Now that we have seen a variety of common operations expressed in MDX, let us turn our attention to more advanced expressions and queries. This chapter continues from the last chapter, expanding the scope of topics. In this chapter, we’re going to work through some more advanced concepts and techniques. Many of these come from assistance the author has given to other MDX users over the last two years, so if they seem esoteric at first, remember that MDX has many different applications!

The expressions and queries in this chapter are oriented more toward the composition of whole analytical queries, and the principles that go into constructing them are useful for constructing all manner of sophisticated analyses, even if the particular examples we present in this section do not address the applications you are going to build.

The types of expressions and queries we are going to explore here are:

- Using Generate() to turn tuple operations into set operations
- Calculating dates and date arithmetic
- Report totals-to-parent, totals-to-ancestor, and percentage contribution to report totals
- Hierarchical sorting that skips levels in the hierarchy
- Pareto analysis and cumulative sums
Sorting on multiple criteria
- Reporting on ranks (and ranks from another time period)
- Building a set that sums backward in time
- Drilling down from virtual dimensions
- One formula calculating different things in different places (also called “formula application ranges”)
- Logical aggregations (ForAny, ForEach, ForNone)
- Including all tuples with tied ranking in sets

After we work through these expressions and queries, we will apply the same principles for using MDX we discuss here in a complete series of analytical queries.

Using Generate() to Turn Tuple Operations into Set Operations

How do I get the descendants / ancestors / children / and so on from a set of members?

Most of the MDX functions and operators that return members based on hierarchical references (such Descendants(), Ancestor(), and the like) only work on a single member. However, you may find that you need to take the descendants or ancestors for a set of members. The way to accomplish this is to use Generate() around the function or operator to get it to return the set that you want.

For example, say that you want to take the average of a measure over a set of day-level descendants for a set of quarters or months that the user has chosen. The following query is tempting but will not work:

```mdx
WITH
SET InterestingMonths AS
  'LastPeriods ([Time].[Month].[Apr 2000], 3)'
MEMBER [Time].[AverageVal] AS
  'Average (Descendants ([InterestingMonths], // problem!
[Time].[Day])
)', SOLVE_ORDER = 10
SELECT ...
```
Instead, we need to define `[Time].[AverageVal] as

'Average {
  Generate {
    [InterestingMonths],
    Descendants {
      [Time].CurrentMember,
      [Time].[Day]
    }
  }
}', SOLVE_ORDER = 10

which loops over each given time period (be it a month, a quarter, or even a day) and produces the related day members. We’ll see another example of this use of Generate( ) in the Report Totals-to-Ancestor section later in this chapter.

Calculating Dates/Date Arithmetic

Sometimes, an application calls for calculating values based on dates. Perhaps you need to calculate the number of days between two dates, or the number of days between a date-valued measure or member property and a day member in a cube. Perhaps you need to calculate the number of weekdays between two day-level members (or in a month, which works out to nearly the same thing). Or, perhaps you need to calculate an expected date for something. SQL has a fairly rigorous definition of dates and a set of date calculation functions built in. Microsoft’s OLAP/Analysis Services does not contain a notion of a date/time data type, and MDX does not provide much by itself, but we can leverage external functions and MDX capabilities to do a lot of what we need.

As usual, these calculations can be implemented in more than one way. Depending on your database design, your MDX will look different.

NOTE

All of the examples here are assuming a U.S. locale for constructing and parsing text representations of date values. If you are implementing in Europe, Canada or elsewhere, you will need to adjust for our assumptions.

If you are simply trying to calculate the number of days between two day-level members, and every day in the calendar has a corresponding day in the database, then you just need to count the members.

{ First_Day : Last_Day }.Count

is a brief expression that performs the count of the days (and you could subtract one from it to get the number of days in between the first and last days).
Other types of date-related calculations can rely on VBA or Excel functions. For example, let’s say that you are trying to count the weeks between two dates. The VBA DateDiff() function can directly calculate this number, so you only need to pass the two dates into it. Assuming that the datestamp for the day members is stored as the DateStamp member property, you will also need the VBA CDate() function to convert it from a string into an input for DateDiff().

You can simply define the member (in the WITH clause of a query) as:

```vba
MEMBER [Measures].[WeeksBetween] AS
    'VBA!DateDiff ('
    "w", // this tells it to calculate weeks
    CDate ([Time].[Day1].Properties ("DateStamp")),
    CDate ([Time].[Day2].Properties ("DateStamp"))
    ')

(The VBA! prefix makes explicit which library the DateDiff() function is being called from; we discuss this notation in Chapter 7 “Extending MDX Through External Functions”) This will only produce one number. To count the number of weeks between a particular day and each day returned in a query, define the member as:

```vba
MEMBER [Measures].[WeeksBetween] AS
    'VBA!DateDiff ('
    "w",
    CDate ([Time].[Day1].Properties ("DateStamp")),
    CDate ([Time].CurrentMember.Properties ("DateStamp"))
    ')
```

Let’s say that the starting date is a member property of the customer dimension that refers to the date the customer enrolled with our organization, named [Date Enrolled]. Then, for each customer we could get the number of weeks the customer has been enrolled. The [Time].[Day1].Properties("DateStamp") in these expressions would be replaced by [Customer].CurrentMember.Properties ("Date Enrolled"). If we have unique names across all levels of our time dimension in Analysis Services and the format of the string stored in [Date Enrolled] matches the format of our time member names (that is, “May 01, 2000”), we could substitute StrToMember ("[Time].[ " + [Customer].CurrentMember.Properties ("Date Enrolled") + "]") in for the [Time].[Day1] of the preceding examples.

**TIP**

Using unique names across the members of a time dimension (for example, [2001], [Q2, 2001], [May 2001], and [May 11, 2001]) makes it easy to perform date translations in MDX.
Another variation on this is when you are forming a query that involves a set of days from the time dimension, perhaps not every day but some days (for example, December 4, 5, 6, 9, and 10, 1999). If you want to find out how many calendar days are between the first and the last day members in the query, then you need to extract those members and plug them into the kinds of expressions that we have been looking at.

Let’s assume that you simply need to calculate a time member that returns the span of days represented by the set. The VBA DateDiff() function comes in handy in this example:

```
WITH
SET [Times] AS
  '{ [Dec 04, 1999], [Dec 05, 1999], [Dec 06, 1999], [Dec 09, 1999], [Dec 10, 1999] }'
MEMBER [Time].[DaysInQuery] AS
  'VBA!DateDiff (d, CDate ([Times].Item(0).Item(0).Properties("DateStamp")), CDate ([Times].Item ([Times].Count - 1).Item(0).Properties("DateStamp")))'
```

To calculate a date in the future, you once again might want to look at the VBA functions. For example, given a date value and a calculated measure that yields a number of days, you can use the VBA DateAdd() function to add the two together. (The result might be converted to a string and returned to the client as a raw value, or it might be used as an argument to the Members() function to look up a time member for further work.)

If your time dimension is broken up so that years are in one dimension and quarters/months/days are in another, then you need to rely on different techniques. This is because your day members won’t necessarily have a timestamp that corresponds to a particular year as opposed to a generic year like 1900. For example, in order to count the number of days between a customer’s enrollment date and a particular date in the cube, you will need to combine the day-level datestamp with the year-level year value in some way. Let’s say that you have datestamps for days that are days in 1900 (Jan-01-1900 for January 1, and so on). The VBA DateAdd() function can add a number of years to the date value, so you could combine the year number from the year dimension with the day value in the day/month/quarter dimension to arrive at the date value to work with.
Combining the strings that form the member names is the easiest technique for this example and avoids problems with datestamps and leap years. For example

```
MEMBER Measures.WeekDaysBetween AS 'VBA!DateDiff ( 
  "d",  
  CDate ([Customer].CurrentMember.Properties ("EnrollDate")),  
  CDate ([Time].CurrentMember.Name + "/" + [Year].CurrentMember.Name) 
),
```

would construct the name and convert to a date.

---

**Report Totals-to-Parent, Percentage Contribution to Report Totals**

Some reports need to provide a subset of children or a subset of descendants for a member, along with a total of those children or descendants. For example, for a group of interesting customers within a sales region, we want to see summary values for them at the city and state level. In addition to seeing the summary values, we want to see the influence of each of them as a percentage of all the interesting customers in that group. OLAP Services provides intrinsic aggregation of all children to their parents (and all descendants to their ancestors), but this may include many children and descendants that are not part of the query. (In a query of this sort, in MDX we are really creating a proxy set of levels and ancestor members whose names are the same as the database’s, but who represent a different and interesting subset of the original dimension.)

**NOTE**

You can set a connection property when the client connects to the server (Default MDX Visual Mode) that can automatically set every query to use a VisualTotals aggregation for parent members. We describe this in some detail in Appendix B.

VisualTotals() is a Microsoft OLAP/Analysis Services-specific function that will give us the report total to a parent member, which makes some of this process easy when only parents and children are involved. In essence, VisualTotals() returns a set that includes a new calculated member as a part of it, defined within VisualTotals(). For example, the query

```
SELECT { [Measures].[Sales], [Measures].[Costs] } on columns,
{ [Customer].[Fargo, ND],
  VisualTotals ( {
    [Customer].[Fargo, ND],
```

```
Advanced MDX Application Topics

would generate a straightforward report of three customers in Fargo, North Dakota, and their total sales and costs, along with the database total for Fargo, North Dakota, for contrast. The result would appear as shown in Figure 4.1. This approach is expedient when we only want the total values for children values, grouped according to their hierarchy. VisualTotals() also let us total up descendants farther away than children and in fact enable us to use descendants at multiple levels without double-counting their results. Using VisualTotals() is very convenient because it does not require us to devote any logic to the creation of calculated members:

```
WITH
SET [InterestingCustomers] AS '{
    [Customer].[Fargo, ND].[C#4521772],
    [Customer].[Fargo, ND].[C#2329384],
    [Customer].[Fargo, ND].[C#3847321]
}
MEMBER [Customer].[Explicit Total in Fargo, ND] AS
    'Aggregate ( [InterestingCustomers] )'
SELECT
    { [Measures].[Sales], [Measures].[Costs] } on columns,
    { [Customer].[Fargo, ND], [Customer].[Explicit Total in Fargo, ND],
      [InterestingCustomers] } on rows
FROM Shipments
```

However, when we wish to calculate percentage-of-total contributions for each customer in our [InterestingCustomers] set, we do need to create calculated members:

```
WITH
SET [InterestingCustomers] AS '{
    [Customer].[Fargo, ND].[C#4521772],
    [Customer].[Fargo, ND].[C#2329384],
    [Customer].[Fargo, ND].[C#3847321]
}'
MEMBER [Customer].[Explicit Total in Fargo, ND] AS
    'Aggregate ( [InterestingCustomers] )'
SELECT
    { [Measures].[Sales], [Measures].[Costs] } on columns,
    { [Customer].[Fargo, ND], [Customer].[Explicit Total in Fargo, ND],
      [InterestingCustomers] } on rows
FROM Shipments
```

### Figure 4.1
Report totals versus VisualTotals.

<table>
<thead>
<tr>
<th></th>
<th>Sales</th>
<th>Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fargo, ND</td>
<td>25,000</td>
<td>18,000</td>
</tr>
<tr>
<td>Total in Fargo, ND</td>
<td>9,000</td>
<td>8,850</td>
</tr>
<tr>
<td>C#4521772</td>
<td>6,000</td>
<td>7,000</td>
</tr>
<tr>
<td>C#2329384</td>
<td>2,000</td>
<td>1,000</td>
</tr>
<tr>
<td>C#3847321</td>
<td>1,000</td>
<td>850</td>
</tr>
</tbody>
</table>
members. This is because the member total created by VisualTotals does not have a name that we can reference in our query. When the query is parsed, the name has not yet been generated, and though we could form the expression (sales / [Total in Fargo, ND]), the MDX parser will not know what member it corresponds to because it will not exist until later.

The percentage-to-total calculation does not belong in the customer dimension. It is best suited to the measures dimension, on a measure-by-measure basis. For example, consider the following query:

```mdx
WITH
SET [InterestingCustomers] AS
'{
    [Customer].[Fargo, ND].[C#4521772],
    [Customer].[Fargo, ND].[C#2329384],
    [Customer].[Fargo, ND].[C#3847321]
}
MEMBER [Customer].[Explicit Total in Fargo, ND] AS
'Aggregate ([InterestingCustomers])'
MEMBER [Measures].[Percent to Report Sales Total] AS
'Sales / (Sales, [Customer].[Explicit Total in Fargo, ND])'
SELECT
{ [Measures].[Sales], [Measures].[Costs],
  [Measures].[Percent to Report Sales Total] } on columns,
{ [Customer].[Fargo, ND], [Customer].[Explicit Total in Fargo, ND],
  [InterestingCustomers] } on rows
FROM Shipments
```

We can use this approach of creating a named set, a calculated member on the same dimension, and one or more calculated measures for any set of members that aggregate into one member only. More difficult, however, is when we wish to see a set of descendants and their contributions to ancestors for more than one ancestor. For example, we may have a set of interesting customers in North Dakota, South Dakota, Minnesota, and Wisconsin. Our last approach would run into a few complications when we try to extend it. The [Percent to Report Sales Total] measure hardwires the Fargo, North Dakota, ancestor member within it. We would need to add a separate percentage measure for each ancestor, which is an ugly solution. It would create a set of four measures, each of which would only have a valid intersection with one of the four state-level members, something we would prefer to avoid. In addition, we would need to do other things like generate multiple sets of interesting customers, one for each Explicit Total pseudo-parent.

We can avoid this ugliness if we create a new calculated report total measure and use some of the set manipulation functions provided in MDX. Let’s say that we have a set of interesting customers in three different states:

```mdx
SET [InterestingCustomers] AS ' {
    [Customer].[Fargo, ND].[C#4521772], [Customer].[Fargo,
Let's also say we wish to get the sum of sales and costs within each of these customers' states, along with the percentage contribution of each customer to that total. We do need to include the ancestor members within the query. If we are starting only with a set of customers, we can do that with Generate() (because Ancestor() only works on members, not on sets). A hierarchically ordered set to be used as one axis of the query could be obtained from an arbitrary set of customers with the following query:

```plaintext
SET [rowset] AS 'Hierarchize(
  [InterestingCustomers],
  Generate(
    [InterestingCustomers],
    { Ancestor([Customer].CurrentMember, [Customer].[State]) }
  )
)'
```

The real heart of the query is in the calculated measures that compute the report total of the sales and cost measures. Each is similar. The one for sales looks like this:

```plaintext
MEMBER [Measures].[Report Total Sales] AS 'Sum(
  Intersect([InterestingCustomers],
  Descendants([Customer].CurrentMember, [Customer].[Individual Customer]))
),
[Measures].[Sales])'
```

At each customer member, we take the individual customer(s) that corresponds to that customer member. Intersecting that with our customers of interest gives us only the interesting individual customers that are under our customer member. (Or it will give us the interesting customer back, if one of the customers is our current member.) For each state-level member, the sum over that set gives us the report total sales for the interesting customers within that state.

Because we are using real state-level members to organize our report total sales, we can simply take the ratio of individual customer sales to the ancestor's Report Total Sales to obtain our percentage of total, as in the following code:

```plaintext
MEMBER [Measures].[Sales Percent of Report Total] AS '[Measures].[Sales] / ([Measures].[Report Total Sales], Ancestor([Customer], [Customer].[State]))'
```

So, the whole query rolled together would be as follows (excluding the details of the individual customers), with the query's result shown in Figure 4.2.
WITH
SET [InterestingCustomers] AS '...' 
SET [Rowset] AS 'Hierarchize ([InterestingCustomers],
    Generate ([InterestingCustomers],
        { Ancestor([Customer].CurrentMember, 
            [Customer].[State]) }) )'
MEMBER [Measures].[Report Total Sales] AS 'Sum ( 
Intersect ([InterestingCustomers],
    Descendants ([Customer].CurrentMember, 
        [Customer].[Individual Customer]), [Measures].[Sales])' 
MEMBER [Measures].[Report Total Costs] AS 'Sum ( 
Intersect ([InterestingCustomers],
    Descendants ([Customer].CurrentMember, 
        [Customer].[Individual Customer]), [Measures].[Costs])' 
MEMBER [Measures].[Sales Percent of Report Total] AS '[Measures].[Sales] / ([Measures].[Report Total Sales], Ancestor ([Customer], [Customer].[State]))', FORMAT_STRING = '#.00%' 
SELECT
    { [Measures].[Report Total Sales], [Measures].[Report Total Costs],
        [Measures].[Sales Percent of Report Total], [Cost Percent of Report Total] } on columns,
    { [Rowset] } on rows
FROM Shipments

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MN</td>
<td>15,500</td>
<td>10,500</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>C#3492945</td>
<td>3,500</td>
<td>4,500</td>
<td>22.58%</td>
<td>42.86%</td>
</tr>
<tr>
<td>C#7123693</td>
<td>12,000</td>
<td>6,000</td>
<td>77.42%</td>
<td>57.14%</td>
</tr>
<tr>
<td>ND</td>
<td>8,000</td>
<td>8,000</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>C#2329384</td>
<td>2,000</td>
<td>1,000</td>
<td>25.00%</td>
<td>12.5%</td>
</tr>
<tr>
<td>C#4521772</td>
<td>6,000</td>
<td>7,000</td>
<td>75.00%</td>
<td>87.5%</td>
</tr>
<tr>
<td>SD</td>
<td>10,300</td>
<td>6,500</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>C#1012233</td>
<td>5,800</td>
<td>3,000</td>
<td>56.31%</td>
<td>46.15%</td>
</tr>
<tr>
<td>C#8212633</td>
<td>4,500</td>
<td>3,500</td>
<td>39.69%</td>
<td>53.85%</td>
</tr>
</tbody>
</table>

Figure 4.2  Full report totals and percent total results.
Hierarchical Sorting That Skips Levels in the Hierarchy

The hierarchical sorting provided by the Order() function is very convenient when we wish to display data sorted within all hierarchical relationships. Sometimes, however, we may wish to sort by hierarchical relationships but not use all levels in the hierarchy. For example, we may use a geography dimension in a cube that has levels of state, city, and store. If we sort our geography members hierarchically by profit per unit, we will get each store sorted per city and each city sorted per state, whether or not those intermediate levels are in the report. How, in a report, can we sort each store per state, leaving out the cities?

The answer is to break apart the sorting so we are sorting the cities within each state independently. The Generate() function provides us with the iterative framework we need to do this. The first set can be our set of states, and the second set can be an expression that sorts the descendants of the current member of the first set:

```mdx
Generate(
    { [Geography].[State].Members },   // for each state
    Order(     // sort its stores
        { Descendants( [Geography].CurrentMember, [Geography].[Store], SELF) },
        [Measures].[Profit Per Unit],
        BDESC
    )
)
```

It is important, of course, to specify BDESC or BASC rather than ASC or DESC. In a report where we wish to hierarchically represent the states with the stores, we can add them into the second set for Generate():

```mdx
Generate(
    { [Geography].[State].Members },  // for each state
    { [Geography].CurrentMember,   // add the state
        Order(                       // and its sorted stores
            { Descendants( [Geography].CurrentMember, [Geography].[Store], SELF) },
            [Measures].[Profit Per Unit],
            BDESC
        )
    }
)
```

As an aside, the default hierarchical ordering always places parents before children, whether the sorting is ascending or descending. We can use the same sort
of construct we just gave to put our states after their children, in a typical subtotal format:

```plaintext
Generate(
    { [Geography].[State].Members },  // for each state
    Order(  // add its sorted stores
        Descendants([Geography].CurrentMember, [Geography].[Store], SELF),
        [Measures].[Profit Per Unit],
        BDESC
    ),
    [Geography].CurrentMember    // and add in the state
)
```

## Pareto Analysis and Cumulative Sums

When you query for parents and children, you can use `Order( )` with ASC or DESC to preserve the hierarchy while sorting. For example, given the store and employee count data values shown in Figure 4.3, the following set expression results in the ordering of stores shown in Figure 4.4:

```plaintext
Order(
    { [Store].[California],
      [Store].[California].Children,
      [Store].[Nevada],
      [Store].[Nevada].Children
    ),
    [Measures].[Employee Count],
    DESC
)
```

Now, let's say that for each of the children, we want the cumulative sum as we go from the first child in this ordering to the last child. This will essentially give us a Pareto analysis within each state. If we used the following expression,

```plaintext
[Measures].[Cum Employee Count] AS
'Sum ( {{Employee}.FirstSibling : {Employee}.CurrentMember},
[Measures].[Employee Count])'
```

our results will be quite wrong. This is so because .FirstSibling is not relative to this ordering, even though .CurrentMember will be.

To get the cumulative sum for each of the children as we go from the first child in this ordering to the last, we must re-create the range of ordered children within our calculated member. We also need to find our current tuple within the range, using `Rank( )`, and we need to create a range from the first tuple in the set to the referent child's tuple, using `Head( )`. The following expression gives us the proper ordering of members:
Order ( ([Employee].Parent.Children), [Measures].[Employee Count], DESC)

We obtain the ranking of any tuple in that set with

\( \text{Rank} \{ [[\text{Store}], \text{Set}] \} \)

The \( \text{Rank}() \) term gives us the rank of the current store member within that set to use as our cutoff point for aggregating. In addition, the expression

\( \text{Head}(\text{Set}, \text{Index}) \)
gives us the subset to aggregate. We must use the set once to get the members to aggregate over and another time to get them to rank over. Rolling it all together, we get

```
MEMBER [Measures].[Cum Employee Count] AS 'Sum (Head (Order ( {
[Store].Parent.Children}, [Measures].[Employee Count], BDESC),
  Rank ( [Store],
    Order ( {
      [Store].Parent.Children}, [Measures].[Employee Count], BDESC)
  )
  ),
  [Measures].[Employee Count] )'
```

A sample complete query would look like the following, with its results shown in Figure 4.5:

```
WITH
MEMBER [Measures].[Cum Employee Count] AS '. . .'
SELECT
  ( [Measures].[Employee Count], [Measures].[Cum Employee Count] ) on columns,
  ( [Store].[Nevada], [Store].[Nevada].Children, [Store].[California],
    [Store].[California].Children ) on rows
FROM [Employee Cube]
```

<table>
<thead>
<tr>
<th>Store</th>
<th>Employee Count</th>
<th>Cum. Employee Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nevada</td>
<td>129</td>
<td>129</td>
</tr>
<tr>
<td>Las Vegas</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td>Reno</td>
<td>55</td>
<td>117</td>
</tr>
<tr>
<td>Tahoe</td>
<td>12</td>
<td>129</td>
</tr>
<tr>
<td>California</td>
<td>124</td>
<td>253</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>72</td>
<td>72</td>
</tr>
<tr>
<td>Sacramento</td>
<td>49</td>
<td>121</td>
</tr>
<tr>
<td>Ukiah</td>
<td>3</td>
<td>124</td>
</tr>
</tbody>
</table>

**Figure 4.5** Cumulative employee counts.
The exact results for Nevada and California are dependent on the contents of their siblings, of course. If you want to only return results for [Cum Employee Count] at the city level and omit cumulative counts at higher levels, use the techniques we discuss for “One Formula Calculating Different Things in Different Places” later in this section.

For comprehensible if nothing else, it would be highly desirable to abstract the Order(...) clause into one definition and two uses. Although for each store member the set could be different (because it will be evaluated for the children of multiple parents), within the sum it doesn’t change. However, named sets are evaluated at the time they are parsed, not each time they are referenced, so we cannot use them. Microsoft does allow an alias to be used when the set is constructed that we can use for referencing it later on. We can make this a little easier on our eyes through the following re-phrasing:

```
MEMBER [Measures].[Cum Employee Count] AS 'SUM(
  Head(
    Order(
      {[Store].Parent.Children},
      [Measures].[Employee Count],
      BDESC
    ) AS OrdSet, // create an alias for the set
    Rank {
      [Store],
      OrdSet        // use it here
    }
  ),
  [Measures].[Employee Count]
)
'
```

Note that we have added an alias to the set through the use of “AS OrdSet”. We can then refer to OrdSet later on in the expression.

If constructing, maintaining, and executing this sort of MDX is less onerous to you than maintaining code in an ActiveX Automation language, then you can certainly do it in MDX. If you would rather maintain Automation-related code, then a UDF is the way to go. For example, with the following query you could call a UDF named PartialSum to take an ordered set of stores, an ordered set of values, and the name of the store to stop summing at

```
WITH
  MEMBER [Measures].[StoreUniqueName] AS ' [Store].UniqueName', VISIBLE = ' 0',
  MEMBER [Measures].[CumSum] AS 'PartialSum {
    SetToArray (Order( {[Store].Parent.Children}, [Measures].[Employee Count], DESC), [Measures].[StoreUniqueName]),
```

```
Notice that you need to create a dummy measure that contains the name of the current store member (which we called [Measures].[StoreUniqueName] in the example). The first release of Microsoft OLAP Services generates a parser error if you try to access the unique name directly in SetToArray( ). The same is true in Analysis Services, but at least we can hide the calculated member from the client application by setting its (Microsoft-specific) HIDDEN property to True.

**CHAPTER 4**

Sorting on Multiple Criteria

In MDX, sorting a set of tuples based on one criterion is directly supported through the Order( ) function. Somewhat surprisingly, it doesn’t support multiple criteria for sorting. Frequently, the default database ordering for dimensions provides the necessary primary and secondary sorting that a report needs with data providing the last level of sorting needed. (To see how this works in full detail, take a look at the description of the Order( ) function in Appendix A.) However, at times, you may need to sort based on more than one criterion. Although somewhat awkward, it is possible.

For example, let’s say that you need to sort a set of policyholders primarily by age and secondarily by total claims to date per customer. In this example, age is stored as a member property named [P Age] and is measured in whole years, whereas claims to date is represented by a calculated measure named [Measures].[Claims To Date]. We can directly use Order( ) to sort policy holders by age or by claims to date, but we need to resort to cleverness to sort by both of these. The trick is to combine the two numbers (age and claims) into a single number that can sort appropriately. How can we do this?

The answer is to scale the age value up by some amount and add the claims value to it. For example, if two policy holders have an age of 75 and one has claims to date of 300 and another of 3,000, we can get these in increasing order by multiplying the 75 by 10,000 and adding 300 and 3,000 to it; the numbers 750,300 and 753,000 will sort in correct order. (We are assuming ascending sorts in both criteria; you can apply the techniques we describe for mixed ascending and descending sorts as well.) All we need to figure out is what the right scaling factor is. For a wide range of queries, you can probably get by with an educated guess. (For example, if you were to sort on claims to date first and age second, you could safely multiply the age by 1,000.) If you can’t make a guess based on knowledge of the data, the right scaling factor is the absolute value of the dif-
ference between the max value and the min value across all values. For example, claims values could range from zero to millions of U.S. dollars. We could assume that multiplying age by 1,000,000,000 is safe, but it might not be for too long. In this case, because claims won’t go below zero, we can take the all-time, all-policy holder sum of [Claims to Date] as our scale factor; it is larger than necessary but also faster to calculate than the Min() or Max() across the database (unless we actually added Min and Max measures to the cube). The following query orders the policyholders primarily in terms of descending age, secondarily in terms of descending claims to date:

```mdx
WITH
... 
MEMBER [Measures].[P Age] AS 'Val ([Policy Holder].CurrentMember.Properties ("P Age"))'
MEMBER [Measures].[Claims Scale] AS '([Measures].[Claims To Date], [Policy Holder].[All Policy Holder])'
  // Add other [All] members to the Claims Scale tuple as necessary
SET [Ordered Policy Holders] AS 'Order ( 
  { Selected Policy Holder Members },
  ([Measures].[P Age] * [Measures].[Claims Scale])
  + [Measures].[Claims To Date]),
  BDESC
)
SELECT
  { [Measures].[Claims To Date], [Measures].[Policy Cost] } on columns,
  { [Ordered Policy Holders] } 
DIMENSION PROPERTIES [Policy Holder].[P Age] 
on rows
FROM [Claims Analysis]
WHERE ( [Time].[Q4 2001] )
```

If you need to sort based on multiple string criteria, you can do this by concatenating the strings together. If the strings have different lengths, you can use the VBA Space() function to add spaces to the end of the primary sort string (and the Len() function to help you figure out how many spaces to add) before appending the second string to sort by. If you need to sort based on a mixture of strings and numbers, you will need to convert the numbers to strings and sort based on the resulting string. This means that you should use the VBA Format-Number() function to format the numbers to an equal number of digits before and after the decimal place.

### Reporting on Ranks

Although the various ranking filter functions (TopCount(), BottomCount(), TopSum(), and so on) return members in their ranking order, sometimes a
report needs to return the actual rank number (1, 2, 3, and so on). Furthermore, rank numbers may be necessary sometimes because the members are being returned in another order. For example, salespeople may be ranked both for this year and last year, or the top 10 suppliers in terms of delivery times could have their cost rankings reported as a measure.

Let's examine the case of looking at the top 10 suppliers in terms of delivery time (shorter time is better) and their cost rankings. The [Cost Ranking] will be a calculated member in the query. MDX offers us the Rank() function, which returns the index that a tuple has in a set. This is suited to our purpose because we can sort our suppliers by costs and use that as the set. Rank counts from one, so the first one will be our #1-costing supplier. We don't want to sort the suppliers over and over again, so we'll define a named set to hold the sorted suppliers. Notice that we sort in descending order so the highest-cost supplier is at position #1, and we break the hierarchy so that the order is meaningful:

```
WITH
SET [Cost-Ordered Suppliers] AS
  'Order (
    [Supplier].[Supplier].Members,
    ([Measures].[Total Cost], [Time].[2000]),
    BDESC
  )
',
MEMBER [Measures].[Cost Ranking] AS
  'Rank {
     [Cost-Ordered Suppliers],
     [Supplier].CurrentMember
  }',
SELECT
  { [Measures].[Total Cost], [Measures].[Cost Ranking] } on columns,
  { BottomCount {
      [Supplier].[Supplier].Members,
      10,
      [Measures].[Delivery Time]
    } } on rows
FROM [Purchasing]
WHERE [Time].[Year].[2000]
```

We chose BottomCount() instead of TopCount() because the business problem wanted the top performers, which is opposite of those that have the top times. The supplier with the lowest time appears first in the list.

Note that the cost ranking returned from this query for each supplier is that supplier's rank among all suppliers in the database, not among the set of 10. If we only wanted to take the cost ranking among the 10, then we would re-phrase the query like so:

```
WITH
SET [Delivery-Ordered Suppliers] AS
```

Now, let’s tackle a more demanding yet very real-life query. Let’s say that we need to generate a report which lists the top 10 salespeople according to the year-to-date units sold, their ranking number according to those units, their previous year’s rank, and the difference in units sold between year-to-date and the previous year’s YTD.

We will need to derive the year-to-date units sold to calculate this. We’re also going to take the ranking of our 10 salespeople within an ordered set of all salespeople, so we should name that ordered set. Let’s assume that our time dimension is marked as being a time dimension and that the year level in it is tagged as being a year-typed level, so that we can make use of the YTD() function as well:

WITH

// define our year-to-date units count
MEMBER [Measures].[YTD Units Count] AS
'Sum(YTD(), [Measures].[Units Sold])'

// define a set of ordered salespeople for repeated references
// break the hierarchy, and put the top-valued ones first in the list
SET [Last Year Ordered SalesPeople] AS
'Order ([SalesPerson].[Individual].Members,
  ([Measures].[YTD Units Count], ParallelPeriod ([Time].[Year],1)),
  BDESC
  )'

MEMBER [Measures].[Previous Year Rank] AS
'Rank ([SalesPerson].[Individual].Members,
  [Last Year Ordered SalesPeople],
  [SalesPerson].CurrentMember
  )'

SET [This Year Top 10 SalesPeople] AS
'TopCount ([SalesPerson].[Individual].Members,
  10,'
[Measures].[YTD Units Count]
)
MEMBER [Measures].[This Year Rank] AS
'Rank {
[This Year Top 10 SalesPeople],
[SalesPerson].CurrentMember
}')
MEMBER [Measures].[YTD Units Change] as
'([YTD Units Count] -
{(YSUM+[YTD Units Count], ParallelPeriod ([Time].[Year], 1)))'
SELECT
{ [Measures].[This Year Rank], [Measures].[YTD Units Count],
[Measures].[Previous Year Rank], [Measures].[YTD Units Change] } on columns,
{ [This Year Top 10 SalesPeople] } on rows
FROM Sales
WHERE ([Time].[Month].[Aug. 2000])
Note that the WHERE clause defines the date that the year-to-date accumulates all the values to as August 2000.

### Building a Set That Sums Backward in Time

*How to select a set of times back from some date such that a measure totaled across them reaches some value.*

Another challenge that has shown up in constructing queries is building a set of prior time periods such that their sum of a measure is at least greater than a given quantity. One application of this is creating a set that contains the last time periods required to accumulate 500 new customers from a cube where [New Customer Count] is a measure. (This set could then be used to total up advertising and sign-up costs over that time range.)

From a glance at the TopSum() function, it seems nearly suited to this need, as it returns a set of tuples based on their overall sum meeting or just exceeding a certain value. However, it sorts all of the tuples first in terms of the value being summed, which scrambles them from the point of view of our goal—we want our set of time periods to be in database order.

There are straightforward solutions to this using either straight MDX or an external function call. Actually, the larger the number of time periods being considered, the more efficient an external function call will be compared to straight MDX. We will consider how to do this in Chapter 7 “Extending MDX Through External Functions.”

We can solve this problem with a recursive calculated member that computes the total new customer count backwards in time, which lets us build a named
set that contains the time members where the accumulated value does not exceed the threshold of 500. We will look for the set of time periods up to October 2000. This query is shown in Figure 4.6.

Since we do not know in advance how far back in time to go, we take all months prior to the time that we are counting back from.

The named set [Times Until Sum] contains a set that you can use for a result axis or for further filtering and so on. If you need to include the time period whose associated item value causes the sum to become greater than 500, then some additional logic is required. The simplest way would be to add a test for the calculated member that flags when it reflects the sum becoming greater than or equal to 500:

```
SET [Times Until Sum] AS
'Filter { 
  [Time].[Month].Item(0) : [Time].[Month].[Oct 2000],
  [Measures].[Accum New Count] < 500
  OR ([Measures].[Accum New Count] >= 500
    AND ([Measures].[Accum New Count],
      [Time].CurrentMember.NextMember) < 500
  )
}''
```

The [Times Until Sum] set is defined only for the course of the query. If you wanted to share the set among queries by using CREATE SET to define a named

```
WITH
MEMBER [Measures].[Accum New Count] AS
'Amount { 
[Measures].[Accum New Count]
'}
SET [Times Until Sum] AS
'Filter { 
  [Time].[Month].Members.Item(0) : [Time].[Month].[Oct 2000] },
[Measures].[Accum New Count] <= 500
}''
SELECT
{ [Times Until Sum] } on columns,
{ [Measures].[Advertising Cost], [Measures].[Processing Cost] } on rows
FROM [Subscriptions]
```

**Figure 4.6** Query for counting backwards.
set, you would need to first define the calculated [Accum New Count] member using CREATE MEMBER (or define it at the server).

In terms of efficiency, you should note that [Measures].[Accum New Count] may be calculated up to three times per time member being filtered, with cells for later time periods being recursively referenced through more time members than cells for earlier time periods, so this is even less efficient compared with using an external function. Depending on the number of actual calculations performed, this could be quite tolerable. However, cell calculations in Analysis Services 2000 provide a mechanism for making these more efficient, by calculating the [Accum New Count] in one pass and referencing that pass in Filter() statement.

Drilling Down from Virtual Dimensions

The members of a virtual dimension exist in an N - 1 hierarchical relationship with the members of the real level upon which it is based. However, because the virtual dimension is distinct from the real dimension, you cannot directly drill down on it to the underlying real members. This is, however, something that you might wish to allow in an application if a virtual member’s slice is interesting to the user. We can always directly drill down on the virtual dimension to the underlying real members because the real dimension exists in a cube with the property’s level enabled if the virtual dimension is used in that cube. Furthermore, the name of the member will be identical to the value of the corresponding member property. We simply filter the real dimension based on the property value that was used to create the virtual dimension. The abstract template for this operation is as follows:

\[
\text{Filter( Set Of Real Members On Real Dimension, Member.Properties ("PropertyName") = Virtual Dimension Member.Name)}
\]

Notice that we are comparing the member property values to the Name of the virtual member, not to the UniqueName. The unique name has extra information in it (like the name of the virtual dimension) that isn’t part of the property. For any particular virtual member, this doesn’t give us any extra power. If we know the name of the virtual member, we know the property value to compare it with. (Properties whose values are dates, numbers, and so on, may require a data type conversion because the name is always a text value. Nevertheless, the principle still holds.) Consider the following trivial example. With a product dimension and a virtual dimension whose members are formed out of the product color property, we can select only those products where:

\[
\text{Filter ([Products].[SKU].Members, [Product].Properties ("Flavor") = [ProductSKU].[Zesty Garlic].Name)}
\]
However, we can do some powerful things with sets of virtual members when we are filtering on virtual members that are themselves determined in the course of the query. As an example, suppose we want to see the top-selling three flavors of marinara sauce, and for each of these we want to see in a report the product SKUs nested within them. If we simply cross-joined SKUs within flavors, we wouldn't get the association—we would get a grid with an awful lot of empty cells within it. We would once again use Generate() to provide the iterative framework. The following query would yield the result shown in Figure 4.7:

```plaintext
WITH 
SET [Top3flavors] AS 'TopCount ([ProductSku].[ProductSku].Members, 3, ([Measures].[Unit Sales], [Time].[2000], [Store].[All Stores]))' 
SET [Flavors and Skus] AS 
'Generate ( 
   [Top3Flavors], 
   CrossJoin ( ([ProductSku].CurrentMember ), 
   [Product].[All Products], 
) )' 
```

<table>
<thead>
<tr>
<th>All Of This Flavor</th>
<th>Unit Sales</th>
<th>Dollar Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zesty Garlic</td>
<td>All Products</td>
<td>540,000</td>
</tr>
<tr>
<td>Zesty Garlic</td>
<td>SKU 254</td>
<td>223,000</td>
</tr>
<tr>
<td>Zesty Garlic</td>
<td>SKU 996</td>
<td>205,000</td>
</tr>
<tr>
<td>Zesty Garlic</td>
<td>SKU 223</td>
<td>112,000</td>
</tr>
<tr>
<td>All Of This Flavor</td>
<td>Tomato Alarm</td>
<td>350,000</td>
</tr>
<tr>
<td>Tomato Alarm</td>
<td>SKU 105</td>
<td>180,000</td>
</tr>
<tr>
<td>Tomato Alarm</td>
<td>SKU 099</td>
<td>50,000</td>
</tr>
<tr>
<td>Tomato Alarm</td>
<td>SKU 313</td>
<td>120,000</td>
</tr>
<tr>
<td>All Of This Flavor</td>
<td>Pesto Walnut</td>
<td>All Products</td>
</tr>
<tr>
<td>Pesto Walnut</td>
<td>SKU 291</td>
<td>120,000</td>
</tr>
<tr>
<td>Pesto Walnut</td>
<td>SKU 293</td>
<td>195,000</td>
</tr>
</tbody>
</table>

**Figure 4.7** Virtual dimension drill-down report.
SELECT
  { [Measures].[Unit Sales], [Measures].[Dollar Sales] } on columns,
  { [Flavors and SKUs] } on rows
FROM SalesCube
WHERE ([Time].[2000])

You may expect that you could use the LinkMember() function to provide this
drill-down (see Appendix A if you are unfamiliar with it), but actually, you can’t.
It links one member to another via keys, and each virtual ProductSku member
essentially is a parent for a set of products. There is no key in common between
ProductSku members and Product members.

One Formula Calculating Different Things in Different Places

We have seen that calculated members create slices across all intersections of
all other dimensions. Sometimes, however, we need a calculated member to
give a different result depending on where in the hypercube it is being evalu-
ated. How can we make one calculated member return two different things at
two different places? The answer is to use a conditional function and the vari-
ous properties of members, levels, hierarchies, tuples, and sets in a calculated
member to test for the location where the members are being evaluated, and
then choose the correct formula to use for that place in the cube.

For example, if we wish to create a calculated measure that shows a three-
month moving average of sales, but we don’t want it to show any values for any
level other than month, we can use the Level.Ordinal property to tell us at what
level the formula is being evaluated:

    iif ( 
      [Time].[CurrentMember].[Level].Ordinal = [Time].[Month].Ordinal,
      Avg( LastPeriods (3, [Time].[CurrentMember]), [Measures].[Sales]),
      NULL
    )

Notice that the test for whether or not this formula is being evaluated at the
month level is implemented as a test to determine if the level’s ordinal (depth
from the root level) is equal to the month level’s ordinal. In standard MDX, we
can test to see if the level’s name was “Month,” or we can test for a hard-wired
depth like 2. However, this seems to capture the best combination of efficiency
(comparing numbers instead of strings) and grace (we know what [Time].[Month] refers to, whereas 2 would not be as easily understood). This
example has a maintenance issue as well: If the dimension changed and a new
level was inserted between the month level and the root, then the 2 would need
to be updated to a 3. Using Microsoft’s extensions to MDX, we can also use the
IS statement to test for equivalence of object identity, as with the following:
This formulation should not only be efficient, but will also be graceful to changes in the set of levels in the time dimension (so long as the Month level does not disappear.)

Earlier in this chapter, we showed an example of how to spread quarter-level input values down to lower time levels based on the number of time periods in the quarter. The calculated measure that resulted would only show values at or below the quarter level and result errors above it. We can use the same sort of testing for level ordinals to create one calculated measure that shows aggregated values above the month level and allocated values below the month level:

```
CREATE MEMBER [Competitor Earnings Rate] AS
  'iif ([Time].[CurrentMember].[Level].Ordinal < [Time].[Quarter].Ordinal,
      [Measure].[Earnings],
      ([Measure].[Earnings] / Descendants (Ancestor (  
        [Time].CurrentMember, [Time].[Quarter]),  
        [Time].CurrentMember. Level, SELF).Count)
  )'
```

Sometimes, you may find that you need to use a different formula depending upon what member you are evaluating a calculated member at. For example, a multinational corporation may use a slightly different net profit calculation depending upon the country in which it is performing the calculation. For our case, we can use the unique name or the name of a member. The following example demonstrates three different profit formulas, two of which are specific to a single country as exceptions to the general rule:

```
iif ([Geography].CurrentMember IS [Geography]."[Japan]",
    [Sales] - [Deductible Expenses 1],
    iif ( [Geography].CurrentMember IS "[Geography].[Malaysia]",
        [Sales] - [Deductible Expenses 2],
        [Sales] - [Deductible Expenses 3]
    )
)
```

Although the MDX specification provides the CASE ... WHEN ... construct, Microsoft’s OLAP products do not support it, so we need to resort to nested IF-THEN logic using iif().

If you are using IF-THEN logic to specify a constant value based on member name, you might also think about putting that information into a table and using it either as a member property or as a measure. The less conditional the logic in a formula, the easier it is to comprehend and maintain over time. The numbers
that you are specifying might be exceptions to a general rule, for example, commission rates to charge based on product. In that case, the logic of the formula would test for the special commission being empty or not and to use the special commission value if it is present or the default value if the special commission is absent.

Three types of tests you may wish to perform on the current member are

- Is it at a particular member or members?
- Is it at a particular level or levels? Is it at the leaf level?
- Is it at a descendant of a particular member?

To test whether the current member is at a particular level (or above or below a particular level, including the level of an arbitrary named level), compare its level with the target level using IS. Compare the current member's level ordinal to the ordinal of the level you are interested in using level.Ordinal. To test whether the current member is a descendant of a particular member, you can check the name of the current member's ancestor at that level with the target ancestor's name, as with the following expression:

\[ \text{Ancestor( [Dimension], Member).UniqueName = Ancestor-Member.UniqueName} \]

The six standard functions you can use against the current member or its related information for this purpose are

- Member.Level.Ordinal
- Member.Level.Name
- Member.Level.UniqueName
- Member.Name
- Member.UniqueName
- Member.Properties ("Property Name")

Microsoft extends the set of tests you can use with the following:

- IsAncestor(target-member, source-member)
- IsGeneration(member, generation-number)
- IsLeaf(member)
- IsSibling(member1, member2)

They work as shown in Table 4.1.

Also, \[ \text{Rank( [Dimension].CurrentMember, \{ member1, member2, \ldots \ memberN \} ) > 0} \] can be used to test for whether or not the current member is in the given set. Using IsGeneration() is a bit complicated and requires an understanding of how generation numbers are assigned; please refer to its discussion in Appendix A.
When you are testing for member equivalence in Analysis Services, you should favor the `dimension.CurrentMember IS [Specific Member]` construct. This is because unique names can have many different formats and you are not always in control of the format chosen. We talk about name construction in both Chapter 2 “MDX in More Detail,” and Chapter 8 “Building Queries in MDX.”

### Logical Aggregations

You may want some queries to return a result that is essentially a logical aggregation: Are there any cells at lower levels where some condition is true? Is it

<table>
<thead>
<tr>
<th>EXTENSION?</th>
<th>EXPRESSION</th>
<th>PURPOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>dimension.Level.Ordinal = Named_Level.Ordinal</code></td>
<td>tests for being at level</td>
</tr>
<tr>
<td></td>
<td><code>dimension.Level.Ordinal &gt; Named_Level.Ordinal</code></td>
<td>tests for being below level</td>
</tr>
<tr>
<td></td>
<td><code>dimension.Level.Ordinal &lt; Named_Level.Ordinal</code></td>
<td>tests for being above level</td>
</tr>
<tr>
<td></td>
<td><code>dimension.Currentmember.Uniquename = [Specific Member].Uniquename</code></td>
<td>tests for member equivalence</td>
</tr>
<tr>
<td></td>
<td><code>dimension.UniqueName = &quot;Unique Name&quot;</code></td>
<td>tests for arbitrary condition on property; can be applied to this purpose</td>
</tr>
<tr>
<td>*</td>
<td><code>dimension.CurrentMember IS [member]</code></td>
<td>Tests for member equivalence</td>
</tr>
<tr>
<td>*</td>
<td><code>dimension.CurrentMember.Level IS [level]</code></td>
<td>Tests for being at level</td>
</tr>
<tr>
<td>*</td>
<td><code>IsAncestor ([member], [dimension].CurrentMember)</code></td>
<td>Tests for current member being descendant of specific member</td>
</tr>
<tr>
<td>*</td>
<td><code>IsLeaf ([dimension].CurrentMember)</code></td>
<td>Tests for current member being at the leaf level</td>
</tr>
<tr>
<td>*</td>
<td><code>IsSibling([dimension].CurrentMember, [Specific Member])</code></td>
<td>Tests for current member being sibling of target member</td>
</tr>
</tbody>
</table>
true at every cell? True at none? Essentially, these would be logical OR and AND operators that take cell ranges as inputs rather than discrete cells. For example, we could look at a report of states by quarters and see if sales were below some threshold at any related store for any related week.

Although MDX doesn’t provide a direct way to do this, we can do it indirectly and get some useful side benefits along the way. We can count the cells for which the condition is true and compare that number with the number of cells in the range. Counting the cells itself is done indirectly too. The Count() function only counts cells, but we can sum an expression over the cells, where the expression returns one if the expression is true, and zero otherwise. We know that the condition is true for some location if the sum is greater than zero, and we know it’s true everywhere if the sum is equal to the count of cells in the range. Using the rules of logic, we also know it’s true everywhere if its opposite condition has a sum of zero. A useful analytical byproduct of this approach is that we can say what percentage of the time the condition is true by taking the ratio of the sum to the count of cells in the range.

For example, the following would be a query to obtain the count of stores per week where the profit was less than half of the year’s average for stores in that province of Canada and for all Canada’s provinces and cities in 1999:

```mdx
WITH
    MEMBER [Measures].[Avg Profit] AS
        '[Measures].[Profit] / [Measures].[Units Sold]
    MEMBER [Measures].[Condition Count] AS
        'Sum (Descendants ([Geography], [Geography].[Store], SELF),
             Sum (Descendants ([Time], [Time].[Week], SELF),
                 iif ([Measures].[Profit] <
                     ([Measures].[Avg Profit], Ancestor ([Time], [Time].[Year]),
                     Ancestor ([Geography], [Geography].[Province])),
                     1, 0))
        )
    )
SELECT
    CrossJoin ({ [Time].[Quarter].Members }, { [Measures].[Profit],
          [Measures].[Condition Count] }) on columns,
    { Descendants ([Geography].[Canada], [Geography].[City],
                  SELF_AND_BEFORE) } on rows
FROM Sales
WHERE ([Time].[1999])
```

**Including All Tuples with Tied Ranking in Sets**

Following from the preceding example, if you’re truly trying to be fair to your salespeople and it’s possible that two or more are tied for tenth place, then you need to apply more MDX. The trick is simply to add to the set all tuples (sales-
people in this case) whose values are equal to the 10th-place salesperson, and unite this filtered set with the ranked set. Because MDX preserves the ordering in the sets when they are united, the final set that is created will be in the correct order:

WITH
MEMBER [Measures].[YTD Units Count] AS
'\texttt{SUM(YTD(), [Measures].\[Units Sold\])}'
SET [Last Year Ordered SalesPeople] AS
'Order {
  [SalesPerson].\[Individual\].Members,
  ([Measures].[YTD Units Count], ParallelPeriod ([Time].[Year],1)),
  BDESC
}'
MEMBER [Measures].[Previous Year Rank] AS
'Rank {
  [Last Year Ordered SalesPeople],
  [SalesPerson].CurrentMember
}'
SET [This Year Top 10 SalesPeople] AS
'TopCount {
  [SalesPerson].\[Individual\].Members,
  10,
  [Measures].[YTD Units Count]
}'

/* We add some new set declarations */

// collect all salespeople whose units are equal to the 10th's
SET [This Year Tied SalesPeople] AS
'Filter {
  [SalesPerson].\[Individual\].Members,
  [Measures].[YTD Units Count] =
  ([Measures].[YTD Units Count],
  [This Year Top 10 SalesPeople].Item {
    [This Year Top 10 SalesPeople].Count - 1
  })
}'

// Put the two sets together, and eliminate the duplicated 10th member
SET [Top Ranked SalesPeople] AS
'Union {
  [This Year Top 10 SalesPeople],
  [This Year Tied SalesPeople]
}'

/* Done adding the set declarations */

MEMBER [Measures].[This Year Rank] AS
'Rank (
MEMBER [Measures].[YTD Units Change] as
    
    \([\text{YTD Units Count}] - \\
    (\{\text{YTD Units Count}, \text{ParallelPeriod} ([\text{Time}].[Year], 1)\})\)

SELECT
    { [Measures].[This Year Rank], [Measures].[YTD Units Count],
    [Measures].[Previous Year Rank], [Measures].[YTD Units Change]
} on columns,
    { [Top Ranked SalesPeople] } on rows
FROM SALES
WHERE ([Time].[Month].[Aug. 2000])

But how do we get the rank numbers right? If three salespeople are in the 10th position, then one of them will be ranked 11th and another 12th. Also, if two of them share the same second-ranked number of units sold, one will have a rank number of 2 and another will have a rank number of 3 (as opposed to both of them having a rank number of 2 or 2.5). In standard MDX, this is quite complicated. We had worked out an example of this for an early draft of this book. However, for those of you that are using Microsoft Analysis Services, a direct solution is actually built in. The Rank() function now takes an optional third argument that specifies an ordering value for the set. If it is supplied, Analysis Services uses the values it supplies to determine tied rankings within the set. (Without it, it looks for the position of the tuple within the set, which has no information that can be used to determine ties.) We can modify the definitions of the measures, the text of which we provide in Figure 4.8 and the results of which are shown in Figure 4.9.

We can do the fair ranking in an external function, but the external function will be called once for every [This Year Rank] and once for every [Previous Year Rank]. Only 10 members will be passed into it small for [This Year Rank] and it will only be called 10 times, so for [This Year Rank], performance should be good. For [Previous Year Rank], the entire set of salespeople will be passed into the external function; as that number climbs, the performance feasibility drops and it becomes much more desirable to implement the calculation in some combination of client logic with MDX.

A Sample Analysis

MDX is rich enough to support advanced analytical queries. Indeed, once you have created a database schema for Microsoft OLAP Services and populated it with data, MDX provides many of the tools you need to support complex analyses within queries. We will spend the next several pages rolling up our sleeves and using MDX to explore a useful set of questions related to the topic. What are the important products to our important customers? Although your
MEMBER [Measures].[Previous Year Rank] AS
'Rank ( [Last Year Ordered SalesPeople], [SalesPerson].CurrentMember,
([Measures].[YTD Units Count], ParallelPeriod ([Time].[Year],1))
''

MEMBER [Measures].[This Year Rank] AS
'Rank ( [This Year Top 10 SalesPeople],
[SalesPerson].CurrentMember,
[Measures].[YTD Units Count]
''

---

**Figure 4.8**  Query that results in fair rankings for this year and last.

<table>
<thead>
<tr>
<th>Salesperson</th>
<th>This Year Rank</th>
<th>YTD Units Count</th>
<th>Last Year Rank</th>
<th>YTD Units Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nishi Agrawal</td>
<td>1</td>
<td>45,000</td>
<td>4</td>
<td>2,000</td>
</tr>
<tr>
<td>Sarah Mayer</td>
<td>2</td>
<td>44,020</td>
<td>1</td>
<td>-100</td>
</tr>
<tr>
<td>Pat Bates</td>
<td>3</td>
<td>42,500</td>
<td>6</td>
<td>10,000</td>
</tr>
<tr>
<td>Ian Walter</td>
<td>3</td>
<td>42,500</td>
<td>15</td>
<td>22,000</td>
</tr>
<tr>
<td>Rama Lahori</td>
<td>5</td>
<td>40,000</td>
<td>3</td>
<td>-4,500</td>
</tr>
<tr>
<td>Kate Higgins</td>
<td>6</td>
<td>30,100</td>
<td>9</td>
<td>10,000</td>
</tr>
<tr>
<td>Levi Chen</td>
<td>7</td>
<td>22,000</td>
<td>22</td>
<td>6,000</td>
</tr>
<tr>
<td>Joe Bergman</td>
<td>7</td>
<td>22,000</td>
<td>16</td>
<td>5,500</td>
</tr>
<tr>
<td>Larry Levy</td>
<td>9</td>
<td>16,500</td>
<td>11</td>
<td>2,000</td>
</tr>
<tr>
<td>Joanne Luongo</td>
<td>10</td>
<td>15,000</td>
<td>10</td>
<td>-150</td>
</tr>
<tr>
<td>Ted Johnson</td>
<td>10</td>
<td>15,000</td>
<td>12</td>
<td>250</td>
</tr>
</tbody>
</table>

**Figure 4.9**  Results of fair ranking query.
applications may not be related to customers and products, a generalization of the analysis we discuss is “What are the important factors to each of a set of interesting members?” This is a form of data mining, even though we are performing it in an OLAP tool. Within the framework of this analysis, we will explore set construction and the use of the Generate() and Extract() functions, and make extensive use of named sets.

The dimensions of primary interest in this analysis are customer, product, and time. A familiar simple OLAP query to use as a starting point is “Who are our best customers?” If we are interested in the top 10 customers in terms of total profitability in 2001, we may start with the set defined by the following:

\[
\text{TopCount} \left( [\text{Customer}].[\text{Individual}].\text{Members}, 10, \right. \\
\left. ([\text{Measures}].[\text{Profit}], [\text{Time}].[2001]) \right)
\]

A query that shows profitability for 2000 and 2001 for these customers is as follows:

\[
\begin{align*}
\text{SELECT} \\
&\{ [\text{Time}].[2000], [\text{Time}].[2001] \} \text{ on columns,} \\
&\{ \text{TopCount} \left( [\text{Customer}].[\text{Individual}].\text{Members}, 10, \right. \} \\
&\left. ([\text{Measures}].[\text{Profit}], [\text{Time}].[2001]) \right\} \text{ on rows} \\
\text{FROM Sales} \\
\text{WHERE} \{ [\text{Measures}].[\text{Profit}] \}
\end{align*}
\]

Now, this query is useful in its own right, but we want to go a bit deeper and learn about the products that these customers are buying. We can learn about the top three product brands (in terms of profit) that some customers are buying with the following expression:

\[
\text{TopCount} \left( [\text{Product}].[\text{Brand}].\text{Members}, 3, \right. \\
\left. ([\text{Measures}].[\text{Profit}], [\text{Time}].[2001]) \right)
\]

Because this is along a separate dimension, we can put it on a different query axis than the customer dimension, or we can put it on the same axis as the customers with CrossJoin(). For example:

\[
\begin{align*}
\text{WITH} \\
\text{SET} [\text{Top3Prods}] \text{ AS} \\
\{ \text{TopCount} \left( [\text{Product}].[\text{Brand}].\text{Members}, 3, \right. \} \\
\left. ([\text{Measures}].[\text{Profit}], [\text{Time}].[2001]) \right\}
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} \\
&\{ \text{CrossJoin} \left( [\text{Top3Prods}], \left( [\text{Time}].[2000], [\text{Time}].[2001]) \right) \} \text{ on columns,} \\
&\{ \text{TopCount} \left( [\text{Customer}].[\text{Individual}].\text{Members}, 10, \right. \} \\
&\left. ([\text{Measures}].[\text{Profit}], [\text{Time}].[2001]) \right\} \text{ on rows} \\
\text{FROM Sales} \\
\text{WHERE} \{ [\text{Measures}].[\text{Profit}] \}
\end{align*}
\]

However, this still doesn’t tell us about the products that these top customers are buying. Regardless of the way products are oriented in the query or whether the product set is evaluated as a named set or within the body of the SELECT clause, the context in which the product set is evaluated is going to be at the All-
customer member because that is the default member in the query context. What we want is the set of products that these customers are buying. We can explore a couple of different paths for determining this.

One path is to ask about the top three product categories over the entire set of top 10 customers. To do this, we need to calculate profitability across the top 10 customers. Calculating the top 10 customers from the leaf level of a fairly large dimension takes significant CPU time because of the sorting involved, so we should use a named set to hold the result customer set. We can then sum across that directly or use a calculated member to hold the result:

```
WITH
SET [Top10Custs] AS
'{ TopCount ([Customer].[Individual].Members, 10,
([Measures].[Profit], [Time].[2001])) }'
MEMBER [Measures].[Top10profit] AS
'Sum ([Top10Custs], [Measures].[Profit])'
SET [Top3Prods] AS
'{ TopCount ([Product].[Brand].Members, 3,
([Measures].[Top10profit], [Time].[2001])) }'
SELECT
{ CrossJoin ([Top3Prods], {[Time].[2000], [Time].[2001]}) on columns,
{ TopCount ([Customer].[Individual].Members, 10,
([Measures].[Profit], [Time].[2001])) } on rows
FROM Sales
WHERE ( [Measures].[Profit] )
```

This helps us out. We are now looking at the top three products of the top 10 customers, which gives us a better picture of those products and customers in particular. Although these customers could be a fairly homogeneous group, each of them may be fairly different from the others. (They also may be pretty different from the average customer in terms of product preferences; we will explore that later in this section as well.) In terms of our individual customers’ favorite product mixes, we can get even more precise than we have, but the MDX requires a quantum leap in sophistication.

Exploring possibilities that don’t give us what we are looking for helps us understand the MDX that gives us what we need. MDX’s tuple orientation enables us to take the top N tuples from a set from multiple dimensions. However, taking the top N tuples from a customer-product set, by cross-joining customers and products together, won’t give us what we are looking for. We are interested in 10 customers and three products for each customer. However, the top 30 customer-product combinations (TopCount (CrossJoin ( ... ), 30, ... ) could be dominated by 30 products sold to the most profitable customer. The top 30 customer-products of the top 10 customers aren’t any better. Taking the top three products of the top 10 customers involves somehow breaking it up among the customers, instead of cross-joining customer and product sets. The MDX function that enables us to do this is Generate().
If we are going to create a set named Top3ProdsOf10Custs, the basic template for this is going to be:

```
WITH
SET [Top10Custs] AS
  '{ TopCount ([Customer].[Individual].Members, 10,
    ([Measures].[Profit], [Time].[2001])) }'

SET [Top3ProdsOf10Custs] AS
  '{ Generate ([Top10Custs], . . . )))'
```

Within the Generate() function, for each tuple in the set Top10Custs we want to find the top three products. The following looks tempting:

```
Generate ([Top10Custs], TopCount ([Product].[Brand].Members, 3,
  ([Measures].[Profit], [Customer].CurrentMember, [Time].[2001])))
```

But even that will not get us there. (The [Customer].CurrentMember is completely superfluous and is only included to clarify our interest in the operation. The current member of a dimension does not need mentioning in this context unless we are going to modify it or access something related to it.) It will get us partway there, but when we put it into a query, the total result (as shown in Figure 4.10) isn’t what we want:

```
WITH
SET [Top10Custs] AS
  '{ TopCount ([Customer].[Individual].Members, 10,
    ([Measures].[Profit], [Time].[2001])) }'
SET [Top3ProdsOf10Custs] AS
  '{ Generate ([Top10Custs], TopCount ([Product].[Brand].Members, 3,
    ([Measures].[Profit], [Time].[2001]))) }'
SELECT
  { [Time].[2000], [Time].[2001] } on columns,
  { [Top3ProdsOf10Custs] } on rows
FROM Sales
WHERE ( [Measures].[Profit] )
```

<table>
<thead>
<tr>
<th></th>
<th>2000</th>
<th>2001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gobi Crab Cakes</td>
<td>25,000</td>
<td>28,400</td>
</tr>
<tr>
<td>Silver Scales</td>
<td>24,500</td>
<td>26,900</td>
</tr>
<tr>
<td>Poseidon’s Platter</td>
<td>21,100</td>
<td>19,000</td>
</tr>
</tbody>
</table>

Figure 4.10  Results of first try.
Our Generate() clause returned the right products but no customers. How do we get the customers in? The Generate() function returns a set with the dimensionality of the second set, not the first set. We cannot express

\[
\text{Generate(} \{ \text{Top10Custs}\},
\{ \text{Customer} . \text{CurrentMember} , \text{TopCount} \{ \{\text{Product} . \text{[Brand]} . \text{Members} \}, 3,
\{ \{\text{Measures} . \text{[Profit]} , \text{Time} . \text{[2001]}\}\}\}\)
\]

because that is syntactically and semantically illegal (we are mixing customer and product members in a single set). Using the following query we can, however, combine the customer member with the product set using the CrossJoin() function, which gives us the customer by product tuples that we want:

\[
\text{Generate (} \{ \text{Top10custs}\},
\text{CrossJoin (}
\{ \text{Customer} . \text{CurrentMember } \},
\text{TopCount \{ } \{\text{Product} . \text{[Brand]} . \text{Members} \}, 3,
\{ \{\text{Measures} . \text{[Profit]} , \text{Time} . \text{[2001]}\}\}\)
\)
\]

At last, we have the tuples that we want (see Figure 4.11). Note that in this last version of the query, we needed to reference the current member of the customer dimension in order to have the result set contain both customers and products.

Analytically, this particular result is useful for the fairly arbitrary cutoffs that we chose to define important products for important customers. Importance is defined for this example as importance to our organization (we are looking at

<table>
<thead>
<tr>
<th></th>
<th>2000</th>
<th>2001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hudson Food Dists.</td>
<td>Gobi Crab Cakes</td>
<td>1,200</td>
</tr>
<tr>
<td>Hudson Food Dists.</td>
<td>Silver Scales</td>
<td>1,400</td>
</tr>
<tr>
<td>Hudson Food Dists.</td>
<td>Poseidon’s Platter</td>
<td>1,100</td>
</tr>
<tr>
<td>Barbara, Levin Inc.</td>
<td>Gobi Crab Cakes</td>
<td>1,120</td>
</tr>
<tr>
<td>Barbara, Levin Inc.</td>
<td>Briny Deep</td>
<td>1,040</td>
</tr>
<tr>
<td>Barbara, Levin Inc.</td>
<td>Silver Scales</td>
<td>1,200</td>
</tr>
</tbody>
</table>

...     ...     ...     ... 

Figure 4.11  Results of desired query.
profitability, not units, revenue, value of long-term contracts, and so on). The thresholds are very arbitrary because we are choosing the top N products and customers. Choosing sets of customers and products based on their percentage contribution to profits, revenues, and the like would be a less arbitrary choice, and these are equally easy. For example, for the set of customers that form our most profitable 10 percent, what are the most profitable 20 percent of the products they each buy? Replacing the TopCount() function in our expression with TopPercent() gives us the following:

```mdx
SET [Top 10% of Custs] AS
' ( TopPercent ([Customer].[Individual].Members, 10,
   ([Measures].[Profit], [Time].[2001]) ) )
SET [Top 20% Prods Of Top 10% Custs] AS
' ( Generate ( [Top 10% of Custs],
   CrossJoin ( { [Customer].CurrentMember },
   TopPercent ([Product].[Brand].Members, 20,
   ([Measures].[Profit], [Time].[2001])
   )
   )
 )
''
```

This can lead us to several other useful related queries. For example, this expression gives us the products per customer. If our goal is to then focus on the production, distribution, or pricing of these products, we may be interested in removing the specific customers and looking just at the products. Each customer is likely to have products in common with other customers. In fact, it is not immediately obvious how many different products are in this group. The set of customers is unknown in advance of the query. How can we find out the number of products or the set of products?

The answer can be found by using the MDX Extract() function, which returns a set of selected dimensionality. We can use it to return the unique set of products from our selected customer by product tuples, as with the following expression:

```mdx
SET [Top 10% ’s Top 20 Products] AS
' Extract ([Top 20% Prods Of Top 10% Custs], [Product])
```

Extract() returns only unique tuples, so we don’t have to worry about finding duplicates. The products will be in a fairly arbitrary order, and we can sort them further if we want to. We can also take this set and count the tuples to find out how many products make up this group.

If our goal is to understand how these customers are or are not representative of our customers as a whole, we may want to compare the product mix purchased by the top 10 percent of customers with the product mix purchased by the average customer. Which brands that are in the top 20 percent by prof-
Itability for our top 10 percent of customers are also in the top 20 percent of profitability for all customers? Which ones are peculiar to the top customers? Which ones, if any, are among the most profitable products for all customers, but not among the most profitable products for our top customers? If we are exploring customers and products by profitability, these are also important questions.

This last set of three questions could be answered in one query, and we will continue our train of thought to create it. We will make one creative leap to put it together. The goal is to create a grouping of products into three different groups: those that correspond to only the top 10 customers, those which correspond to both the top 10 group and across all customers, and those that correspond only across all customers and not within the top 10. These three groups represent customer populations, and we will use three calculated members on the customer dimension to represent these groups. (If no products exist in one of these three groups, we will not have any product-customer group tuples that use that customer group, and that group will not appear in the query result.)

We need to construct two basic sets and then manipulate them to get the third. The first set, of top products for top customers, we have already created as [Important Products]. The second set, of top products across all customers, is similar:

\[
\text{SET } \text{[Top 20\% Prods Across All Custs]} \text{ AS}
\]
\[
\text{'TopPercent } (\text{[Product].[Brand].Members, 20, (\text{[Measures].[Profit], [Time].[2001]})}]
\]

The trick now is to create three divisions between those two sets. We need to pool them together before dividing them up, using the following expression:

\[
\text{SET } \text{[Product Union]} \text{ AS 'Union } ([\text{Top 20\% Prods Across All Custs}], [\text{Top 10\%'s Top 20 Products}]')
\]

We could also create the same effect with the following expression:

\[
\text{Distinct } (\{\text{Top 20\% Prods Across All Custs}, [\text{Top 10\%'s Top 20 Products}] \})
\]

Now, we simply create three subsets using set functions in sequence:

\[
\text{SET } \text{[Top10 Only]} \text{ AS}
\]
\[
\text{'Except } (\text{[Product Union]}, \text{[Top 20\% Prods Across All Custs]})'
\]

\[
\text{SET } \text{[Both Groups]} \text{ AS}
\]
\[
\text{'Intersect } (\text{[Top 10\%'s Top 20 Products]}, \text{[Top 20\% Prods Across All Custs]})'
\]

\[
\text{SET } \text{[All Customers Only]} \text{ AS}
\]
\[
\text{'Except } (\text{[Top 20\% Prods Across All Custs]}, \text{[Product Union]})'
\]

The last step is to create the calculated members that will group these three subsets. “Calculated members” implies computation; what formula calculates
the cells within these sets without altering the contents of the cells?

We know that we want to use some sort of default member for this calculation. These members are on the Customer dimension, so a formula of \([\text{Customer}]. [\text{All Customer}]\) makes sense. That formula causes the values of each of the products to be taken from the [All Customer] member for whatever measure is being calculated. So, the three calculated members can each have a very simple formula:

\[
\begin{align*}
\text{MEMBER} & \ [\text{Customer}]. [\text{Top 10\% Only Group}] \ \text{AS} \ ' [\text{Customer}]. [\text{All Customer}]' \\
\text{MEMBER} & \ [\text{Customer}]. [\text{Top 10\% And All Group}] \ \text{AS} \ ' [\text{Customer}]. [\text{All Customer}]' \\
\text{MEMBER} & \ [\text{Customer}]. [\text{All Customers Only Group}] \ \text{AS} \ ' [\text{Customer}]. [\text{All Customer}]'
\end{align*}
\]

And we can create our final set of tuples for reporting on as follows:

\[
\begin{align*}
\text{SET} & \ [\text{Report Tuples}] \ \text{AS} \ ' \\
& \ \text{CrossJoin} ( \ { [\text{Customer}]. [\text{Top 10\% Only Group}] }, \ [\text{Top10 Only}] ), \\
& \ \text{CrossJoin} ( \ { [\text{Customer}]. [\text{Top 10\% And All Group}] }, \ [\text{Both Groups}] ), \\
& \ \text{CrossJoin} ( \ { [\text{Customer}]. [\text{All Customers Only Group}] }, \ [\text{All Customers Only}] )'
\end{align*}
\]

When we put it all together, it forms the following (long) query, whose results are shown in Figure 4.12:

\[
\begin{align*}
\text{WITH} \\
\text{SET} & \ [\text{Top 10\% of Custs}] \ \text{AS} \ ' \\
& \ \text{TopPercent} ([\text{Customer}]. [\text{Individual}]. Members, 10, \ ( [\text{Measures}]. [\text{Profit}], [\text{Time}]. [2001])')) \\
\text{SET} & \ [\text{Top 20\% Prods Of Top 10\% Custs}] \ \text{AS} \\
& \ \text{Generate} ( \ [\text{Top 10\% of Custs}], \\
& \ \text{CrossJoin} ( \ { [\text{Customer}]. \text{CurrentMember}}, \\
& \ \text{TopPercent} ( [\text{Product}]. [\text{Brand}]. Members, 20, \\
& \ \text{[Measures]}. [\text{Profit}], [\text{Time}]. [2001])
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>2000</th>
<th>2001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top 10% Only Group</td>
<td>Gobi Crab Cakes</td>
<td>25,000</td>
</tr>
<tr>
<td>Top 10% Only Group</td>
<td>Silver Scales</td>
<td>24,500</td>
</tr>
<tr>
<td>Top 10% And All Group</td>
<td>Poseidon's Platter</td>
<td>21,100</td>
</tr>
<tr>
<td>Top 10% And All Group</td>
<td>Mako Steak-o</td>
<td>18,300</td>
</tr>
<tr>
<td>All Customers Only Group</td>
<td>Atlantic Trench Mouthfuls</td>
<td>18,100</td>
</tr>
</tbody>
</table>

Figure 4.12  Results of full query.
Whew! Although this is an involved query, it is not as complex as it would have been if we had tried to perform the same operation in SQL against the original source tables!

Summary

Although we have discussed a number of topics in this chapter, we have used only a fraction of the MDX functions available. It would not be possible to exhaustively cover all applications and MDX techniques in any one book. Hopefully, this chapter has given you all of the conceptual tools you need to understand how MDX functions as a language and how Microsoft OLAP Services and MDX work together so you can construct as sophisticated an application as your situation requires.