CHAPTER 6

Sorting and Ranking in MDX

A very common requirement for analytical applications is providing sorted and ranked views of data. Users are always interested in Top-10 lists, and often in seeing who or what was #1 last month or year. We'll cover the basics of sorting and ranking in this chapter, and also look at some ins and outs for more advanced kinds of sorting and ranking calculations.

Functions used in this chapter are `Order()`, `Hierarchize()`, `TopCount()`, `BottomCount()`, `TopSum()`, `BottomSum()`, `TopPercent()`, `BottomPercent()`, `YTD()`, `CoalesceEmpty()`, `Rank()`, `ParallelPeriod()`, `Generate()`, `Ancestor()`, `Descendants()`, `.Properties()`, `DrillDownLevelTop()`.

The Function Building Blocks

We took a brief look at the `Order()` function in Chapter 1, so we've seen one building block already. The full set of functions provided for sorting and ranking purposes are as follows:

FUNCTION	PURPOSE
`Order()`	Sort a set of tuples
`Hierarchize()`	Sort a set of tuples into hierarchical order.
`TopCount()`	Select the top *N* tuples of a set.
`BottomCount()`	Select the bottom *N* tuples of a set.
`TopSum()`	Select the top tuples of a set whose sum meets a threshold.
`BottomSum()`	Select the bottom *N* tuples of a set whose sum meets a threshold.
`TopPercent()`	Select the tuples of a set corresponding to the top *N*% of all in set.
`BottomPercent()`	Select the tuples of a set corresponding to the bottom *N*% of all in set.
`Rank()`	Find the ordinal position of a tuple in a set.

Note that all of these functions operate on sets of tuples, not just one-dimensional sets of members. The following are additionally provided, and are intended to help support GUI interaction, although any MDX-based application may use them. (They are only mentioned briefly within the chapter.)

`DrillDownLevelTop()`	Drills down on members of a specified level in a set; adds only the top-*N* members.
`DrillDownLevelBottom()`	Drills down on members of a specified level in a set; adds only the bottom-*N* members.
`DrillDownMemberTop()`	Drills down on specified members in a set; adds only the top-*N* children.
`DrillDownMemberBottom()`	Drills down on specified members in a set; adds only the bottom-*N* children.

Classic Top-*N* Selections

The `TopCount()` function is what we would use to create a classic Top-10 list (or Top-*N*). Its syntax is as follows:

```
TopCount (set [ , numeric_expression ] )
```

The optional *numeric*_expression gives the number of tuples to take, and it returns up to that many from the *set*, in order from largest to smallest. Here is a query to return the top 5 product subcategories for the whole quarter in the Northeast, and the results are shown in Figure 6-1:

```
SELECT
{ [Measures].[Dollar Sales] } on columns,
TopCount (
  [Product].[Subcategory].Members,
  5,
  [Measures].[Dollar Sales]
) on rows
FROM [Sales]
WHERE ([Q2, 2005], [Northeast])
```

Note that the members are returned in order from largest Dollar Sales to smallest.

Ranking queries often require a total across the top *N*, and also a total of all others, so you can see how the top 10 performed as a group and compare to all others not in the group. This is a perfect case to use a named set, so we only run the ranking operation once and use its results multiple times. Let's update the last query to incorporate a set total and an all-other total. The fastest way to calculate the all-other total is not to figure out the set of all-other members and total across that, but rather to subtract the top-10 group total from the `[All Product]` total, which was automatically aggregated. (Remember to leverage precalculated aggregates wherever you can!)

```
WITH
SET [TopSelection] AS
'TopCount (
  [Product].Levels(2).Members,
  5,
  [Measures].[Dollar Sales]
)'
MEMBER [Product].[Total] AS
'Sum ([TopSelection])'
MEMBER [Product].[All Others] AS
'[Product].[All Product] - [Product].[Total]'
SELECT
{ [Measures].[Dollar Sales] } on columns,
{
[TopSelection],
[Product].[Total], [Product].[All Others]
} on rows
FROM WM2005.Sales
WHERE [Q2, 2004]
```

	Dollar Sales
Baseball	29,930.05
Coolers	28,994.72
Foot Massagers, Spas	27,085.48
CD Players	24,589.87
Upgrades	23,941.65

Figure 6-1 Top-5 query result.

The standard behavior for `TopCount()` is to always return as many tuples as you specify, unless the input set had fewer to begin with. Suppose that the set is larger than *N*, but you have fewer associated data cells that had any data in them. The topmost tuples will be associated with data, while the remaining ones will have no data associated with them but still be in the set. Users often don't want to see the Top-10 list containing 10 items if only 8 had data and the other 2 are arbitrary blank choices. They'd rather see the list trimmed down to just the eight that had data. So, unless you're confident the data is dense enough, you may want to filter the data set. The standard means to do this is to rank on the results of a filter:

```
TopCount ( Filter ( the set, Not IsEmpty (criterion ) ), 10, criterion)
TopCount ( Filter ( the set, criterion <> 0 ) ), 10, criterion)
```

In Analysis Services 2005 and 2000, you may get much better performance with the following, so long as you don't have real zero values that you also want to remove:

```
TopCount ( NonEmptyCrossJoin(the set, criterion-measure), 10, criterion)
TopCount ( NonEmpty(the set, criterion-measure), 10, criterion)
```

	Dollar Sales
Baseball	29,930.05
Coolers	28,994.72
Foot Massagers, Spas	27,085.48
CD Players	24,589.87
Upgrades	23,941.65
Total	134,541.77
All Others	1,637,019.83

Figure 6-2 Top-5 query result with group total and "All-Others" total.

MISSING VALUES IN ESSBASE SORTING AND RANK SELECTIONS

By default, Essbase strips tuples with associated missing values from a set during sorting and ranking operations. To include these, you would need to use the `CoalesceEmpty()` function to put them in your preferred place. The following would virtually guarantee them to be at the end:

```
TopCount ( the set, CoalesceEmpty( criterion, -1.0e+38))
```

Because this works on tuples, you can use the following to take the top 100 brand-channel combinations in terms of year-to-date units sold:

```
TopCount (
  CrossJoin(
    [Product].[Brand].Members,
    [Channel].[Channel ID].Members
  ),
  100,
  Sum (
    YTD(),
    [Measures].[Unit Sales]
  )
)
```

Adding Ranking Numbers (Using the Rank() function)

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 returned in terms of delivery times could have their cost rankings reported as a measure. Note that Analysis Services 2005 has changed the behavior of a Microsoft extension to `Rank()` from its behavior in the 2000 version. The change may be more convenient, with perhaps no major difference in performance, and we will highlight this. (Note also that Essbase 9 does not support the `Rank()` function, although future releases may.)

Let's report on 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. We can sort our suppliers by costs and use that as the set within which to rank them. Rank counts start at 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]),
  BDESC
)'
MEMBER [Measures].[Cost Ranking] AS
'Rank (
  [Supplier].CurrentMember,
  [Cost-Ordered Suppliers]
)', FORMAT_STRING = '#;#;-'
SELECT
{ [Measures].[Delivery Time], [Measures].[Cost Ranking] } on columns,
{ BottomCount (
  [Supplier].[Supplier].Members,
  10,
  [Measures].[Delivery Time]
) } on rows
FROM [Purchasing]
WHERE [Time].[Year].[2004]
```

We chose `BottomCount()` instead of `TopCount()` because the business problem wanted the top performers, which is opposite of those that have the top times! BottomCount() is like the reverse of TopCount(). It returns the bottom *N* tuples from the set, ordered from smallest to largest. 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 rephrase the query like this:

```
WITH
SET [Delivery-Ordered Suppliers] AS
'BottomCount (
  [Supplier].[Supplier].Members,
  10,
  ([Measures].[Delivery Time])
)'
MEMBER [Measures].[Cost Ranking] AS
'Rank (
  [Supplier].CurrentMember,
  [Delivery-Ordered Suppliers]  // our 10 suppliers
)', FORMAT_STRING = '#;#;-'
SELECT
{ [Measures].[Total Cost], [Measures].[Cost Ranking] } on columns,
{ [Delivery-Ordered Suppliers] } on rows
FROM [Purchasing]
WHERE [Time].[Year].[2004]
```

Now, let's tackle a more demanding yet very real-life query. Let's say that you 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.

You need to derive the year-to-date units sold to calculate this. You're also going to take the ranking of the 10 salespeople within an ordered set of all salespeople, so you should name that ordered set. Let's assume that the 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 you 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].CurrentMember,
  [Last Year Ordered SalesPeople]
)', FORMAT_STRING = '#;#;-'
SET [This Year Top 10 SalesPeople] AS
'TopCount (
  [SalesPerson].[Individual].Members,
  10,
  [Measures].[YTD Units Count]
)'
MEMBER [Measures].[This Year Rank] AS
'Rank (
  [SalesPerson].CurrentMember,
  [This Year Top 10 SalesPeople]
)', FORMAT_STRING = '#;#;-'
MEMBER [Measures].[YTD Units Change] as
'[YTD Units Count] -
  ([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].[Aug. 2004])
```

Note that the WHERE clause defines August 2004 to be the date that the year-to-date accumulates all the values to.

TIP Rank() returns 0 when the tuple isn't found in the set. In reports like these, users will probably want to see a blank or a symbol like - to indicate "not found." You can use a format string that replaces zero values with the appropriate indicator in these calculated members. Each of the examples in this section will display a dash (-) instead of 0. See Appendix D for format code details.

Handling Tied Ranks: Analysis Services

What if there's a tie for third place in the top 10? You want to see 3 for each of the tuples in positions 3, 4, and 5, and then you want to see "6" for the tuple in position 6. As an extension to the standard, Analysis Services supports an optional third argument to Rank() that is the sort criteria expression used to sort the set. The semantics for this function have changed between Analysis Services 2000 and 2005. When the expression is provided, it is used to determine if ties exist and what the right rank should be. In Analysis Services 2000, the expression was used when the tuple was found to search neighbors in the set and determine fair ranking numbers. For example, if you had defined the rank calculated members above as the following, you would get fair scoring across the salespeople:

```
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]
)'
```

With these semantics, regardless of whether the set is sorted in ascending or descending order in terms of the expression, the rank numbers reflect tied ordering from the first item. The set was not actually ordered in any way by the Rank() function, so if it wasn't sorted by the expression, then you may or may not get the results you wanted.

In Analysis Services 2005, this extended version of the `Rank()` function does sort the set, in ascending order. This means two things:

1. You don't actually need to sort the set prior to calling `Rank()` on it.
2. Whether or not you pass it a sorted set, you need to take the negative of the sort criteria to get the rank number in terms of descending order.

For example, the following rephrases the foregoing example to use the new semantics (note the unary negation operator, `-`, being used):

```
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]
)'
```

Taking the Top-*N* Descendants or Other Related Members across a Set

A single ranked report is as easy as our first example. However, you may need to build a more general report, where you need to drill down on a set of members that is selected by other logic in the query. We have two ways to do this:

- Employ the `DrillDownLevelTop()` function or one of its kin: `DrillDownLevelBottom()`, `DrillDownMemberTop()`, `DrillDownMemberBottom()`.
- Use `Generate()` to repeat the `TopCount()` function over the set.

`DrillDownLevelTop()` is tailor-made if you only need the top/bottom children, and is more convenient if the set has more than one dimension in it. It's really well-suited to a graphical user interface (GUI) that provides a top-*N* drill function, which may not have full knowledge of how the set was constructed in the first place. Otherwise, the for-each capability of `Generate()` can be used here. (Note that Essbase 9 does not support the `DrillDownXXXTop/Bottom` functions.) For example, let's say that you want the top three product subcategories for each product family. You can express that with the following:

```
SELECT { [Measures].[Dollar Sales] } on axis(0),
Generate (
  [Product].[Family].Members,
  {[Product].CurrentMember,
   TopCount (
    Descendants (
      [Product].CurrentMember,
      [Product].[Subcategory]
    ),
    3,
    [Measures].[Dollar Sales]
  )}
)
on axis(1)
FROM [Sales]
```

Note that `Generate()` returns the family-level member followed by the top three results by creating a set using `{}`.

What if you want to add group totals to the Top-3 lists here? See Chapter 7 for the techniques.

To simply drill from our family members down to the top three children each, the `DrillDownLevelTop()` function could hardly be easier:

```
SELECT { [Measures].[Dollar Sales] } on axis(0),
DrillDownLevelTop (
  [Product].[Family].Members,
  3,
  ,   // intentionally blank
  [Measures].[Dollar Sales]
)
on axis(1)
FROM [Sales]
```

The empty third argument to the function tells it to drill down on the members at the lowest level in the set (which is all of them because they're all at the same depth in the hierarchy).

You can employ the `Generate()` function to take the top *N* of the top *N* as well, for example, the top five products for each of the top five customers as shown in the following query; the results are shown in Figure 6-3:

```
Generate (
  TopCount (
    [Customer].[Cust ID].Members,
    5,
    [Measures].[Dollar Sales]
```

```
  ),
  CrossJoin (
    { Customer.CurrentMember },
    TopCount (
      [Product].[Subcategory].Members,
      5,
      [Measures].[Dollar Sales]
    )
  ),
  ALL
)
```

		Dollar Sales
667900	Camp Kitchen	36,447.40
	Tents	36,337.90
	Razor Accessories	35,936.50
	Construction	17,705.10
	Action Figures	11,149.10
507450	Games	27,323.50
	Musical	25,131.80
	Baseball	21,104.50
	Blank Media	21,067.50
	CD Players	13,330.70
465300	Cordless Phones With Call	30,601.40
	Headphones	26,872.80
	Garage Door Openers	23,697.80
	Cordless Phones	21,234.30
	Weights	12,791.20
450850	Pools, Pumps	33,520.00
	Rainwear	23,620.90
	Medical Supplies	18,807.80
	Trampolines	18,237.10
	Camp Furniture	16,075.70
479600	Vitamins, Nutrition	38,006.90
	DVD Players	30,529.60
	Electrical Shop	18,907.00
	GamePlace	4,057.40
	Bags	3,751.20

Figure 6-3 Result of Top-5 of Top-5 query.

Getting the Fewest/Most Tuples to Reach a Threshold

Suppose that you want to find the smallest set of products to focus your efforts on in order to reach some target of sales. Or, you want to obtain the largest set of customers for which the total cost to reach them does not exceed some threshold. Another pair of functions, `TopSum()` and `BottomSum()`, rank and select tuples based on a threshold value, and can be used to solve selection problems like the ones described. The syntax of TopSum() and BottomSum() is

```
TopSum (set [, numeric_expression] )
BottomSum  (set [, numeric_expression] )
```

The optional *numeric_expression* provides the target to which values of the set are summed.

Using `TopSum()` is straightforward. For example, the following selects the top products that will meet the expected quota of $5,000,000:

```
TopSum (
  [Product].[Category].members,
  5000000,
  ([Measures].[Dollar Sales], [Scenario].[Quota])
)
```

`BottomSum()` is almost as straightforward. Depending on your application, you may or may not want to exceed the threshold. If you want to include the last tuple that causes you to exceed your threshold, it is just as simple. If you want to get the group of the smallest such that you do *not* exceed your budget, you need to trim the last tuple from the returned set only if the sum was greater than your target. Also, it's almost guaranteed that you want to trim tuples with related values that are missing. You would want to trim the tuples from the input to `BottomSum()`, and you can only remove the extra tuple on the results. So, you would compose MDX like the following for Analysis Services:

```
Head (
  BottomSum (
    NonEmptyCrossJoin (
      [Customer].[City].members,
      {[Measures].[Ad Budget]},
      1
    )
```

```
    5000000,
    [Measures].[Ad Budget]
  ) AS [A],
  Iif (
    Sum ([A], [Measures].[Ad Budget]) > 5000000,
    [A].Count - 1,
    [A].Count
  )
)
```

Note that the set alias `[A]` is used to refer to the results of `BottomSum()`, so you don't need to run `BottomSum()` more than once, and you don't need to create a named set outside of the `Head()`.

NOTE **Essbase only supports `Count()` syntax but does filter out empty values, so the following would be the equivalent:**

```
Head (
  BottomSum (
    [Customer].[City].members,
    5000000,
    [Measures].[Ad Budget]
  ) AS [A],
  Iif (
    Sum ([A], [Measures].[Ad Budget]) > 5000000,
    Count ([A], INCLUDEEMPTY) - 1,
    Count ([A], INCLUDEEMPTY)
  )
)
```

`BottomSum()` (and `BottomPercent()`, described next) are the ranking functions from which you will most likely want to strip empty and/or zero values.

BOTTOMSUM() AND TOPSUM() USUALLY WANT POSITIVE NUMBERS

While you use `BottomSum()` to accumulate the set associated with the smallest numbers, note that the function sums values tuples until the sum is greater than the threshold. If your target is positive, summing negative numbers gets you farther away from the goal, not nearer to it. If your target is negative, then you'll either start off above your goal, or you will never get any closer.

Retrieving the Top *N* Percent of Tuples

While many reports are interested in the top *N* tuples, the components of the top *N* percent are also quite useful for understanding true business drivers. (The top 10 customers may account for only 5 percent of the business, which is important but doesn't help you address the 80/20 picture.) The `TopPercent()` and `BottomPercent()` functions let you directly request a proportional threshold. The syntax for these functions is

```
TopPercent (set [, numeric_expression] )
BottomPercent  (set [, numeric_expression] )
```

`TopPercent()` returns the largest values required to hit the percentage, in descending order, while BottomPercent() returns the smallest values required to hit the percentage, in ascending order.

For example, you can determine the top 20 percent of customers in terms of dollar sales with the following:

```
TopPercent (
  [Customer].[Customer ID].Members,
  20,
  [Measures].[Dollar Sales]
)
```

The result set may have one customer, or it may contain many customers.

This is the beginning of Pareto analysis. There are other parts of really providing insight into the 80/20 populations, and we will describe another part of constructing 80/20 reports in Chapter 7.

Retrieving the Top *N* Percent of the Top *N* Percent

When you are attempting to understand business drivers, you may need to see drivers of more than one dimension. For example, once you know the top 20 percent of customers in terms of sales, you may want to understand the top 20 percent of products for each of the customers. This is another case where you see the phrase "for each" in the business statement of the problem and reach for the `Generate()` function. Say that you want to see the top 20 percent of customers, and for each of them you want to see the total across all products and the top 30 percent of product categories for each customer. The query looks very similar to the one we used for the top *N* of the top *N* earlier:

```
SELECT
{ [Measures].[Dollar Sales] } on axis(0),
Generate (
  TopPercent (
```

```
    [Customer].[Customer ID].members,
    20,
    [Measures].[Dollar Sales]
  ),
  CrossJoin (
    { [Customer].CurrentMember },
    { [Product].[All Product],
      TopPercent (
        [Product].[Category].Members,
        30,
        [Measures].[Dollar Sales]
    ) }
  )
)
on axis(1)
FROM [Sales]
```

One notable difference is that you see `CrossJoin()` used here. Remember from Chapter 1 that you can't make a set of tuples without using it. You want to return (Customer, Product) combinations, where customer is a single member, but product is a list. So, you `CrossJoin()` the current customer, which is one of the top 20 percent, with the set of products, to produce the combinations.

A more sophisticated analysis would look at the products across these customers, and we will continue the example in Chapter 7 to see how you can extract more relevant business information from this selection.

Putting Members/Tuples in Dimension Order (Ancestors First or Last)

There are a number of reasons that you might want to arrange a set of members (or tuples) into the order defined by the dimension(s). They may have been picked via a GUI and sequenced in the order in which they were picked. They may have been selected by one of the ranking functions here. And you may be looking to incorporate parents or ancestors, either to include database aggregates, to include Analysis Services visual totals (discussed in Chapter 7), or to enable more coherent drill-up/drill-down. You may simply be trying to get higher-level aggregates to appear below their children, for example on a financial statement. All of these cases are handled by using the `Hierarchize()` function.

Syntactically, Hierarchize() simply takes a set and returns the set put into hierarchical order. The default and standard behavior is to put parents before children as well. Both Analysis Services and Essbase take an additional option flag (POST) that specifies that parents should follow children, as in a financial statement.

The following simply takes the Terms dimension's members and puts parents after children:

```
Hierarchize (
  [Terms].Members,
  POST
)
```

Let's say that you want to drill up from the top 25 product subcategories, instead of drilling down on product families to the top three subcategories each. Once again, you use `Generate()`, as shown in the following expression:

```
Hierarchize(
  Generate (
    TopCount (
      [Product].[Subcategory].Members,
      25,
      [Measures].[Dollar Sales]
    ),
    { Ancestor(
        [Product].CurrentMember,
        [Product].[Family]
      ),
      [Product].CurrentMember
    }
  )
)
```

Instead of concatenating a product with its descendants as in the earlier example, you concatenate the ancestor of the product with the product. It doesn't matter whether the ancestor comes first or second, because `Hierarchize()` is going to sort things afterwards.

However, you may be asking, "Why don't we get duplicates? Can't two subcategories have the same family?" Yes, two subcategories can have the same category, but by default, `Generate()` strips duplicate values from the set it returns. So, there's no harm in putting the ancestor in twice or more.

Reversing a Set

One technical need that comes up occasionally is reversing a set. There is no direct general way to do this in standard MDX. Depending on the origin of the set, you might be able to just sort them, or else you can use the `Rank()` function. You can't use the `Hierarchize(,POST)` function, because that only reverses the order of parents relative to children. Members are still within dimension order overall. In Analysis Services, it's perhaps better done using a stored procedure or external function. You will look at the stored procedure

method in Chapter 10. However, maybe you don't have the ability to add a stored procedure, and you need a pure-MDX solution. In Analysis Services, if the set is in the dimension's hierarchical ordering, you can reverse it by sorting on the intrinsic ID property. For example, the following would sort a collection of time members into reverse-hierarchical order (the extra `TYPED` argument was introduced with AS2005):

```
Order (
  [Time].[2006].Children,
  [Time].CurrentMember.Properties ("ID", TYPED),
  BDESC
)
```

Otherwise, you sort on the rank:

```
Order (
  [Time].[2006].Children,
  Rank( [Time].CurrentMember, [Time].[2006].Children),
  BDESC
)
```

In Analysis Services 2005, you can make use of the .CurrentIndex function to reverse a set without resorting to sorting. The following would accomplish the task (notice that you need to use a named set or alias as the argument for .CurrentIndex):

```
Generate (
  [Time].[2006].Children AS [Alias_Children],
  [Alias_Children].Item (
    [Alias_Children].Count - [Alias_Children].CurrentIndex - 1
  ),
  ALL
)
```

Summary

The sorting and ranking functions directly implement a large number of user requests. Somewhat more advanced requests are readily satisfied by combining these functions with other functions like `Generate()`, `CrossJoin()`, and others that are part of the core "moving parts" of MDX. Hopefully, this chapter has both introduced you to the ranking and sorting functions and deepened your capacity for translating user requirements into MDX compositions. The next chapter will start the real focus on patterns of MDX and how to combine component expressions into elegant assemblies of high value to the users.