rows, others on columns, others on pages, and yet others on the query slicer. However, the intersection of two or more tuples is yet another tuple, so combining them all together yields a cell in the end. The tuple ([Product].[Leather Jackets], [Time].[June-2005], [Store].[Fifth Avenue NYC], [Measures].[Dollar Sales]) may completely define a cell with a value of $13,000.

Although a tuple either refers to a combination of members, when it is used in an expression where a number or string might be used, the default behavior is to reference the value in the cell that the tuple specifies. This is a little point that makes a big difference when trying to understand what some of the MDX functions do. How these references play out is the subject of Chapter 4.

In addition to creating tuples by explicitly coding them, a few functions also return tuples. We will describe them later in this book.
Sets

A set is simply an ordered collection of tuples. A set may have more than one tuple, have only one tuple, or be empty. Unlike a mathematical set, an MDX set may contain the same tuple more than once, and ordering is significant in an MDX set. Although sets might be better called “collections” or “sequences,” we are stuck with “set” for now. Depending on the context in which a set is used, it either refers to that set of tuples or to the value(s) in the cell(s) that its tuples specify.

Syntactically, a set may be specified in a number of ways. Perhaps the most common is to list the tuples within `{}`. The following query uses two sets, where the rows use a set of one dimension and the columns a set of two dimensions:

```
SELECT
  {  ([Time].[2005], [Measures].[Dollar Sales] ),
     ([Time].[Feb, 2005], [Measures].[Unit Sales] )
} ON COLUMNS ,
{ [Product].[Tools], [Product].[Toys] } ON ROWS
FROM [Sales]
```

Whenever one or more tuples are explicitly listed, you will need to enclose them within braces. Some MDX operators and functions also return sets. The expressions that use them do not need to be enclosed in braces if the set is not being combined with more tuples, but we will usually enclose set expressions in braces for the sake of style.

Although a single member is by default a tuple of one dimension, a set that has only one tuple is not equivalent to a tuple. As far as standard MDX is concerned, the following two examples are quite different:

```
{ ([Time].[2001 Week 1], [Product].[HyperGizmos])
{ ([Time].[2001 Week 1], [Product].[HyperGizmos]) }
```

The first of these is a tuple, and the second is a set containing that tuple. You might think it reasonable that wherever a set is called for, you can use a single tuple and it will be interpreted as a set of one. Analysis Services 2005 can make use of tuples or members in some contexts that otherwise ask for a set. For other servers, you will need to wrap the tuple in curly braces as in the second sample just given. Hence, the following is a valid query only for Analysis Services 2005 (it needs braces around the tuple to be valid otherwise):

```
SELECT
  { ([Time].[Jun, 2005], [Geography].[Chicago, IL])
  WHERE ([Measures].[Dollar Costs])
  FROM [Sales]
```

SELECT
  { ([Time].[Jun, 2005], [Geography].[Chicago, IL])
  WHERE ([Measures].[Dollar Costs])
  FROM [Sales]
The following is valid across all MDX providers:

```mdx
SELECT
  '{ ([Time].[Jun, 2005], [Geography].[Chicago, IL]) }' on columns
FROM [Sales]
WHERE ([Measures].[Dollar Costs])
```

Similarly, a set that happens to contain only one tuple is still considered to be a set. To use it in a context that calls for a tuple (for example, in a `WHERE` clause), even if you have guaranteed that it only contains one tuple, you must still employ an MDX function (such as `.Item()`) that takes a tuple from a set.

A set can also be empty, both explicitly (for example, `{}`) and because a function returns a set that is empty for some reason.

Every tuple in a set must have the same dimensionality (that is, the dimensions represented and their order within each tuple). The following would result in an error, because the order of the dimensions changes:

```mdx
{ ([Time].[2005], [Measures].[Dollar Sales] ),
  ([Measures].[Unit Sales], [Time].[Feb, 2005] )
}
```

However, different sets can have whatever differing dimensionality and dimension order they need in order to serve your purposes. An empty set has no dimensionality.

In addition to creating sets by explicitly coding them, a large number of functions return sets. We will explore many of them in the following chapters.

**Queries**

An MDX query result is just another cube that is a transformation of the cube that is being queried. This is analogous to a standard SQL query result, which is essentially another table. The result cube can have one, two, three, four, or more axes (up to 128 in Analysis Services 2005, and up to 64 in Analysis Services 2000 and Essbase). It is also technically possible for a query to have zero axes, but it will still return a single-cell value. Each tuple in a result axis is essentially a member of the result cube.

As described earlier, each axis of the query result is composed of a set of tuples, each of which can have one or more dimensions. When multidimensional tuples end up on the axis of a query, the order in which the dimensions appear in the tuples affects the nesting order in the axis. The first dimension listed becomes the outermost dimension, the second becomes the next outermost, and so on. The last dimension is the innermost. For example, suppose that the following set was placed on the “rows” axis in a query:
SELECT
...
{ ([Time].[2001], [Product].[Leather Jackets]),
 ([Time].[2001], [Product].[Silk Scarves]),
 ([Time].[1997], [Product].[Leather Jackets]),
 ([Time].[1997], [Product].[Silk Scarves])
} ON ROWS
...

In this case, the expected presentation for data brought back to a client through OLE DB for OLAP or ADO is shown in Figure 1-10. Note that the layout shown in Figure 1-10 is simply conventional; your applications may do something different with the results.

**Queries with Zero Axes**

We have mentioned twice that a query may have zero axes without really describing what that means. Two examples of zero-axis queries are

```
SELECT FROM SalesCube
```

and

```
SELECT FROM SalesCube
WHERE ([Time].[2004], [Geography].[Quebec],
       [Product].[Snorkels], [Channel].[Superstores])
```

Because no members were assigned to any (non-slicer) axis in either query, the results are considered to have zero axes and by convention would be single unlabeled cells, or at least cells with no distinct row or column headers. In all APIs that currently support MDX, the slicer information is returned as part of the query result. Whether you consider the results here to be zero-dimensional depends on whether or not you choose to ignore the dimensional information conveyed in the slicer information returned with the cell.

<table>
<thead>
<tr>
<th>Year</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>Leather Jackets</td>
</tr>
<tr>
<td>2001</td>
<td>Silk Scarves</td>
</tr>
<tr>
<td>1997</td>
<td>Leather Jackets</td>
</tr>
<tr>
<td>1997</td>
<td>Silk Scarves</td>
</tr>
</tbody>
</table>

**Figure 1-10** Typical expected client data layout.
**Axis-Only Queries**

Note that all MDX queries return cells. However, many useful queries ask “What members belong in this set?” where the result that is of real interest is not cell data, but the members that are associated with cell data or member property values. A query of this form, such as “Show me the customers that make up the top 10 percent of our revenue,” will at least implicitly prepare a set of cell values as well. (It may not actually retrieve their values due to internal optimizations, though.)

This is in contrast to SQL, which will return only the columns that you request. There are uses for queries that have no interest in cells; we will look at an example of this in Chapter 7.

**More Basic Vocabulary**

Now that we have introduced tuples and sets and the earlier bits of vocabulary, we can introduce three more functions that are extremely common in the MDX you will write:

- CrossJoin()
- Filter()
- Order()

**CrossJoin()**

In many cases, you will want to take the cross-product of members (or tuples) in two different sets (that is, specify all of their possible combinations). The CrossJoin() function is the most direct way of combining the two sets in this way.

The syntax is as follows:

```
CrossJoin (set1, set2)
```

For example, you may want to lay out on the columns of a query the first two quarters of 2005 and the two measures Dollar Sales and Unit Sales. You would generate this set with the following expression:

```sql
CrossJoin {
    [Time].[Q1, 2005], [Time].[Q2, 2005]
  },
  { [Measures].[Dollar Sales], [Measures].[Unit Sales] }
```
You would use it in this way; the results are shown in Figure 1-11:

```
SELECT
    CrossJoin (
        { [Time].[Q1, 2005], [Time].[Q2, 2005]},
        { [Measures].[Dollar Sales], [Measures].[Unit Sales] }
    )
ON COLUMNS ,
    { [Product].[Tools], [Product].[Toys] } ON ROWS
FROM Sales
```

CrossJoin() only takes two sets as inputs. If you want to take the CrossJoin() of three or more sets, such as times, scenarios, and products, you can do it by nesting calls to CrossJoin(), like the following:

```
CrossJoin ( 
    [Time].Members,
    CrossJoin ( 
        [Scenario].Members,
        [Product].Members
    )
)
CrossJoin ( 
    CrossJoin(
        [Time].Members,
        [Scenario].Members
    ),
    [Product].Members
)
```

Notice that each of these results in a set whose dimensionality is, in order: time, scenario, product. While you may have a personal preference for one method or another, when the sets are large, you may want to look out for performance differences between them in your MDX provider.

<table>
<thead>
<tr>
<th></th>
<th>Q1, 2005</th>
<th>Q1, 2005</th>
<th>Q2, 2005</th>
<th>Q2, 2005</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dollar Sales</td>
<td>Unit Sales</td>
<td>Dollar Sales</td>
<td>Unit Sales</td>
</tr>
<tr>
<td>Tools</td>
<td>183,042.00</td>
<td>7,179</td>
<td>199,809.20</td>
<td>7,912</td>
</tr>
<tr>
<td>Toys</td>
<td>149,435.30</td>
<td>5,878</td>
<td>169,513.10</td>
<td>6,476</td>
</tr>
</tbody>
</table>

Figure 1-11  CrossJoined dimensions on columns.
CrossJoin() is standard MDX. Microsoft Analysis Services also provides an extension to express this as "set multiplication" by using * (asterisk):

{ [Time].Members } * { [Scenario].Members } * { [Product].Members }

This performs the same operation as CrossJoin(), and may be easier for you to read and write when you are not concerned with portability.

A common use for CrossJoin() is to combine a single member of one dimension with a set of members on another dimension, such as creating a set in which a particular measure is referred to over a set of tuples from other dimensions. When the formula of one calculated measure involves the count of nonempty cells for another measure, this construct is required. Although it might seem preferable, you cannot construct tuples on multiple dimensions by using range operators. For example, to express the range "toothpaste in stores 1 through 10," you might want to write something like the following:

( {[Product].[Toothpaste],
   {[Geography].[Store 1] : [Geography].[Store 10] })

Instead, you will need to use CrossJoin() (or the * variant), such as in the following:

CrossJoin {
    { [Product].[Toothpaste] },
    [Geography].[Store 1] : [Geography].[Store 10]
}

In the phrasing in the CrossJoin() example, we did not use curly braces around the set; they were not needed there. However, since the function requires a set, we did use them around the single member [Toothpaste], so we could convert the tuple to a set.

**NOTE** In Analysis Services 2005, CrossJoin() doesn’t always create a full Cartesian product. See Chapter 4 for more details.

---

**Filter()**

Operators like CrossJoin() and : help you construct sets. In contrast, Filter() lets you reduce a set by including in the resulting set only those elements that meet some criteria. Filter() takes one set and one Boolean expression as its arguments and returns that subset where the Boolean expression is true.
The syntax for Filter() is:

```
Filter (set, boolean-expression)
```

For example, the expression

```
Filter (
    { [Product].[Product Category].Members },
    [Measures].[Dollar Sales] >= 500
)
```

will return a set of all product category members in which the associated Dollar Sales value was at least 500. This is the first time we have used comparisons. Any Boolean expression may be used to filter the set. As a more complex example, the expression

```
Filter (
    { [Product].[Product Category].Members },
    ([Measures].[Dollar Sales] >= 1.2 * [Measures].[Dollar Costs])
    AND [Measures].[Dollar Sales] >= 150
)
```

will return a set of all product category members in which the associated sales measure value was at least 1.2 times the associated cost measure value and the sales value was at least 150. How do you specify the associated values? The answer to that has important details, which we will address in Chapter 4. For the moment, rely on your intuition.

Filter() works on general sets of tuples, not just on sets of one dimension’s members. The following expression returns the set of all (product category, city) tuples in which the associated sales value was at least 500:

```
Filter (CrossJoin (
    { [Product].[Product Category].Members, [Store].[City].Members
),
    [Measures].[Dollar Sales] >= 500
)
```

In determining the value of sales associated with each product category, or each (product category, city) tuple, you must take into account the other dimensions that are associated with sales values. For example, the first two Filter() expressions and the last one did not account for the time or times with which the sales values were associated. You can specify any additional dimensions’ members that you need to in either the Boolean condition or in
the set. For example, if you wanted to filter based on 2000’s sales in Baton Rouge, you would simply say:

```
Filter (  
    [Product].[Product Category].Members,  
    ([Measures].[Dollar Sales], [Time].[2000],  
    [Store].[Baton Rouge, LA]) >= 500
)
```

Within the filtering operation, the cell value will be taken from the 2000 Baton Rouge sales at each product category. The result is a set of product category members.

On the more advanced side, you can also specify more members in the set. For example, the preceding operation could be specified as follows:

```
Filter (  
    CrossJoin (  
       {([Time].[2000], [Store].[Baton Rouge, LA]) },  
       [Product].[Product Category].Members  
    ),  
    [Measures].[Dollar Sales] >= 500
)
```

This expression filters a set of tuples that all include 2000 and Baton Rouge, thus fixing on the correct time and store. However, the set returned would consist of tuples with dimensionality:

```
{[Time], [Store], [Product]}
```

These Filter() expressions have introduced the concept of query context (the relevant members for the dimensions that are not part of the filter condition or the set being filtered). Every MDX expression ultimately operates in a context that is set up outside it. Nested MDX operations are resolved within the context of the operation that invokes the nested operation. We’ll defer discussion of contexts here; Chapter 4 explains query evaluation and context in far more detail.

---

**POWER OF TUPLES**

The expression `([Measures].[Dollar Sales], [Time].[2000],  
[Store].[Baton Rouge, LA]) >= 500` is the first use of the power of tuple references to precisely refer to cells holding data values of interest. We will show how to exploit this many times; it is one of the powerful aspects of MDX.
**Order()**

To put the tuples in a set into a sequence based on associated data values, we need to use the `Order()` function.

The syntax for the `Order()` function is:

```
Order (set1, expression [, ASC | DESC | BASC | BDESC])
```

`Order()` takes a set, a criterion for ordering the set, and, optionally, a flag that indicates what sorting principle to use (ascending or descending, including or ignoring hierarchical relationships between the tuples). `Order()` returns a set that consists of the original set’s tuples placed in the new order. The precise operations of the orderings that include hierarchical relationships are fairly complex. Appendix A includes a complete description. Here, we will use simpler examples that don’t show as much complexity.

For example, given the set of product categories in our database, we may want to sort them in descending order by sales. A very simple query for this may look like the following; its results are shown in Figure 1-12:

```
SELECT
  { [Measures].[Dollar Sales] } on columns,
Order (  
  [Product].[Product Category].Members,
  [Measures].[Dollar Sales],
  BDESC
)
on rows
FROM [Sales]
WHERE [Time].[2004]
```

Often, you will want to use particular tuples to precisely specify the sort criteria. For example, let’s say that you want to sort specifically by the profit realized in 2005 by all customers. This would be expressed by the following:

```
Order (  
  [Product].[Product Category].Members,
  ([Measures].[Profit], [Time].[2005], [Customer].[All Customers]),
  BDESC
)
```

For example, the following query (whose results are shown in Figure 1-13) provides multiple time periods, customers, and measures, so you would want to use a tuple to pick out which ones to sort on:
SELECT
CrossJoin (
  {[Time].[2004], [Time].[2005]},
  CrossJoin (  
    { [Customer].[Northeast], [Customer].[West] },
    { ([Measures].[Dollar Sales], [Measures].[Unit Sales] }
  )
) on columns,
Order (  
  [Product].[Product Category].Members,
  ([Measures].[Unit Sales], [Time].[2005],
   [Customer].[All Customers]),
  BDESC
) on rows
FROM [Sales]

As you can see, the rows are sorted by the values related to the (2005, West, Dollar Sales) tuple, which corresponds to the next-to-last column.

Because Order() works on tuples, you can also sort the interesting (product and store) combinations by their unit sales. For example, the following expression filters (product and promotion) tuples by Dollar Sales, then orders each resulting (product and promotion) tuple according to its unit sales in 2005, and returns them; Figure 1-14 shows the resulting order.

ORDER (  
  Filter(
    CrossJoin (  
      [Product].[Product Category].Members,
      [Promotion].[Media].Members
    )
    , [Measures].[Dollar Sales] >= 500
  )
  , ([Measures].[Unit Sales], [Time].[2005])
  , BDESC
)

<table>
<thead>
<tr>
<th>Tools</th>
<th>894,495.80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computers, Peripherals</td>
<td>847,595.00</td>
</tr>
<tr>
<td>Toys</td>
<td>768,126.20</td>
</tr>
<tr>
<td>Camping, Hiking</td>
<td>646,902.40</td>
</tr>
<tr>
<td>Phones</td>
<td>640,023.80</td>
</tr>
<tr>
<td>Outdoor Gear</td>
<td>572,151.00</td>
</tr>
<tr>
<td>Sports Equipment</td>
<td>541,098.50</td>
</tr>
<tr>
<td>Exercise, Fitness</td>
<td>534,700.60</td>
</tr>
<tr>
<td>TV, DVD, Video</td>
<td>500,910.80</td>
</tr>
</tbody>
</table>

**Figure 1-12** Result of query using Order().
Note that the BDESC variant breaks (that is, ignores) the hierarchy. You’d get back a more complex and possibly more interesting ordered set if you instead chose DESC, which respects the hierarchy and the dimensional components of tuples. See the section “Ordering Sets” in the description of `Order()` in Appendix A for a full description of ordering sets.

### Querying for Member Properties

An MDX query can also retrieve member properties defined for members in a cube along the query’s axes using the `DIMENSION PROPERTIES` clause. For example, the following will query for the zip code and hair color of customers returned in the query:

```sql
SELECT
  { [Customer].[Akron, OH].Children }
DIMENSION PROPERTIES [Customer].[Zip Code],
  [Customer].[Individual].[Hair Color]
on columns,
  { [Product].[Category].Members } on rows
FROM Sales
WHERE ([Measures].[Units Sold], [Time].[July 3, 2005])
```
The **DIMENSION PROPERTIES** clause comes between the set for the axis and the **ON Axis(n)** clause. The **DIMENSION** keyword is optional; you could also write:

```
SELECT
  { [Customer].[Akron, OH].Children }
PROPERTIES [Customer].[Zip Code],
            [Customer].[Individual].[Hair Color]
ON COLUMNS,
...
```

Member properties can be requested on any axis but not on the slicer in standard MDX. (Essbase 9 extends MDX to allow a **DIMENSION PROPERTIES** clause on the slicer.)

If a member property is defined for only a single level of a dimension, and the query axis includes multiple levels, the query will succeed and you will just not get values for members at the inapplicable levels. If a member is repeated in a result axis, its related member property value will be repeated too.

Properties can be identified either by using the name of the dimension and the name of the property, as with the zip code property just given, or by using the unique name of the dimension’s level and the name of the property, as with the hair color property.

**NOTE** While the values of properties requested with the **PROPERTIES** statement in an MDX query are returned, along with all other result information, in the result data structures, it is up to the client application to retrieve and utilize this information.

Both intrinsic and database-defined member properties can be queried. Analysis Services defines intrinsic properties named **KEY**, **NAME**, and **ID**, and every level of every dimension has them. (These, and other kinds of properties for Analysis Services, are described in Appendix C.) For example, the **KEY** property of a Product dimension’s SKU level is named `[Product].[SKU].[KEY]`. The member key property contains the values of the member keys as represented in the dimension table. The member name property contains the values of the member names as represented in the dimension table. The **ID** property contains the internal member number of that member in the dimension-wide database ordering. (Since these properties are maintained internally, your application should not use them to avoid problems with ambiguous names.) Essbase 9 defines intrinsic properties named **MEMBER_NAME**, **MEMBER_ALIAS**, **GEN_NUMBER**, **LEVEL_NUMBER**, **MEMBER_UNIQUE_NAME**, **IS_EXPENSE**, **COMMENTS**, and **RELATIONAL_DESCENDANTS**. These are the names for which Essbase is case-sensitive: if you enclose these names in [], they should be in all caps (for example, `[MEMBER_NAME]`).
When property names between levels of a dimension are ambiguous, you can get ambiguous results if you query for member properties on the axis of a query. For example, every layer of an organizational dimension may have a Manager property for each member above the leaf. Consider the following query fragment:

```mdx
SELECT { Descendants ([Organization].[All Organization], [Organization].[Junior Staff], SELF_AND_ABOVE ) PROPERTIES [Organization].[Manager] on columns ...
```

When the query is executed, it will return the specific Manager property for only one level. It is not a good idea to rely on whatever level that would happen to be. (In our experience, it would be the lowest level in the query, or the Junior Staff level in this case.) Members belonging to that level will have a valid [Manager] value; members belonging to other levels won’t. Suppose that, instead, you queried for each level’s properties independently, as with the following:

```mdx
SELECT { Descendants ([Organization].[All Organization], [Organization].[Junior Staff], SELF_AND_ABOVE ) PROPERTIES [Organization].[Executive Suites].[Manager], [Organization].[Middle Managers].[Manager], [Organization].[Junior Staff].[Manager] on columns ...
```

In this case, the property for each level at each level’s member will arrive appropriately filled in (and be empty at members of the other levels). However, when you access properties in member calculations, there won’t be any ambiguity. Suppose, for example, that some calculated member referred to (in Microsoft syntax) `[Organization] .CurrentMember.Properties ("Manager")`. (Appendix A provides a detailed reference to this function, and we also use it in the “Using Member Properties in MDX Expressions” section of Chapter 3.) The lookup of this value is done on a cell-by-cell basis, and at each cell the particular manager is unambiguous (though the level of manager to which it refers may change). For this case, you can easily and simply reference member properties on multiple levels that share the same name.

**Querying Cell Properties**

Querying for specific cell properties is fairly tightly bound to the programming layer that retrieves results from a query. In keeping with the nonprogramming focus of this book, we won’t cover all of the programming details
here. (We do introduce client programming in Chapter 15.) However, we will 
explain the basic model that querying for specific cell properties supports and 
how an application might use it. This explanation is relevant to OLE DB for 
OLAP, ADO MD and ADO MD.Net, and XMLA.

Every query is a specification of one or more result cells. Much as each mem-
ber is able to have one or more related properties, each result cell also has more 
than one possible result property. If a query specifies no cell properties, then 
three properties are returned by default: an ordinal number that represents the 
index of the cell in the result set, the raw value for the cell in whatever data 
type is appropriate, and the formatted textual value for the cell. If the query 
specifies particular cell properties, then only those are returned to the client. 
We discuss formatting the raw value into text in the section “Precedence of 
Display Formatting” in Chapter 4. The ordinal cell index value is germane to 
client tools that are querying the data that has been generated through OLE DB 
for OLAP or XMLA. Other cell properties can be queried for, which can be 
specified for any measure or calculated member in the cube. The full list of cell 
properties and how they are used in OLE DB for OLAP and ADO MD/.Net is 
found in Appendix C.

The way to specify cell properties in a query is to follow the slicer (if any) 
with the CELL PROPERTIES keywords and the names of the cell properties. 
For example, the following query

```
SELECT
{ [Measures].[Units Returned], [Measures].[Value Returned] } on columns,
{ [Time].[2000], [Time].[2001] } on rows
FROM InventoryCube
CELL PROPERTIES FORMATTED_VALUE
```

returns to the client only the formatted text strings that correspond to the 
query results. Generally speaking, clients that render their results as text 
strings (such as spreadsheet-style report grids in ASP pages) will be most 
interested in the formatted values. Clients that render their results graphically 
(such as in bar charts where each height of each bar represents the value of the 
measure at that intersection) will be most interested in the raw values. An 
Excel-based client will benefit from retrieving the raw value and the format 
string together. Other OLE DB for OLAP standard properties available in 
Analysis Services enable string formatting, font, and color information to be 
stored and retrieved for measures and calculated members. This gives you 
server-side control over useful client rendering operations. Analysis Services 
adds more cell properties covering cell calculation specifics.

Our discussion of CREATE MEMBER in Chapter 12 describes how to specify 
the various cell properties that should be associated with calculated members. 
In Chapter 4, we describe how calculated members influence cell properties in 
queries.
Some development teams avoid using formatted values, claiming that it is preferable to separate formatting from the database. However, that requires mid-tier or front-end code to be conscious of the meanings of individual members and tuples, making generic display code quite complicated, while not delivering one jot of different capability. You can also think of that as breaking encapsulation. In our experience, they are extremely helpful, and can always be overridden by client code anyway if you really want to.

Client Result Data Layout

Although the result of an MDX query is a data structure, it is typically converted to some human-accessible rendering. When more than one dimension is on an axis, there is a convention that you should be aware of for how clients use the results. Recall that the order of dimensions in a tuple is important in MDX. Consider, again, the sample query involving `CrossJoin()`:

```sql
SELECT
  CrossJoin (
    { [Time].[Q1, 2005], [Time].[Q2, 2005] },
    { [Measures].[Dollar Sales], [Measures].[Unit Sales] }
  )
ON COLUMNS ,
( [Product].[Tools], [Product].[Toys] ) ON ROWS
FROM Sales
```

The Measures dimension is the second dimension in each of the tuples, and the Time dimension the first. The first dimension in a tuple is the one at the outermost level of nesting as it comes out of the `CrossJoin()`, and it is also the outermost one in a client’s rendering of the results. The last dimension in a tuple is the innermost, both from `CrossJoin()` and in client’s rendering.

The order of dimensions in tuples, when they show up in an axis, is reflected in the order of the dimensions in the returned data structures.

<table>
<thead>
<tr>
<th></th>
<th>Q1, 2005</th>
<th>Q1, 2005</th>
<th>Q2, 2005</th>
<th>Q2, 2005</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dollar Sales</td>
<td>Unit Sales</td>
<td>Dollar Sales</td>
<td>Unit Sales</td>
</tr>
<tr>
<td>Tools</td>
<td>183,042.00</td>
<td>7,179</td>
<td>199,809.20</td>
<td>7,912</td>
</tr>
<tr>
<td>Toys</td>
<td>149,435.30</td>
<td>5,878</td>
<td>169,513.10</td>
<td>6,476</td>
</tr>
</tbody>
</table>

Figure 1-15  Typical client rendering of rows and columns.
Summary

While we have covered only the basics of MDX queries, we really have covered quite a lot, and fairly quickly. Let’s summarize what we have gone over:

- The SELECT . . . FROM . . . WHERE framework of a query
- Axis names and numbering, and the slicer
- The tuple and set data model
- The most basic and frequent functions and operators: braces ({}), commas (,), parentheses for tuple construction (()), and the colon (:) for range construction; .Members, .Children, Descendants(), CrossJoin(), Filter(), and Order()
- Referencing member properties as additional query results
- Referencing cell properties
- Including comments
- Removing empty slices with NON EMPTY

We have also tried to emphasize the modular nature of MDX and explain expressions as an independent concept from queries. Whew! You might want to take a break before plunging on to the next chapter. In the next chapter, we will build on this understanding to start performing calculations.