Using MDX with DeepSee

Version 2018.1
2024-05-02

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using MDX with DeepSee
Caché Version 2018.1 2024-05-02

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

About This Book

This book describes how to use the MDX (MultiDimensional eXpressions) query language with DeepSee. It includes the following sections:

- Background
- Introduction to MDX Queries
- Working with Levels
- Working with Dimensions and Hierarchies
- Working with Sets
- Tuples and Cubes
- Filtering a Query
- Adding Summaries
- Creating Calculated Members

For a detailed outline, see the table of contents.

The other developer books for DeepSee are as follows:

- *Getting Started with DeepSee* briefly introduces DeepSee and the tools that it provides.
- *DeepSee Developer Tutorial* guides developers through the process of creating a sample that consists of a cube, subject areas, pivot tables, and dashboards.
- *DeepSee Implementation Guide* describes how to implement DeepSee, apart from creating the model.
- *Defining DeepSee Models* describes how to define the basic elements used in DeepSee queries: cubes and subject areas. It also describes how to define listing groups.
- *Advanced DeepSee Modeling Guide* describes how to use the more advanced and less common DeepSee modeling features: computed dimensions, unstructured data in cubes, compound cubes, cube relationships, term lists, quality measures, KPIs, plugins, and other special options.
- *DeepSee MDX Reference* provides reference information on MDX as supported by DeepSee.
- *Tools for Creating DeepSee Web Clients* provides information on the DeepSee JavaScript and REST APIs, which you can use to create web clients for your DeepSee applications.

The following books are for both developers and users:

- *DeepSee End User Guide* describes how to use the DeepSee User Portal and dashboards.
- *Creating DeepSee Dashboards* describes how to create and modify dashboards in DeepSee.
- *Using the DeepSee Analyzer* describes how to create and modify pivot tables, as well as use the Analyzer in general.

Also see the article *Using PMML Models in Caché*.

For general information, see the *InterSystems Documentation Guide*.

1

Background

This chapter provides an overview of DeepSee and explains how DeepSee supports MDX (MultiDimensional eXpressions), which is a query language implemented by many vendors.

Be sure to consult the online *InterSystems Supported Platforms* document for this release for information on system requirements for DeepSee.

1.1 Purpose of DeepSee

The purpose of InterSystems DeepSee is to enable you to embed business intelligence (BI) into your applications so that your users can ask and answer sophisticated questions of their data. Your application can include *dashboards*, which can include *pivot tables*.

A *pivot table* is an interactive, drillable display of data, designed for specific user roles or for specific areas of your user interface.

Each pivot table has an underlying MDX query that is executed at runtime. Instead of directly querying your transactional tables, DeepSee queries its *cubes*, which are synchronized with the transactional tables. (For information on defining cubes, see *Defining DeepSee Models*.)

1.2 Introduction to Pivot Tables

Pivot tables are central to DeepSee; they select and aggregate data and display it in an interactive format.

The following figure shows an example pivot table. It shows the number of patients and the average allergy count per patient, grouped by age and gender.

		Female		Male	
		Patient Count	Avg Allergy Count	Patient Count	Avg Allergy Count
0 to 29	0 to 9	674	0.76	740	0.76
	10 to 19	691	0.78	769	0.79
	20 to 29	680	0.75	704	0.81
30 to 59	30 to 39	772	0.79	741	0.84
	40 to 49	710	0.79	777	0.77
	50 to 59	573	0.85	568	0.82
60+	60 to 69	361	0.83	341	0.76
	70 to 79	336	0.81	236	0.79
	80+	211	0.77	116	0.79

Because the concepts are interrelated, making it difficult to discuss each concept without reference to the others, it is useful for us to start with preliminary definitions:

- A *level* enables you to group records. A level has *members*. Each member, in turn, corresponds to a specific group of records in the source data.

  For example, the `Age Group` level has the members `0 to 29`, `30 to 59`, and `60+`. The `Age Bucket` level has the members `0-9`, `10-19`, `20 to 29`, and so on. The `Gender` level has the members `Female` and `Male`.

- A *measure* is a value displayed in the body of the pivot table; it is based on values in the source data, for selected records. For a given context, a measure aggregates the values for all applicable source records and represents them with a single value.

  For example, the measure `Patient Count` is the number of patients, and the measure `Avg Allergy Count` is the average number of allergies per patient.

1.3 Introduction to MDX

MDX is a standard query language for OLAP (online analytical processing) databases. The MDX language provides syntax for referring to cube elements. Most of the statements and functions in the language enable you to execute queries against a cube. The returned data is a result set, which can be displayed as a pivot table.

MDX also provides the capability of extending a cube definition. In particular, you can define new elements based on existing elements, and then use those new elements in MDX queries.

DeepSee supports MDX as follows:

- When you create a pivot table in the Analyzer, DeepSee generates and uses an MDX query, which you can view directly.
- The Analyzer provides an option for directly running MDX queries.
- You can run MDX queries in the MDX shell and see their results.
- DeepSee provides an API that you can use to run MDX queries.
- Within a DeepSee model, you use MDX expressions and queries to define certain elements, as discussed in the following subsection.

Note that some MDX queries are too complex to create within the current user interface. You can execute such queries in the shell or via the API, but you cannot create them via drag and drop actions in the Analyzer.

For further information, see the following sources:

- For information on the Analyzer, see *Using the DeepSee Analyzer*.
- For information on the MDX shell, see *Getting Started with DeepSee*.
- For information on the MDX API, see the *DeepSee Implementation Guide*.

Note: DeepSee provides an implementation of MDX. Results may differ from other implementations.

1.3.1 MDX in DeepSee Models

In DeepSee models, you can use MDX expressions and queries in the following places:

- Within a cube definition:
  - You use an MDX member expression to define calculated members.
  - You use an MDX set expression to define named sets.
  - You use an MDX set expression to filter the cube.

  These are all optional.
- Within a subject area definition, you use an MDX set expression to filter the subject area. This is optional; a subject area does not have to include a filter.
- Within a KPI (key performance indicator) definition, you can use an MDX query to define the KPI. This is optional; you can use an SQL query instead.

For information, see *Defining DeepSee Models* and the *DeepSee Implementation Guide*.

2

Introduction to MDX Queries

This chapter introduces MDX queries, and it covers the following topics:

2.1 Contents of the DemoMDX Cube

When you create SQL queries in an unfamiliar database, you start by becoming acquainted with the tables and their columns. Similarly, when you create MDX queries, you start by becoming acquainted with the available cubes and their contents.

1. Start the Terminal.
2. Switch to the SAMPLES namespace.
3. To access the MDX shell, enter the following command:

   ObjectScript

   ```
   Do ##class(%DeepSee.Utils).%Shell()
   ```

4. To see the available cubes, enter the following command (note that it is not case-sensitive):

   ```
   CUBE
   ```

The Terminal then displays a list of cubes.

5. To see the available contents of a cube, enter the following command:

```
CUBE cubename
```

For example:

```
CUBE demomdx
```

The shell ignores the case of the command and of the cube name.

The Terminal displays the following:

```
Measures
  %COUNT
  Age
  Avg Age
  Allergy Count
  Avg Allergy Count
  Test Score
  Avg Test Score
AgeD
    All Patients
  H1
    All Patients
    Age Group
    Age Bucket
AllerD
  H1
    Allergies
BirthD
  H1
    Year
    Quarter Year
BirthQD
  H1
    Quarter
DiagD
  H1
    Diagnoses
GenD
  H1
    Gender
ColorD
  H1
    Favorite Color
HomeD
  H1
    ZIP
    City
DocD
  H1
    Doctor
```

The `DemoMDX` cube represents patients. The contents of this cube are as follows:

- The `Measures` section lists the available measures: `%COUNT`, `Age`, `Ave Age`, `Allergy Count`, and so on. These measures are associated with patients and can be aggregated across patients.
- The `Dimensions` section contains dimensions. This cube contains the dimension `AgeD`, `AllerD`, and so on.

  For now, a dimension is the container for one or more hierarchies; for more detail, see the chapter "Working with Dimensions and Hierarchies."
- The first element within a dimension is a hierarchy. By convention, in this sample, each dimension contains one hierarchy named `H1`.

  For now, a hierarchy is the container for one or more levels; for more detail, see the chapter "Working with Dimensions and Hierarchies."
- The elements within a hierarchy are levels. This cube includes the levels `Age`, `Age Group`, `Gender`, `ZIP`, `City`, and others. These levels enable you to select different groups of patients.

In many MDX applications, the same name is used for a dimension, a hierarchy in it, and a level in that hierarchy. This practice can be confusing for someone who is learning MDX, so this sample cube uses the following arbitrary naming conventions:

- Dimension names are short and end with the letter *D*.
- Each dimension contains one hierarchy named `H1`.
- Level names are meant to be user friendly. (In the Analyzer, users see both dimension and level names but primarily work with levels.)

As you will see later, in DeepSee MDX, you can omit parts of identifiers. The naming conventions in this sample make it clear which parts can be omitted.

Note: A cube can also contain calculated members and named sets. The CUBE command in the MDX shell does not display these elements, although you can use them in the shell and elsewhere.

2.2 The Simplest Query

In the MDX shell, enter the following MDX query (this is not case-sensitive):

```
SELECT FROM demomdx
```

The shell displays the results as follows:

```
Result:             1,000
```

This query simply counts patients.

MDX is not case-sensitive except for member keys, which are discussed in the chapter "Working with Levels."

2.3 Members

A key component of an MDX query is the *member*. Each level contains one or more members. For example, the `City` level contains multiple members, one for each city in the data. A level enables you to select records; specifically, each member of the level allows you to access a subset of the records.

In the `DemoMDX` cube, each member of each level in this cube allows you to select some group of patients.

In this section, we execute a simple query to see members of a level in the `DemoMDX` cube:

1. In the MDX shell, enter the following MDX query (this is not case-sensitive):

   ```
   SELECT homed.h1.city.MEMBERS ON ROWS FROM demomdx
   ```

   The shell displays the members of the `City` level, as follows:

   ```
   1 Cedar Falls                       110
   2 Centerville                        99
   3 Cypress                           112
   4 Elm Heights                       118
   5 Juniper                           122
   6 Magnolia                          114
   7 Pine                              121
   8 Redwood                           111
   9 Spruce                             93
   ```

For now, let us discuss only the member names, which are shown in the second column.

The City level contains the members Cedar Falls, Centerville, Cypress, and so on. Each member of this level represents the set of patients with that home city. For example, the Centerville member represents all patients whose home city is Centerville.

2.4 Measures

Another key component of an MDX query is the *measure*. All DeepSee queries use at least one measure. If you do not specify a measure, DeepSee uses the default measure defined in the cube. For most cubes, the default measure is %COUNT, which is a count of the records. Let us examine some of the measures in the sample cube:

1. In the MDX shell, enter the following simple query:

   ```
   SELECT MEASURES.[%COUNT] ON COLUMNS FROM demomdx
   ```

   This query returns a result set that contains one column of data — the aggregate value for the %COUNT measure — across the entire data set that the cube represents. Depending on the data in your sample, the shell displays something like the following:

   ```
   %COUNT
     1,000
   ```

   In this example, there are 1000 patients.

2. In the MDX shell, enter the following query:

   ```
   SELECT MEASURES.[avg test score] ON COLUMNS FROM demomdx
   ```

   This query returns a result set that shows the aggregate value for the Avg Test Score measure across the entire data set.

   Depending on the data in your sample, the shell displays something like the following:

   ```
   Avg Test Score
            74.75
   ```

   This number is the average test score across all patients.

2.5 Referring to Members and Measures

In the preceding sections, you explored the elements of the DemoMDX cube, in particular its measures and levels, and you should have some sense of the data contained in it. You also wrote simple MDX queries. The next step is to learn the syntax that you use to refer to members and measures:

- To refer to a member:

  ```
  [dimension_name].[hierarchy_name].[level_name].[member_name]
  ```

- To refer to all members of a level:

  ```
  [dimension_name].[hierarchy_name].[level_name].MEMBERS
  ```

MEMBERS is the MDX function that returns the members of the level. This book introduces some key MDX functions. The *DeepSee MDX Reference* provides reference information for all MDX functions that DeepSee supports.

- To refer to a measure:

```
[MEASURES].[measure_name]
```

Note the following variations:

- In any of these names, you can omit the square brackets ([]) if the name consists only of alphanumeric characters.

  For a more formal discussion of identifiers, see the section "Identifiers" in the *DeepSee MDX Reference*.

- When referring to a level or member, you can omit the hierarchy name. If you do, MDX uses the first level with the given name, as defined in this dimension. (This variation is an InterSystems extension to MDX.)
- When referring to a member, you can omit the level name. If you do, MDX uses the first member with the given name, as defined within this dimension. (This variation is an InterSystems extension to MDX.)

You cannot omit the dimension name.

The following examples are all equivalent in DeepSee MDX:

```
[GenD].[H1].[GENDER].Female
[GenD].Female
GenD.H1.GENDER.Female
GenD.H1.Female
GenD.Female
```

2.6 Simple MDX Queries with %COUNT

This section presents simple forms of MDX queries, which do not refer to a measure and thus use the default measure defined in the cube (which is usually %COUNT).

- To use the members of a given level as columns, use a query of the following form:

```
SELECT [dim_name].[hier_name].[lev_name].MEMBERS ON COLUMNS FROM cubename
```

- To use the members of one level as columns and use members of another level as rows, use a query of the following form:

```
SELECT [dim_name].[hier_name].[lev_name].MEMBERS ON COLUMNS,
[dim_name].[hier_name].[lev_name].MEMBERS ON ROWS FROM cubename
```

  Note: Do not include the line break that is shown here. The book includes this line break only for readability (especially in the printed form of the book). The MDX shell does not permit this line break.

- To use a single member as a column, use a query of the following form:

```
SELECT [dim_name].[hier_name].[lev_name].[member_name] ON COLUMNS FROM cubename
```

You can use 0 instead of COLUMNS, and you can use 1 instead of ROWS. (For reasons of space, this book uses 0 and 1 rather than COLUMNS and ROWS.)

In all cases, the SELECT statement returns a result set, which the MDX shell displays in tabular form.

Let us try queries that use these variations:

1. Enter the following MDX query:

```
SELECT gend.h1.gender.MEMBERS ON 0 FROM demomdx
```

   The shell executes the query and displays the results like the following (yours will be slightly different):

```
                                Female                  Male
                                    488                   512
```

   Notice the following:

   - Because the query did not specify a measure, the numbers shown are values for `%COUNT`, which counts the patients.
   - There are two members shown as columns (or on the *column axis* of the result set). The `Female` member refers to the female patients, and `Male` refers to the male patients.

2. Try a shorter version of the same query:

```
SELECT gend.gender.MEMBERS ON 0 FROM demomdx
```

   This query returns the same data as the previous query.

3. Now enter the following variation:

```
SELECT gend.gender.female ON 0 FROM demomdx
```

   The result might be like the following:

```
                                  Female
                                      488
```

   In this example, the query selected a specific member rather than both members of this dimension.

4. Try this variation (with the member name in a different case):

```
SELECT gend.gender.FEMALE ON 0 FROM demomdx
```

   This returns the same result as the preceding query.

5. Enter a slightly more complex query:

```
SELECT gend.h1.gender.MEMBERS ON 0,homed.h1.zip.MEMBERS ON 1 FROM demomdx
```

   The shell executes the query and displays the results like the following:

```
                                Female                  Male
1 32006                             105                   110
2 32007                              58                    53
3 34577                             173                   174
4 36711                              41                    58
5 38928                             111                   117
```

   In this case, the results contain multiple rows, one row for each patient ZIP code. The counts are shown for each ZIP code, by gender.

   If there are multiple rows of results, the MDX shell displays a column that indicates the row numbers of the results.

2.6.1 Axis Skipping

In other implementations of MDX, you cannot omit an axis if you use a higher-numbered axis. That is, you cannot use `ROWS` unless you also use `COLUMNS`.

In DeepSee MDX, however, if you omit COLUMNS, DeepSee uses %COUNT, as follows:

```
SELECT gend.h1.gender.MEMBERS ON ROWS FROM demomdx
```

```
1 Female                                  488
2 Male                                    512
```

2.7 Sets

In MDX, the columns and the rows are axes of the query and of the result set. The following result set, for example, has gender on the column axis and home ZIP codes on the row axis:

```
                              Female                Male
1 32006                          105                 110
2 32007                           58                  53
3 34577                          173                 174
4 36711                           41                  58
5 38928                          111                 117
```

An axis uses a *set*. The general syntax for a *set expression* is as follows:

```
{expression1, expression2, ...}
```

This list can include any number of items. In DeepSee MDX, if the list includes only one item, you can omit the curly braces. Also, a set can be empty, but if so cannot be used on a query axis.

Within the set, each expression can be one of the following:

- A *member expression*, which is either of the following:
  - An explicit reference to a single member by name. For example:

    ```
    [PatDim].[GENDERH1].[GENDER].[F]
    ```

  - An expression that uses an MDX function to return a single member. For example:

    ```
    [PatDim].[GENDERH1].[GENDER].[F].NEXTMEMBER
    ```

    (NEXTMEMBER is the MDX function that returns the next member of the level. The chapter "Working With Levels" introduces this and other functions.)

- An expression that uses an MDX function, like MEMBERS, to return a set. For example:

  ```
  [dimension_name].[hierarchy_name].[level_name].MEMBERS
  ```

There are other forms of expressions and other kinds of set elements; see the chapter "Working With Sets" and the *DeepSee MDX Reference*.

You can use any non-null set expression within a SELECT statement. In general, SELECT has the following basic syntax for a query that uses one axis:

```
SELECT set_expression ON COLUMNS FROM cubename
```

Or:

```
SELECT set_expression ON 0 FROM cubename
```

The following form is a query that uses two axes:

```
SELECT set_expression ON COLUMNS,set_expression ON ROWS FROM cubename
```

Or:

```
SELECT set_expression ON 0,set_expression ON 1 FROM cubename
```

A SELECT statement can use additional axes, but the shell does not display their results in a readable form.

2.7.1 Examples

Now try some query variations that use different kinds of sets, as shown in the preceding section.

1. The following example uses a set created by a comma-separated list:

```
SELECT {gend.h1.gender.MEMBERS,homed.h1.city.MEMBERS} ON 0 FROM demomdx
```

```
                    Female      Male   Cedar F   Centerv   Cypress   Elm Hei ...
                       488       512       110        99       112       118...
```

   As you can see, the results have too many columns to be shown in full.

2. Try a variation that uses the same set as rows instead of columns:

```
SELECT {gend.h1.gender.MEMBERS,homed.h1.city.MEMBERS} ON 1 FROM demomdx
```

```
 1 Female                                488
 2 Male                                  512
 3 Cedar Falls                           110
 4 Centerville                            99
 5 Cypress                               112
 6 Elm Heights                           118
 7 Juniper                               122
 8 Magnolia                              114
 9 Pine                                  121
10 Redwood                               111
11 Spruce                                 93
```

3. Let us expand the preceding by moving gender to the columns and adding home ZIP codes as another set of rows:

```
SELECT gend.h1.gender.MEMBERS ON 0,{homed.h1.city.MEMBERS,homed.h1.zip.MEMBERS} ON 1 FROM demomdx
```

```
                                Female                  Male
 1 Cedar Falls                       58                    52
 2 Centerville                       41                    58
 3 Cypress                           51                    61
 4 Elm Heights                       53                    65
 5 Juniper                           58                    64
 6 Magnolia                          58                    56
 7 Pine                              64                    57
 8 Redwood                           58                    53
 9 Spruce                            47                    46
10 32006                            105                   110
11 32007                             58                    53
12 34577                            173                   174
13 36711                             41                    58
14 38928                            111                   117
```

4. Try using a member multiple times within a set:

```
SELECT gend.h1.gender.MEMBERS ON 0,{homed.h1.[36711],homed.h1.[36711]} ON 1 FROM demomdx
```

```
                                Female                  Male
1 36711                             41                    58
2 36711                             41                    58
```

2.8 Displaying Measures

Any MDX query uses at least one measure. If you do not indicate the measure to use, DeepSee uses the default measure defined in the cube, usually %COUNT, which is a count of the records. There are multiple ways to display other measures. This section introduces a couple of them.

To use measures in queries, you can do the following:

- You can display a measure as a column and optionally display a set as rows. For example:

```
SELECT MEASURES.[avg allergy count] ON 0,colord.MEMBERS ON 1 FROM demomdx
```

```
                          Avg Allergy Count
1 None                                 1.08
2 Blue                                    1
3 Green                                1.05
4 Orange                               1.16
5 Purple                               1.22
6 Red                                  1.06
7 Yellow                               0.94
```

- You can display a measure as a row and optionally display a set as columns — the reverse of the preceding. For example:

```
SELECT gend.h1.gender.MEMBERS ON 0, MEASURES.[avg test score] ON 1 FROM demomdx
```

```
                               Female                Male
Avg Test Score                   73.49                74.42
```

- You can create a set of multiple measures and use that set as rows or columns. For example:

```
SELECT {MEASURES.[%COUNT],MEASURES.[avg test score]} ON 0,colord.MEMBERS ON 1 FROM demomdx
```

```
                               %COUNT       Avg Test Score
1 None                            239                72.68
2 Blue                            124                76.94
3 Green                           106                   72
4 Orange                          148                72.89
5 Purple                          135                74.87
6 Red                             121                74.92
7 Yellow                          127                74.41
```

- You can use the CROSSJOIN function as follows:

```
SELECT CROSSJOIN(MEASURES.[%COUNT],gend.h1.gender.MEMBERS) ON 0, diagd.h1.diagnoses.MEMBERS ON 1
FROM demomdx
```

```
                               Female                Male
1 None                            399                 429
2 asthma                           46                  44
3 CHD                              14                  23
4 diabetes                         23                  22
5 osteoporosis                     21                   1
```

  (For a more general introduction to this function, see "Combining Sets," later in this book.)

- You can use the MEMBERS function to display all measures (except for %COUNT), as follows:

```
SELECT gend.MEMBERS ON 0, MEASURES.MEMBERS ON 1 FROM demomdx
```

```
                               Female                Male
1 Age                          18,413              17,491
2 Avg Age                       37.73               34.16
3 Allergy Count                   326                 332
4 Avg Allergy Count              1.08                1.07
5 Test Score                   29,542              31,108
6 Avg Test Score                73.49               74.42
```

2.9 Including a Simple Filter in the Query

An MDX query can also include a filter, which reduces the number of rows of the fact table that the query could potentially use. To add a filter to a query, add a clause like the following to the end of your SELECT statement:

```
WHERE filter_details
```

For *filter_details*, the simplest form is as follows:

```
[dim_name].[hier_name].[level_name].[member_name]
```

You can use the same variations here as described in "Referring to Members and Measures," earlier in this chapter.

This expression filters the query so that DeepSee accesses only the records associated with this member. For example, the following query uses only patients who have osteoporosis:

```
SELECT MEASURES.[%COUNT] ON 0,aged.[age bucket].MEMBERS ON 1 FROM demomdx WHERE diagd.osteoporosis

                                    %COUNT
1 0 to 9                                 *
2 10 to 19                               *
3 20 to 29                               *
4 30 to 39                               *
5 40 to 49                               *
6 50 to 59                               *
7 60 to 69                               7
8 70 to 79                               7
9 80+                                    8
```

The MDX shell uses an asterisk (*) to indicate that a value is null.

The chapter "Filtering a Query" discusses WHERE in more detail.

2.10 Understanding the Contents of the MDX Results

Now that you have seen a variety of MDX queries and their results, it is time to review the results more formally. The MDX shell presents the results for an MDX query in the following general form:

		column label	*additional columns ...*
nnn	row label	data cell	data cell
nnn	*additional rows...*	data cell	data cell

The following rules determine the results:

- If the output includes multiple rows of results, the first column contains the row number (*nnn*, starting with 1 for the first row) for easy reference. If there is only one row of results, this column is not included.

  This column of numbers is shown only in the MDX shell and is not part of the result set.

- The output contains one column of data cells for each member of the set that you use on the column axis.
- In general, each column label corresponds to the name of the corresponding member, which might be a measure or might be a "regular" member.
- If you do not specify a set for the rows, the output contains one row with no label.
- If you specify a set for the rows, the output contains one row for each member of that set. The label for a given row is the name of the corresponding member, which might be a measure or might be a "regular" member (that is, a member that is not a measure).
- For any given data cell, the output displays either a value or an asterisk (*). The asterisk indicates that the value is null.

  To determine the value to use, DeepSee finds the *intersection* of the member used for the column and the member (if any used) for the row:

  – If one member is a measure and the other is not a measure, DeepSee finds the value of that measure for that member. For example, if one member is the Ave Age measure, and the other member is the 34577 ZIP code, then the corresponding data cell contains the average age of patients whose home ZIP code is 34577.

  – If neither member is a measure, DeepSee uses the default measure, which is usually %COUNT. For example, if one member is the gender F, and the other member is the 34577 ZIP code, then the corresponding data cell contains the count of all female patients whose home ZIP code is 34577.

  – If both members are measures, DeepSee uses the measure that is on the Columns axis.

    (Note that if both members are calculated measures, DeepSee also considers SOLVE_ORDER. For details, see "SOLVE_ORDER Clause" in the *DeepSee MDX Reference*.)

2.10.1 Notes on Independence of Query Axes

DeepSee considers each query axis independently of the others. Sometimes the result is counter-intuitive. This section shows two examples.

2.10.1.1 Set Order Is Unaffected by Other Sets in the Query

In all cases, it is important to remember that the order of the set returned is independent of any other sets used in the query, and sometimes the result is counter-intuitive. For example, consider the following query:

```
SELECT MEASURES.[%COUNT] ON 0,
TOPCOUNT(homed.city.MEMBERS,100,MEASURES.[%COUNT]) ON 1 FROM demomdx
```

```
                                    %COUNT
1 Juniper                               122
2 Pine                                  121
3 Elm Heights                           118
4 Magnolia                              114
5 Cypress                               112
6 Redwood                               111
7 Cedar Falls                           110
8 Centerville                            99
9 Spruce                                 93
```

This query shows the sort order that you obtain when you sort cities by patient count. (In this example, the number of members to select is 100, which is greater than the number of members; therefore all members are shown.)

If you modify the preceding query to return the top three members, you see the following:

```
SELECT MEASURES.[%COUNT] ON 0, TOPCOUNT(homed.city.MEMBERS,3,MEASURES.[%COUNT]) ON 1 FROM demomdx
```

```
                                    %COUNT
1 Juniper                               122
2 Pine                                  121
3 Elm Heights                           118
```

Now consider the results when you break out the patients by gender:

```
SELECT CROSSJOIN(MEASURES.[%COUNT],gend.gender.MEMBERS) ON 0,
TOPCOUNT(homed.city.MEMBERS,3,Measures.[%COUNT]) ON 1 FROM demomdx
```

```
                              Female                Male
1 Juniper                          58                   64
2 Pine                             64                   57
3 Elm Heights                      53                   65
```

The cities are listed in the same order in this query as in the preceding query, which did not specify a breakout for the columns. In this example, Juniper is the top-rated city by total patient count and so appears first. That is, the sorting is controlled by the total patient count in a city, not by any of the displayed values.

2.10.1.2 Set Membership Is Unaffected by Other Sets in the Query

It is also important to remember that the members of the returned set are independent of any sets used in the query, and sometimes the result is counter-intuitive. For example, consider the following query:

```
SELECT MEASURES.[%COUNT] ON 0, TAIL(birthd.year.MEMBERS,10) ON 1 FROM demomdx
```

```
                                    %COUNT
 1 1912                                  3
 2 1918                                  1
 3 1919                                  1
 4 1920                                  4
 5 1921                                  2
 6 1922                                  1
 7 1923                                  2
 8 1924                                  1
 9 1925                                  2
10 1927                                  5
```

Now consider the results when we show only a single gender:

```
SELECT CROSSJOIN(gend.male,MEASURES.[%COUNT]) ON 0, HEAD(birthd.year.MEMBERS,10) ON 1 FROM demomdx
```

```
                                    %COUNT
 1 1912                                  *
 2 1918                                  *
 3 1919                                  *
 4 1920                                  1
 5 1921                                  *
 6 1922                                  *
 7 1923                                  2
 8 1924                                  *
 9 1925                                  1
10 1927                                  1
```

The birth years are the same as in the preceding query, which shows data aggregated across genders.

2.11 DeepSee Name Resolution

In some cases, it is possible for multiple entities of the same type to have the same name. For example, an MDX cube can have two levels with the same name, as long as they are in different hierarchies (or possibly different dimensions). Suppose that the cube command showed the contents of a cube as follows:

```
...
Dimensions
  Geography
    ShipToHierarchy
      State
      City
    OrderByHierarchy
      State
      City
```

In DeepSee, you can omit the hierarchy name when you refer to a level. If the dimension contains multiple levels with the same name, DeepSee uses the first level with the given name. To refer to a level unambiguously, include the hierarchy name as well.

For another example, a level can have multiple members with the same name. Different states could have cities that have the same name, and those cities are different members. Or if your cube has a fine-grained level such as doctor name, that level could contain multiple members with the same name. In DeepSee, if you refer to the member by name, you access the first member of that name within the level. To refer to a member unambiguously, use its key. See "Member Keys" in the next chapter.

2.11.1 Nonexistent Members

In most cases, for a nonexistent member, DeepSee returns null, which the shell represents as an asterisk (*). For example, consider the following query:

```
SELECT colord.h1.color.pink ON 0 FROM demomdx
                                  No Member
                                          *
```

The exception is for measures, which are members of the MEASURES dimension. For a nonexistent measure, the system returns an error. For example:

```
SELECT MEASURES.[pat count] ON 0 FROM demomdx
ERROR #5001: Measure not found: pat count
```

2.11.2 Typographical Errors

In most situations, DeepSee treats typographical errors in the same way that it treats nonexistent members. For example:

```
SELECT colord.h1.color.MEMBERSSS ON 0 FROM demomdx
                                  No Member
                                          *
```

For another example:

```
SELECT colord.MEMBERSSS ON 0 FROM demomdx
                                  No Member
                                          *
```

When you refer to a dimension or an element within a dimension, however, the dimension name is required. If you mistype the dimension name, DeepSee treats that as an error:

```
SELECT colorddd.h1.color.MEMBERS ON 0 FROM demomdx
ERROR #5001: Dimension not found: colorddd
```

If you mistype the name of the cube or subject area, DeepSee treats that as an error:

```
SELECT colord.h1.color.MEMBERS ON 0 FROM demo
ERROR #5001: Cannot find Subject Area: 'DEMO'
```

2.12 Conventions Used in Remainder of the Book

For reasons of space, the rest of this book uses the following conventions:

- It uses 0 and 1 rather than `COLUMNS` and `ROWS`.
- It omits the square brackets in dimension, hierarchy, level, and member names, wherever possible. (See the section "Identifiers" in the *DeepSee MDX Reference*.)
- It omits the hierarchy names except where needed. (This is permitted in DeepSee MDX.)

Also, to help you quickly scan query examples:

- MDX statements, keywords, and functions are shown in uppercase.
- Cube elements and other user-supplied details are given in lowercase, except in running text.

3

Working with Levels

This chapter provides more information on levels, as well as an overview of the key MDX functions for working with them. It discusses the following topics:

- Overview of levels
- How to access single members of a level
- How to access the members of a level
- Order of members in a level
- How to select a member based on its relative position
- An introduction to time levels
- Special features for use with time levels
- How to access properties of members

3.1 Overview of Levels

A level enables you to group the data, and a level has members.

A member selects a set of records from the cube. For the `City` level, the `Juniper` member selects the patients whose home city is Juniper. Conversely, a record in the cube belongs to one or more members.

3.1.1 Possible Member Overlap

The members of a level can overlap each other. That is, a given record can belong to more than one member; this occurs if the level is based on a list. For example, consider the `Allergies` level, which contains one member for each allergy. A patient can have multiple allergies and thus can belong to multiple members of this level.

3.1.2 Null Values and Null Members

A level can have a Null member; this member selects the records that have no value for the data used by this level. Typically the name of this member is `None`.

3.1.3 Hierarchies

Levels belong to hierarchies. For information, see the chapter "Working with Dimensions and Hierarchies," later in this book.

3.2 Accessing Single Members of a Level

You can select a single member by referring to it directly. The general syntax is as follows:

```
[dimension_name].[hierarchy_name].[level_name].[member name]
```

As noted previously, in DeepSee MDX, you can omit the hierarchy name. Similarly, you can omit the level name.

For example:

```
SELECT MEASURES.[%COUNT] ON 0, allerd.[ant bites] ON 1 FROM demomdx
```

```
                                    %COUNT
ant bites                                47
```

3.2.1 Member Names

In a given level, member names are not required to be unique; that is, when the cube is built, no checking is performed to ensure that member names are unique in a given level. For example, the `Doctor` dimension can include multiple members with the same name.

3.2.2 Member Keys

In a well-defined cube, each member has a unique, case-sensitive key. To refer to a member by its key, use the following syntax:

```
[dimension_name].[hierarchy_name].[level_name].&[member_key]
```

In many cases, *member_key* is the same as the member name. For a generated Null member, the key is `<null>`.

For details on how DeepSee generates member keys, see the reference section "Key Values" in the *DeepSee MDX Reference*.

MDX provides a function (PROPERTIES), which you can use to access the key (or any other property) of a member; this function is discussed later in this chapter.

3.3 Accessing Multiple Members of a Level

You can access multiple members of a level in several different ways.

First, you can use the MEMBERS function. In this case, the syntax is as follows:

```
[dimension_name].[hierarchy_name].[level_name].MEMBERS
```

For example:

```
SELECT MEASURES.[%COUNT] ON 0, allerd.allergies.MEMBERS ON 1 FROM demomdx
```

```
                                    %COUNT
 1 No Data Available                   390
 2 additive/coloring agen               46
 3 animal dander                        34
 4 ant bites                            47
 5 bee stings                           36
 6 dairy products                       30
 7 dust mites                           35
 8 eggs                                 32
 9 fish                                 45
10 mold                                 51
11 nil known allergies                 140
12 peanuts                              58
13 pollen                               57
14 shellfish                            54
15 soy                                  36
16 tree nuts                            45
17 wheat                                52
```

You can also specify a range that selects adjacent members of a level, as follows:

```
member1:membern
```

For example:

```
SELECT MEASURES.[%COUNT] ON 0, {birthd.1942:birthd.1947} ON 1 FROM demomdx
```

```
                                    %COUNT
1 1942                                   6
2 1943                                   7
3 1944                                   6
4 1945                                  11
5 1946                                  12
6 1947                                   9
```

In this case, you can omit the dimension, hierarchy, and level identifiers for the member that you use for the end of the range. For example:

```
SELECT MEASURES.[%COUNT] ON 0, {birthd.1942:1947} ON 1 FROM demomdx
```

You can select multiple, nonadjacent members. To do so, refer to them directly, and place them in a comma-separated list surrounded by curly braces:

```
SELECT MEASURES.[%COUNT] ON 0, {allerd.eggs,allerd.soy,allerd.mold} ON 1 FROM demomdx
```

```
                                    %COUNT
1 eggs                                  32
2 soy                                   36
3 mold                                  51
```

3.4 Order of Members in a Level

Within a cube definition, a level definition determines the members in that level, as well as their *default order*, which is as follows:

- For non-date levels, members are sorted in increasing order alphabetically by name, unless the cube specifies a different sort order.
- For date levels, members are sorted chronologically, in ascending order or descending order, depending on the definitions in the cube.

The MEMBERS function returns the members in their default order, as defined by the level. For example:

```
SELECT gend.gender.MEMBERS ON 0,homed.city.MEMBERS ON 1 FROM demomdx
```

	Female	Male
1 Cedar Falls	58	52
2 Centerville	41	58
3 Cypress	51	61
4 Elm Heights	53	65
5 Juniper	58	64
6 Magnolia	58	56
7 Pine	64	57
8 Redwood	58	53
9 Spruce	47	46

If you have a subset of the members of a level and want to return them to the default order, use the HIERARCHIZE function as in the following example:

```
SELECT MEASURES.[%COUNT] ON 0, HIERARCHIZE({allerd.eggs,allerd.soy,allerd.mold}) ON 1 FROM demomdx
```

	%COUNT
1 eggs	32
2 soy	36
3 mold	51

For a more thorough introduction to this function, see the chapter "Working with Sets."

3.5 Selecting a Level Member by Relative Position

The following MDX functions enable you to select specific members of a level, relative to a given member. These functions all use the default order of members in the level. Note that the details are different for time dimensions and data dimensions (as defined in the cube definition):

- NEXTMEMBER returns the next member from a given level. For example:

  ```
  SELECT MEASURES.[%COUNT] ON 0, birthd.[Q1 1920].NEXTMEMBER ON 1 FROM demomdx
  ```

	%COUNT
Q2 1920	*

- PREVMEMBER returns the previous member.
- LEAD counts forward in the level and returns a later member. For example:

  ```
  SELECT MEASURES.[%COUNT] ON 0, birthd.[Q1 1920].LEAD(3) ON 1 FROM demomdx
  ```

	%COUNT
Q4 1920	1

- LAG counts backward in the level and returns an earlier member.

For time dimensions, each of these functions ignores any parent level. For example, the PREVMEMBER function can return a member that has a different parent. For data dimensions, however, each of these functions does consider the parent level. For example, the PREVMEMBER function considers only the previous member within the given parent member. (Note that the terms *time dimension* and *data dimension* refer specifically to the dimension type as defined in the cube. See *Defining DeepSee Models*.) For examples that show these differences, see the *DeepSee MDX Reference*.

3.6 Introduction to Time Levels

A time level groups records by time; that is, any given member consists of the records associated with a specific date and time. For example, a level called Transaction Date would group transactions by the date on which they occurred. There are two general kinds of time levels, and it is important to understand their differences:

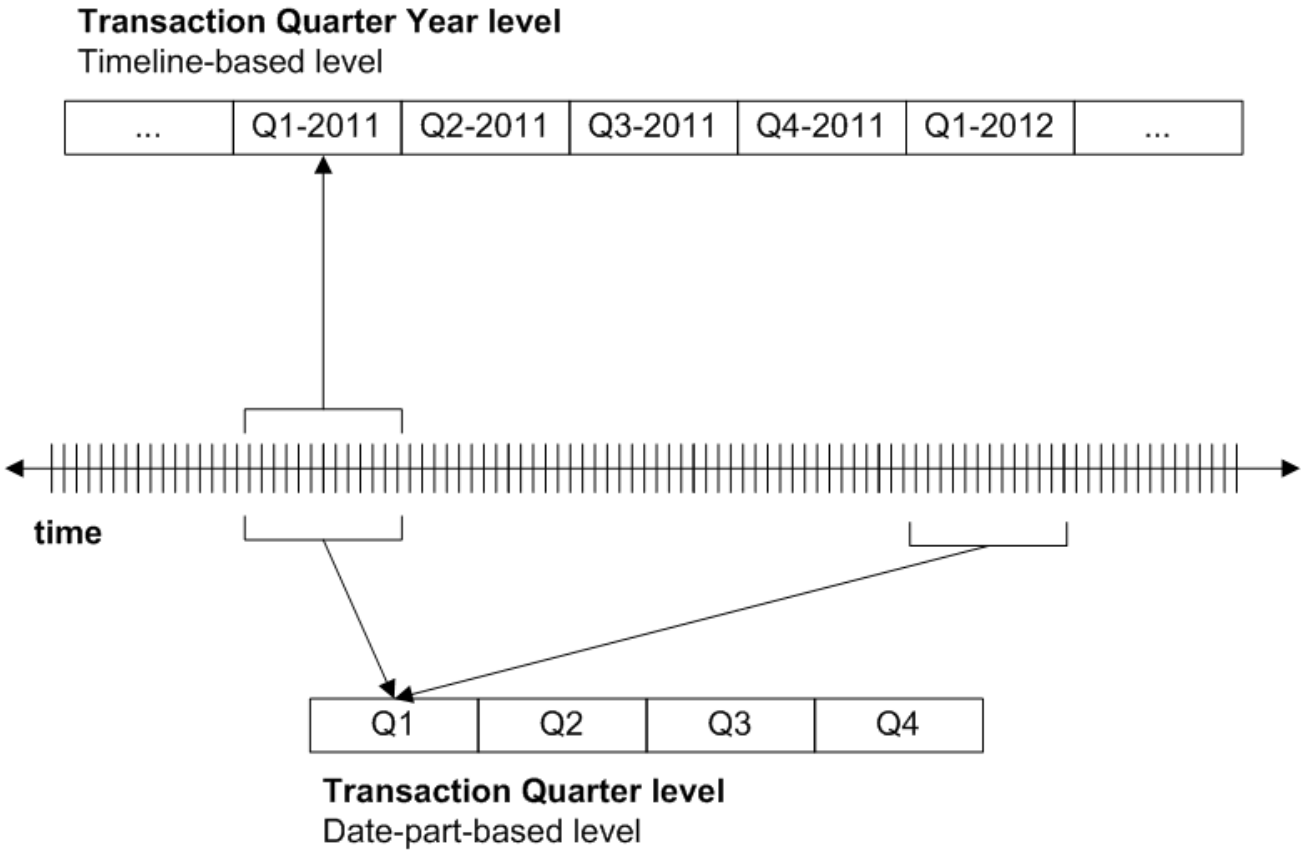
- *Timeline-based time levels.* This kind of time level divides the timeline into adjacent blocks of time. Any given member of this level consists of a single block of time. Or, more accurately, the member consists of the records associated with that block of time. For a level called `Transaction Quarter Year`, the member `Q1-2011` would group all the transactions that occurred in any of the dates that belong to the first quarter of 2011.

  This kind of level can have any number of members, depending on the source data.

- *Date-part-based time levels.* This kind of time level considers only *part* of the date value and ignores the timeline. Any given member consists of multiple blocks of time from different parts of the timeline, as shown in the following figure. Or, more accurately, the member consists of the records associated with those blocks of time. For a level called `Transaction Quarter`, the member `Q1` would group all the transactions that occurred in any of the dates that belong to the first quarter of any year.

  This kind of level has a fixed number of members.

The following figure compares these kinds of time levels:



You can use these kinds of levels together without concern; MDX will always return the correct set of records for any combination of members.

However, it is worth noting that some MDX functions are useful for timeline-based levels but not for date-part-based levels. These functions include PREVMEMBER, NEXTMEMBER, and so on.

For example, consider the following query, which refers to a date-part based level. When we use PREVMEMBER with `Q2`, the engine returns the data for Q1, as expected.

```
SELECT [BirthQD].[Q2].PREVMEMBER ON 1 FROM patients

Q1                                      219
```

However, when we use PREVMEMBER with `Q1`, which is at the start of the set, the engine returns nothing.

```
SELECT [BirthQD].[Q1].PREVMEMBER ON 1 FROM patients

                                        *
```

This result is correct, because the Q1 member refers to records related to quarter 1 in *all* years, and it is not meaningful to access records “earlier” than that.

In contrast, consider the following query, which refers to a timeline-based level:

```
SELECT [BirthD].[Q1 2011].PREVMEMBER ON 1 FROM patients

Q4 2010                                 4
```

In this case, the member refers to records in a specific part of the timeline, and it *is* meaningful to refer to earlier records.

3.7 Special Features for Use with Time Levels

DeepSee MDX includes extensions for use with time levels. These include the NOW member and the %TIMERANGE function.

3.7.1 Selecting a Member Relative to Today (Time Levels)

For date/time levels, DeepSee supports a special member called NOW, which uses the current date (runtime) and accesses the appropriate member of the level.

For example, the following query accesses the current year in the `Year` dimension:

```
SELECT birthd.year.NOW ON 1 FROM demomdx
2011                                    9
```

For another example:

```
SELECT birthd.[quarter year].NOW ON 1 FROM demomdx

Q2 2011                                 5
```

DeepSee also supports variations that indicate members that are offset from NOW. For example, `[NOW-1]` finds the member that precedes NOW by one position:

```
SELECT birthd.[quarter year].[NOW-1] ON 1 FROM demomdx

Q1 2011                                 1
```

You can use these variations within ranges of members like the following:

```
SELECT birthd.[quarter year].[now-1]:birthd.[quarter year].now ON 1 FROM demomdx


1 Q1 2011                                 1
2 Q2 2011                                 5
```

For more details, see "NOW Member for Date/Time Levels" in the *DeepSee MDX Reference*.

3.7.2 Selecting Ranges of Members of a Time Level

DeepSee provides an extension to MDX that enables you to define a range of members, for a time level. This extension is the %TIMERANGE function, which takes three arguments: a starting member, an ending member, and a keyword (either the default `INCLUSIVE` or `EXCLUSIVE`). You can om.it either but not both ends of the range

The following example uses both ends of the range:

```
SELECT NON EMPTY DateOfSale.YearSold.MEMBERS ON 1 FROM holefoods
WHERE %TIMERANGE(DateOfSale.YearSold.&[2009],DateOfSale.YearSold.&[2011])

me
1 2009                                    179
2 2010                                    203
3 2011                                    224
```

The next example shows another open-ended range, this time using the `EXCLUSIVE` keyword:

```
SELECT NON EMPTY DateOfSale.YearSold.MEMBERS ON 1 FROM holefoods
WHERE %TIMERANGE(,DateOfSale.YearSold.&[2009],EXCLUSIVE)


1 2007                                    124
2 2008                                    156
```

3.8 Accessing Properties

In MDX, a level can have properties that are specific to the level. Each member of the level can have a different value for the property. You can access these properties and display them in your query results. There are two kinds of properties:

- User-defined properties. In DeepSee, these are defined within the cube definition. For example, in the `DemoMDX` cube, the `City` level has two properties called `Population` (population of the city) and `Principal Export` (the principal export of the city).
- Intrinsic properties, which contain information such as the member name and the member's key. For a list, see the reference section "Intrinsic Properties" in the *DeepSee MDX Reference*.

Names of properties are not case-sensitive.

To access the property of a member, use the PROPERTIES function. For example:

```
SELECT homed.city.magnolia.PROPERTIES("Principal Export") ON 0 FROM demomdx

                                  bundt cake
```

For another example:

```
SELECT homed.cypress.LEAD(1).PROPERTIES("name") ON 0 FROM demomdx

                                      name
                                    Magnolia
```

3.8.1 Properties As String Expressions

MDX treats property values as strings. MDX also supports string literals (for example, `"my label"`) and a concatenation operator (+). Therefore, you can create expressions like the following:

```
"Next after Cypress: " + homed.cypress.LEAD(1).PROPERTIES("name")
```

And you can use such expressions in MDX queries. For example:

```
SELECT "Next after Cypress: " + homed.cypress.LEAD(1).PROPERTIES("name") ON 0 FROM demomdx
```

```
                              Expression
                        Next after Cypress: Magnolia
```

3.8.2 Properties and Attributes

Properties are not the same as attributes, which are often mentioned when MDX is discussed.

In some implementations of MDX, attributes are used to *define* a cube. No MDX functions, however, directly use attributes.

DeepSee does not use attributes.

4

Working with Dimensions and Hierarchies

This chapter discusses hierarchies and dimensions. These elements are containers for levels but also have their own purposes. This chapter discusses the following topics:

- Introduction to dimensions and hierarchies
- How to access the members of a hierarchy
- Using parent-child relationships
- How to access siblings
- How to access cousins
- How to access descendants
- How to access a member within an iteration

4.1 Introduction to Dimensions and Hierarchies

Most MDX functions refer directly to levels or to their members. In MDX, however, levels belong to hierarchies, which belong to dimensions. Hierarchies and dimensions provide additional features beyond those provided by levels.

Hierarchies are a natural and convenient way to organize data, particularly in space and time. For example, you can group cities into countries, and countries into regions. In such cases, it is useful to be able to query for the child cities of a given country, or to query for the parent postal region for a given country. A DeepSee cube defines the hierarchies among the levels, and MDX provides functions that enable you to work with the hierarchy, so that you can write such queries.

In MDX, a *dimension* contains one or more hierarchies that specify how to categorize the records in a similar manner. There is no formal relationship between two different hierarchies or between the levels of one hierarchy and the levels of another hierarchy. The purpose of a dimension is to define the default behavior of its hierarchies and levels.

4.1.1 The Measures Dimension

All measures belong to a special dimension called Measures. This dimension implicitly includes a single hierarchy that has no name. This hierarchy does not include levels. The members of this hierarchy are the measures.

4.1.2 The All Level

Other than the Measures dimension, each dimension can also define a special, optional level, which appears in all the hierarchies of that level: the *All level*. If defined, this level contains one member, the *All member*, which corresponds to all records in the cube.

For a given dimension, the actual name of the All member depends upon the cube definition. For example, `All Patients` is the All member for the `AgeD` dimension in the sample.

4.1.3 Example

If we use the `cube` command in the MDX shell, we see the following elements in the `demomdx` cube:

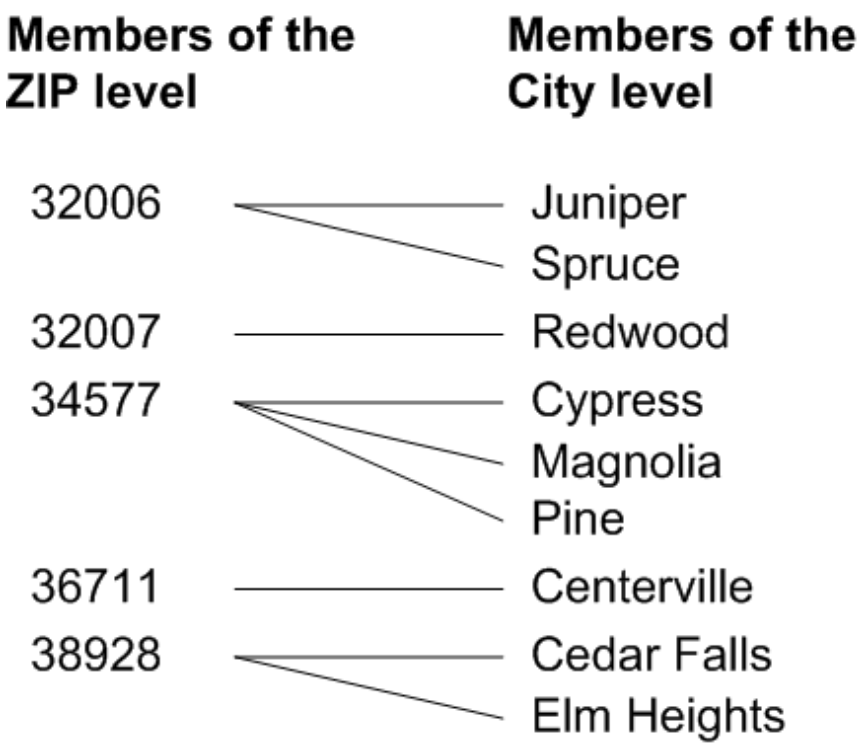
```
Elements of cube 'demomdx':
----------------------------------------
...
Dimensions
...
 HomeD
    H1
      ZIP
      City
...
```

The `HomeD` dimension contains one hierarchy (`H1`), which contains two levels:

- The `ZIP` level
- The `City` level

In a given hierarchy, a level is the parent of the level that is listed after it. This means, for example, that `ZIP` is the parent of `City`. More specifically, each member of `ZIP` is the parent of one or more members of `City`. That is, it is shorthand to say that one level is the parent of another level; the actual relationship is between members, not between levels. This shorthand is in common use, because it is convenient, even though it is not precise.

The following figure shows the relationships among the members of the `HomeD.H1` hierarchy:



The distinguishing feature of a hierarchy is that any given child element is unique to its parent. This example is artificial because in reality there is a many-to-many relationship between ZIP codes and cities.

4.2 Accessing the Members of a Hierarchy

To access the members of a hierarchy (that is, all the members of all its levels), you use the MEMBERS function. In this case, the syntax is as follows:

```
[dimension_name].[hierarchy_name].MEMBERS
```

In DeepSee MDX, if you omit the hierarchy name, the system assumes that you are referring to the first visible hierarchy in the given dimension.

For example, in the DemoMDX cube, the homed dimension has only one hierarchy. The following query shows the members of that hierarchy:

```
SELECT MEASURES.[%COUNT] ON 0, homed.MEMBERS ON 1 FROM demomdx
```

```
                                  %COUNT
 1 32006                             215
 2 Juniper                           122
 3 Spruce                             93
 4 32007                             111
 5 Redwood                           111
 6 34577                             347
 7 Cypress                           112
 8 Magnolia                          114
 9 Pine                              121
10 36711                              99
11 Centerville                        99
12 38928                            228
13 Cedar Falls                      110
14 Elm Heights                      118
```

When you use the MEMBERS function with a hierarchy, it returns the set of members in *hierarchical order*. The first member is the All member, if present. After that, each member is one of the following:

- The first child of the previous member.
- The next sibling of the previous member.

For another example, the following query shows all the measures (apart from %COUNT):

```
SELECT gend.gender.MEMBERS ON 0, MEASURES.MEMBERS ON 1 FROM demomdx
```

	Female	Male
1 Age	18,413	17,491
2 Avg Age	37.73	34.16
3 Allergy Count	326	332
4 Avg Allergy Count	1.08	1.07
5 Test Score	29,542	31,108
6 Avg Test Score	73.49	74.42

4.3 Using Parent-Child Relationships

DeepSee provides the following MDX functions that directly use parent-child relationships:

- CHILDREN returns the children, if any, of a given member. The returned value is a set of members, in the default order specified for the level. For example:

  ```
  SELECT MEASURES.[%COUNT] ON 0, homed.zip.[34577].CHILDREN ON 1 FROM demomdx
  ```

	%COUNT
1 Cypress	112
2 Magnolia	114
3 Pine	121

  For another example:

  ```
  SELECT MEASURES.[%COUNT] ON 0, homed.pine.CHILDREN ON 1 FROM demomdx
  ```

	%COUNT
No Result	

- PARENT returns the parent, if any, of a given member. For example:

  ```
  SELECT MEASURES.[%COUNT] ON 0, homed.city.[Elm Heights].PARENT ON 1 FROM demomdx
  ```

	%COUNT
38928	228

- FIRSTCHILD returns the first child, if any, of a given member. For example:

  ```
  SELECT MEASURES.[%COUNT] ON 0, homed.zip.[34577].FIRSTCHILD ON 1 FROM demomdx
  ```

	%COUNT
Cypress	112

- LASTCHILD returns the last child, if any.

4.4 Accessing Siblings

DeepSee provides the following MDX functions that access siblings of a member:

- FIRSTSIBLING returns the first sibling, if any, of a given member. For example:

```
SELECT MEASURES.[%COUNT] ON 0, birthd.[Q1 1920].FIRSTSIBLING ON 1 FROM demomdx

                                    %COUNT
Q1 1920                                  *
```

- LASTSIBLING returns the last sibling, if any.
- SIBLINGS returns the given member and all its siblings. For example:

```
SELECT MEASURES.[%COUNT] ON 0, homed.cypress.SIBLINGS ON 1 FROM demomdx

                                    %COUNT
1 Cypress                               112
2 Magnolia                              114
3 Pine                                  121
```

4.5 Accessing Cousins

The COUSIN function enables you to access a cousin, given a member at a higher level.

For example, the following query finds the cousin of Q1 1943, within the year 1990:

```
SELECT MEASURES.[%COUNT] ON 0, COUSIN(birthd.[Q1 1943],birthd.1990) ON1 FROM demomdx

                                   %COUNT
Q1 1990                                 5
```

To determine relative positions, DeepSee uses the default order of the members within the level, as determined by the cube definition.

4.6 Accessing Descendant Members

You can use the DESCENDANTS function to obtain descendents of a given member, within one or more lower levels. For example, the following query gets all the descendents of the year 1990, within the `[BirthD].[H1].[Period]` level:

```
SELECT DESCENDANTS(birthd.1990,birthd.period) ON 1 FROM demomdx

 1 Jan-1990                             *
 2 Feb-1990                             2
 3 Mar-1990                             1
 4 Apr-1990                             1
 5 May-1990                             1
 6 Jun-1990                             *
 7 Jul-1990                             2
 8 Aug-1990                             2
 9 Sep-1990                             1
10 Oct-1990                             3
11 Nov-1990                             1
12 Dec-1990                             *
```

The DESCENDANTS function provides many options for accessing descendents within different parts of the hierarchy, but the preceding usage is the most common scenario.

4.7 Accessing the Current Member within an Iteration

In a typical query, you iterate through a set of members, perhaps displaying each as a row. Sometimes you want to do something specific with each member in turn. To do so, you use the CURRENTMEMBER function, which accesses the member used in the current context.

For example, consider the following query:

```
SELECT MEASURES.[%COUNT] ON 0, homed.city.MEMBERS ON 1 FROM demomdx
```

```
                                    %COUNT
1 Cedar Falls                          110
2 Centerville                           99
3 Cypress                              112
4 Elm Heights                          118
5 Juniper                              122
6 Magnolia                             114
7 Pine                                 121
8 Redwood                              111
9 Spruce                                93
```

This query has one row for each city. The data shown is the `%COUNT` measure. Suppose that instead we would like to show the city's population, which we access via the PROPERTIES function. This function requires a reference to the member used in the row; for that, we use the CURRENTMEMBER function, which we can call as follows:

```
[dimension_name].[hierarchy_name].CURRENTMEMBER
```

With this function, we can create the following variation of our query:

```
SELECT homed.h1.CURRENTMEMBER.PROPERTIES("Population") ON 0, homed.city.MEMBERS ON 1 FROM demomdx
```

```
                                       H1
1 Cedar Falls                      90,000
2 Centerville                      49,000
3 Cypress                           3,000
4 Elm Heights                      33,194
5 Juniper                          10,333
6 Magnolia                          4,503
7 Pine                             15,060
8 Redwood                          29,192
9 Spruce                            5,900
```

For another example, the following query shows the internal keys for the members of `Doctor`:

```
SELECT docd.h1.CURRENTMEMBER.PROPERTIES("KEY") ON 0, docd.[doctor].MEMBERS ON 1 FROM demomdx
```

```
                                     KEY
 1 None                             <null>
 2 Ahmed, Thelma                       34
 3 Alton, Chad                         35
 4 Black, Ashley                        4
..
```

5

Working with Sets

This chapter discusses how to create and use sets. It discusses the following topics:

- Introduction to sets
- How to create set expressions
- How to create named sets
- Order of members in a set
- How to sort a set
- How to select a subset
- How to combine sets
- How to filter a set
- How to remove null elements
- How to remove duplicate elements from sets
- How to count the elements in a set

5.1 Introduction to Sets

A set contains zero or more elements, and there are three general kinds of elements: members, scalar values, and tuples (for information on tuples, see the next chapter, "Tuples and Cubes").

You use sets on the axes of MDX queries; you also use them to build other sets. Note that although an MDX set can be empty, you cannot use such an empty set as an axis.

5.2 Creating Set Expressions

The general syntax for a *set expression* is as follows:

```
{expression1, expression2, ...}
```

This list can include any number of elements. In DeepSee MDX, if the list includes only one element, you can omit the curly braces.

Each set element can be one of the following:

- A *member expression*, which is either of the following:
  - An explicit reference to a single member by name.
  - An expression that uses an MDX function to return a single member.
- An expression that uses an MDX function, like MEMBERS, to return a set.
- A range of members in the same level, as shown in the chapter "Working With Levels":

  ```
  member1:membern
  ```

- A scalar value, which is one of the following:
  - A reference to a measure. The expression `MEASURES.[Avg Test Score]` is a scalar value; it returns either a number or null in all contexts.
  - A numeric or string constant such as `37` or `"label"`.
  - A numeric expression such as `(37+3)/2`.
  - A percentage literal. For example: `10%`

    There must be no space between the number and the percent sign.
  - An expression that uses an MDX function to return a scalar value.

  For example, the PROPERTIES function returns a scalar value; for an introduction to this function, see the chapter "Working With Levels," earlier in this book.

  For another example, the AVG function and other summary functions return scalar values; see the chapter "Adding Summaries," later in this book.

For complete details, see the *DeepSee MDX Reference*.

5.3 Creating Named Sets

It is often useful to create a set and assign a name to it, so that you can reuse that set in multiple ways. Also, the syntax of a query is often easier to read when you use named sets.

You can create one or more named sets within a query, as follows:

```
WITH with_clause1 with_clause2 ... SELECT query_details
```

Where:

- Each expression *with_clause1*, *with_clause2*, and so on has the following syntax:

  ```
  SET set_name AS 'set_expression'
  ```

- *query_details* is your MDX query.

Then your query can refer the named set by name in all the places where you can use other set expressions.

For example:

```
WITH SET testset AS '{homed.city.members}'
SELECT MEASURES.[%COUNT] ON 0, testset ON 1 FROM demomdx
```

```
                                    %COUNT
1 Cedar Falls                         184
2 Centerville                         194
3 Cypress                             134
4 Elm Heights                         146
5 Juniper                             176
6 Magnolia                            169
7 Pine                                118
8 Redwood                             182
9 Spruce                              197
```

Note: This named set is a *query-scoped* named set; its scope is the query. For information on session-scoped named sets, see "CREATE SET Statement," in the *DeepSee MDX Reference*.

Your cubes might contain additional named sets that you can use in all queries; see *Defining DeepSee Models*.

5.4 Order of Members in a Set

Every set has an inherent order (a first member, a second member, and so on).

When you use an MDX function to return a set, that function determines the order of the members in the set. Wherever possible, MDX functions use the natural order of members as specified in the cube definition.

For example, the MEMBERS function returns the members of a level in the order specified in the cube definition (typically in alphabetic order or date order, depending on the level).

When you construct a set as described in the previous section, the order in which you list elements controls the order of the members in the set. For example, suppose that you specify a set as follows:

```
{gend.gender.MEMBERS,allerd.allergies.MEMBERS}
```

This set consists of the members of the `Gender` dimension, followed by the members of the `Allergies` dimension.

5.5 Selecting Subsets

SUBSET returns a set of members from a given set, by position. You specify a set, the starting position, and the number of members to return. The starting position is 0. For example:

```
SELECT MEASURES.[%COUNT] ON 0,SUBSET(homed.city.MEMBERS,0,3) ON 1 FROM demomdx
```

```
                                    %COUNT
1 Cedar Falls                          110
2 Centerville                           99
3 Cypress                              112
```

The EXCEPT function provides another way to get a subset; see the next section.

Also see the chapter "Filtering a Query," later in this book.

5.6 Sorting Sets

This section describes ways to sort sets. It discusses the following topics:

- How to sort items by the value of a measure
- How to select a top or bottom subset
- How to apply hierarchical order

5.6.1 Sorting a Set by a Measure Value

It is often useful to sort members by the value of some measure. For example, you might want to sort departments by response, so that you can look at the departments with the slowest responses. Or you might sort products by their sales revenues so that you can look at the top-ranked products.

To return a set in the order that you specify, use the ORDER function. This function takes an argument, typically a measure reference, that specifies the value to use when determining the order of the set members. For example:

```
SELECT MEASURES.[avg test score] ON 0,
ORDER(homed.city.MEMBERS,MEASURES.[avg test score],BDESC) ON 1 FROM demomdx
```

```
                            Avg Test Score
1 Juniper                             75.08
2 Redwood                             75.07
3 Cedar Falls                         75.03
4 Elm Heights                         74.96
5 Pine                                74.76
6 Spruce                              74.47
7 Magnolia                            74.13
8 Cypress                             73.96
9 Centerville                         73.79
```

The optional third argument can be one of the following:

- ASC (the default) — Use this to sort in ascending order, while preserving the hierarchy, if applicable.
- DESC — Use this to sort in descending order, while preserving the hierarchy, if applicable.
- BASC — Use this to break the hierarchy and sort all members in ascending order.
- BDESC — Use this to break the hierarchy and sort all members in descending order.

For example, the following query breaks the hierarchy:

```
SELECT MEASURES.[avg test score] ON 0,
ORDER(homed.MEMBERS,MEASURES.[avg testscore],BDESC) ON 1 FROM demomdx
```

```
                            Avg Test Score
 1 Juniper                            75.08
 2 Redwood                            75.07
 3 32007                              75.07
 4 Cedar Falls                        75.03
 5 38928                              75.00
 6 Elm Heights                        74.96
 7 32006                              74.78
 8 Pine                               74.76
 9 Spruce                             74.47
10 34577                              74.28
11 Magnolia                           74.13
12 Cypress                            73.96
13 Centerville                        73.79
14 36711                              73.79
```

In contrast, the following preserves the hierarchy:

```
SELECT MEASURES.[avg test score] ON 0,
ORDER(homed.MEMBERS,MEASURES.[avg testscore],DESC) ON 1 FROM demomdx
```

```
                            Avg Test Score
 1 32007                              75.07
 2 Redwood                            75.07
 3 38928                              75.00
 4 Cedar Falls                        75.03
 5 Elm Heights                        74.96
 6 32006                              74.78
 7 Juniper                            75.08
 8 Spruce                             74.47
 9 34577                              74.28
10 Pine                               74.76
11 Magnolia                           74.13
12 Cypress                            73.96
13 36711                              73.79
14 Centerville                        73.79
```

5.6.2 Selecting a Top or Bottom Subset

It is useful to sort items in some way and then choose a subset from the top or bottom, such as the top five. The following MDX functions enable you to do so.

- HEAD and TAIL return the first part or the last part of the set, respectively, given a member count. For example:

  ```
  SELECT MEASURES.[%COUNT] ON 0,HEAD(homed.city.MEMBERS,3) ON 1 FROM demomdx
  ```

  ```
                                      %COUNT
  1 Cedar Falls                          110
  2 Centerville                           99
  3 Cypress                              112
  ```

  The members of the returned set have the same order as in the original set.

- TOPCOUNT and BOTTOMCOUNT are similar to HEAD and TAIL, respectively, but also include an optional argument to specify how to sort the set before extracting the subset.

  For example, the following query returns the top-rated cities, by patient count.

  ```
  SELECT MEASURES.[%COUNT] ON 0,TOPCOUNT(homed.city.MEMBERS,4,MEASURES.[%COUNT]) ON 1 FROM demomdx
  ```

  ```
                                      %COUNT
  1 Juniper                              122
  2 Pine                                 121
  3 Elm Heights                          118
  4 Magnolia                             114
  ```

  For another example:

  ```
  SELECT MEASURES.[avg test score] ON 0,TOPCOUNT(homed.city.MEMBERS,5,MEASURES.[avg test score]) ON
   1 FROM demomdx
  ```

  ```
                               Avg Test Score
  1 Juniper                             75.08
  2 Redwood                             75.07
  3 Cedar Falls                         75.03
  4 Elm Heights                         74.96
  5 Pine                                74.76
  ```

- TOPSUM and BOTTOMSUM are similar to TOPCOUNT and BOTTOMCOUNT, respectively. Instead of specifying the number of members to return, however, you specify a cutoff value that is applied to a total across the members. For example, you could retrieve the products that account for the top $5 million in sales.
- TOPPERCENT and BOTTOMPERCENT are similar to TOPCOUNT and BOTTOMCOUNT, respectively. Instead of specifying the number of members to return, however, you specify a cutoff percentage that is applied to a total across the members. For example, you could retrieve the products that account for the top 10% of sales.

5.6.3 Applying Hierarchical Order

The HIERARCHIZE function accepts a set of members from the same dimension and returns a set containing those members in hierarchical order, that is, the order specified by the hierarchy. For example:

```
SELECT MEASURES.[%COUNT] ON 0,
HIERARCHIZE({homed.36711,homed.38928,homed.[elm heights],homed.Spruce}) ON 1 FROM demomdx
```

```
                                    %COUNT
1 36711                              99
2 Spruce                             93
3 38928                             228
4 Elm Heights                       118
```

If the members belong to different hierarchies in the dimension, the different hierarchies are returned in an arbitrary order.

5.7 Combining Sets

Sets are the building blocks of MDX queries. When you write an MDX query, you must specify a set to use on each axis. DeepSee supports the following MDX functions that you can use to combine sets:

- UNION combines two sets (optionally discarding any duplicate members) and returns a set that contains all the members of these sets. For example:

  ```
  WITH SET set1 AS '{allerd.eggs,allerd.soy,allerd.wheat}'
  SET set2 AS '{allerd.[dairy products],allerd.pollen,allerd.soy,allerd.wheat}'
  SELECT MEASURES.[%COUNT] ON 0, UNION(set1,set2) ON 1 FROM demomdx
  ```

  ```
                                      %COUNT
  1 eggs                                32
  2 soy                                 36
  3 wheat                               52
  ```

  Because the query does not use UNION with the ALL keyword, duplicates are removed.

- INTERSECT examines two sets and returns a set that contains all the members that are in both sets, optionally retaining duplicates. For example:

  ```
  WITH SET set1 AS 'TOPCOUNT(homed.city.members,5,MEASURES.[avg allergy count])'
  SET set2 AS 'TOPCOUNT(homed.city.members,5,MEASURES.[avg age])'
  SELECT MEASURES.[%COUNT] ON 0, INTERSECT(set1,set2) ON 1 FROM demomdx
  ```

  ```
                                      %COUNT
  1 Magnolia                            114
  2 Redwood                             111
  3 Cypress                             112
  4 Cedar Falls                         110
  ```

- EXCEPT examines two sets and removes the members in the first set that also exist in the second set, optionally retaining duplicates. For example:

  ```
  WITH SET set1 AS '{allerd.eggs,allerd.eggs,allerd.soy,allerd.wheat}'
  SET set2 AS '{allerd.[diary products],allerd.pollen,allerd.wheat}'
  SELECT MEASURES.[%COUNT] ON 0, EXCEPT(set1,set2) ON 1 FROM demomdx
  ```

  ```
                                      %COUNT
  1 eggs                                32
  2 soy                                 36
  ```

  Also see "The %NOT Optimization," later in this book.

- CROSSJOIN returns a set that consists of the cross-product of two sets. Both sets can consist of members. Or one set can consist of members and the other set can consist of measures. If both sets contain measures, the DeepSee engine issues the error `Two measures cannot be crossjoined`. For example:

```
SELECT MEASURES.[%COUNT] ON 0, CROSSJOIN(diagd.diagnoses.MEMBERS,
aged.[age group].MEMBERS) ON 1 FROM demomdx
```

```
                                    %COUNT
 1 None->0 to 29                        389
 2 None->30 to 59                       333
 3 None->60+                            106
 4 asthma->0 to 29                       40
 5 asthma->30 to 59                      39
 6 asthma->60+                           11
 7 CHD->0 to 29                           *
 8 CHD->30 to 59                         12
 9 CHD->60+                              25
10 diabetes->0 to 29                      1
11 diabetes->30 to 59                    20
12 diabetes->60+                         24
13 osteoporosis->0 to 29                  *
14 osteoporosis->30 to 59                 *
15 osteoporosis->60+                     22
```

  Note that the MDX shell displays a null value as an asterisk (*). For information on suppressing null values, see "Removing Null Elements from a Set," later in this chapter.

  Also see the NONEMPTYCROSSJOIN function.

  Also note that unlike the previous functions, which return sets of members, CROSSJOIN returns a set of tuples (as does NONEMPTYCROSSJOIN). Tuples are discussed later in this book.

5.8 Filtering a Set by a Measure or Property Value

You can also examine measure values for the members in a set and use those values to filter the set. To do so, you use the FILTER function.

The FILTER function uses a set and a logical expression. It examines a set and returns the subset in which the given logical expression is true for each member. The logical expression typically compares a measure value to a constant or to another measure value. For example:

```
SELECT MEASURES.[%COUNT] ON 0, FILTER(homed.city.MEMBERS,MEASURES.[%COUNT]>115) ON 1 FROM demomdx
```

```
                                  %COUNT
1 Elm Heights                         118
2 Juniper                             122
3 Pine                                121
```

It is important to understand that this filtering occurs at an aggregate level: The measure value is computed for each possible member in the query. The FILTER function considers those aggregate values and removes members as appropriate.

You can use the same function with member properties as follows:

```
SELECT homed.h1.CURRENTMEMBER.PROPERTIES("Population") ON 0,
FILTER(homed.city.MEMBERS,homed.h1.CURRENTMEMBER.PROPERTIES("Population")>20000) ON 1 FROM demomdx
```

```
                                ZIP
1 Cedar Falls                    90,000
2 Centerville                    49,000
3 Elm Heights                    33,194
4 Redwood                        29,192
```

5.9 Removing Null Elements from a Set

In some cases, a set might contain null elements. For example, the CROSSJOIN function could potentially return null elements (as is shown in the preceding section).

If you precede the set expression with the keyword NON EMPTY, DeepSee suppresses the null elements. For example:

```
SELECT MEASURES.[%COUNT] ON 0,NON EMPTY CROSSJOIN(diagd.diagnoses.MEMBERS,
aged.[age group].MEMBERS) ON 1 FROM demomdx
```

```
                                    %COUNT
 1 None->0 to 29                        389
 2 None->30 to 59                       333
 3 None->60+                            106
 4 asthma->0 to 29                       40
 5 asthma->30 to 59                      39
 6 asthma->60+                           11
 7 CHD->30 to 59                         12
 8 CHD->60+                              25
 9 diabetes->0 to 29                      1
10 diabetes->30 to 59                    20
11 diabetes->60+                         24
12 osteoporosis->60+                     22
```

5.10 Removing Duplicates

When you combine sets, you may want to remove duplicates. This is true especially when you have created and combined sets in multiple steps. To be certain that the resulting set has no duplicates, you use the DISTINCT function.

For example, suppose that the query must return a specific city as reference, which is needed for comparison to the other cities. Consider the following query, which displays a reference city, followed by a set of cities with a given patient count:

```
WITH SET refcity AS '{homed.juniper}' SELECT MEASURES.[%COUNT] ON 0,
{refcity,FILTER(homed.city.MEMBERS,MEASURES.[%COUNT]>115)} ON 1 FROM demomdx
```

```
                                    %COUNT
1 Juniper                               122
2 Elm Heights                           118
3 Juniper                               122
4 Pine                                  121
```

Compare to the following query, which removes the duplicate reference city:

```
WITH SET refcity AS '{homed.juniper}' SELECT MEASURES.[%COUNT] ON 0,
DISTINCT({refcity,FILTER(homed.city.MEMBERS,MEASURES.[%COUNT]>115)}) ON 1 FROM demomdx
```

```
                                    %COUNT
1 Juniper                               122
2 Elm Heights                           118
3 Pine                                  121
```

5.11 Counting the Elements of a Set

To count the elements of a set, use the COUNT function. For example:

```
SELECT COUNT(docd.doctor.MEMBERS) ON 0 FROM demomdx
```

```
                                     COUNT
                                        41
```

By default, COUNT considers any empty elements and counts them along with the non-empty elements. If you use the EXCLUDEEMPTY keyword as the second argument, this function returns the number of non-empty elements.

To see this, first consider the following query:

```
SELECT aged.[age group].MEMBERS ON 0, diagd.diagnoses.MEMBERS ON 1 FROM demomdx WHERE MEASURES.[%COUNT]
```

```
                          0 to 29          30 to 59               60+
1 None                         389               333               106
2 asthma                        40                39                11
3 CHD                            *                12                25
4 diabetes                       1                20                24
5 osteoporosis                   *                 *                22
```

The following query counts the number of members of the `Diagnoses` level and uses the WHERE clause to get only patients in the age group 0 to 29:

```
WITH SET myset AS 'diagd.diagnoses.MEMBERS' SELECT COUNT(myset) ON 0 FROM demomdx WHERE aged.[0 to 29]
```

```
                                    COUNT
                                         5
```

In this query, COUNT returns 5 because it considers empty members. In contrast:

```
WITH SET myset AS 'diagd.diagnoses.MEMBERS' SELECT COUNT(myset,EXCLUDEEMPTY) ON 0 FROM demomdx WHERE
aged.[0 to 29]
```

```
                                    COUNT
                                         3
```

6

Tuples and Cubes

This chapter discusses two additional key concepts in MDX: tuples and cubes. It discusses the following topics:

- Introduction to tuples
- How DeepSee determines the value of a tuple
- Examples of tuple expressions
- How to use tuples as axes of a query
- Introduction to cubes
- Relationship of higher levels to a cube dimension
- Relationship of multiple hierarchies to a cube dimension

6.1 Introduction to Tuples

A tuple is a combination of members from different dimensions. Each tuple has a single value (possibly null).

Every data cell in a result set is a tuple. For example, consider the following query:

```
SELECT MEASURES.[%COUNT] ON 0, homed.city.members ON 1 FROM demomdx
```

```
                                    %COUNT
1 Cedar Falls                          110
2 Centerville                           99
3 Cypress                              112
4 Elm Heights                          118
5 Juniper                              122
6 Magnolia                             114
7 Pine                                 121
8 Redwood                              111
9 Spruce                                93
```

This query returns a set of nine tuples. For example, the first tuple is a combination of `Cedar Falls` (from the `City` dimension) and `%COUNT` (from the `Measures` dimension).

6.1.1 Creating Tuples

You can create a tuple directly, via the following syntax:

```
(member_expr1, member_expr2, member_expr3, ...)
```

Where *member_expr1*, *member_expr2*, *member_expr3*, and so on are member expressions.

In other implementations of MDX, each of these member expressions must be associated with a different dimension. This means that a tuple cannot include more than one member from the same dimension.

In DeepSee MDX, a tuple expression can include more than one member expression from the same dimension. In most cases, the result is null, because in most cases, a record belongs to only one member. However, in DeepSee, a level can be based on a list value, which means that a given record can belong to multiple members. For example, the tuple `(allerd.soy,allerd.wheat)` represents all patients who are allergic to both soy and wheat.

6.1.2 Fully and Partially Qualified Tuples

A tuple is either fully qualified or partially qualified:

- If the tuple expression refers to each dimension in the cube, the tuple is *fully qualified*. A fully qualified tuple refers to a very small number of records and is too granular to be commonly used.
- If the tuple expression does not refer to each dimension in the cube, the tuple is *partially qualified*. A partially qualified tuple can be very useful, especially when used to filter the data used by the query.

  If a tuple refers to only one member, the tuple is equivalent to that member. For example, the following expressions both access the same data:

  ```
  (colord.red)
  colord.red
  ```

  The expression `(colord.red)` is a tuple expression uses the `Red` member of the `ColorD` dimension.

  The expression `colord.red` is a member expression that refers to the `Red` member of the `ColorD` dimension.

  Each expression accesses only the patients whose favorite color is red.

6.1.3 Sets of Tuples

You can create sets of tuples, by enclosing a comma-separated list of tuple expressions within curly braces:

```
{tuple_expression1, tuple_expression2, ...}
```

(Note that in other implementations of MDX, for any tuple in a set, you must construct each tuple in the same way. For example, if the first tuple uses dimension A in its first list item, all the other tuples must do so as well. DeepSee MDX does not have this restriction.)

You can also create sets of tuples by using the CROSSJOIN or NONEMPTYCROSSJOIN functions. For example:

```
SELECT MEASURES.[%COUNT] ON 0, CROSSJOIN(gend.gender.MEMBERS,homed.city.members) ON 1 FROM demomdx
```

```
                                  %COUNT
 1 Female->Cedar Falls                58
 2 Female->Centerville                41
 3 Female->Cypress                    51
 4 Female->Elm Heights                53
 5 Female->Juniper                    58
 6 Female->Magnolia                   58
 7 Female->Pine                       64
 8 Female->Redwood                    58
 9 Female->Spruce                     47
10 Male->Cedar Falls                  52
11 Male->Centerville                  58
12 Male->Cypress                      61
13 Male->Elm Heights                  65
14 Male->Juniper                      64
15 Male->Magnolia                     56
16 Male->Pine                         57
17 Male->Redwood                      53
18 Male->Spruce                       46
```

You can use these set expressions in all the places where set expressions are permitted:

- As axes of a query
- In the WITH clause
- As an argument to an MDX function that uses a set

6.2 Tuple Values

Every tuple has a value (which might be null).

The value of a tuple is determined as follows:

1. DeepSee finds the rows in the fact table that correspond to all the non-measure members used in the tuple expression.
2. DeepSee then finds values for those rows as follows:
   - If the tuple expression includes a specific measure, DeepSee finds the value of that measure for each relevant row of the fact table.
   - If the tuple expression does not include a specific measure, DeepSee uses the default measure (typically, `%COUNT`).
3. DeepSee aggregates those values together, using the aggregation function specified for the measure.

For example, consider the following tuple:

```
(homed.32006,colord.red,allerd.[dairy products],MEASURES.[avg test score])
```

To determine the value of this tuple, DeepSee finds all the patients in the fact table that belong to the 32006 ZIP code, and whose favorite color is red, and who are allergic to dairy products. DeepSee then accesses the values for the `Test Score` measure for those patients and averages those values.

For another example, consider the following tuple (permitted in DeepSee MDX):

```
(allerd.soy,allerd.wheat)
```

To determine the value of this tuple, DeepSee counts the patients who are allergic to both soy and to wheat.

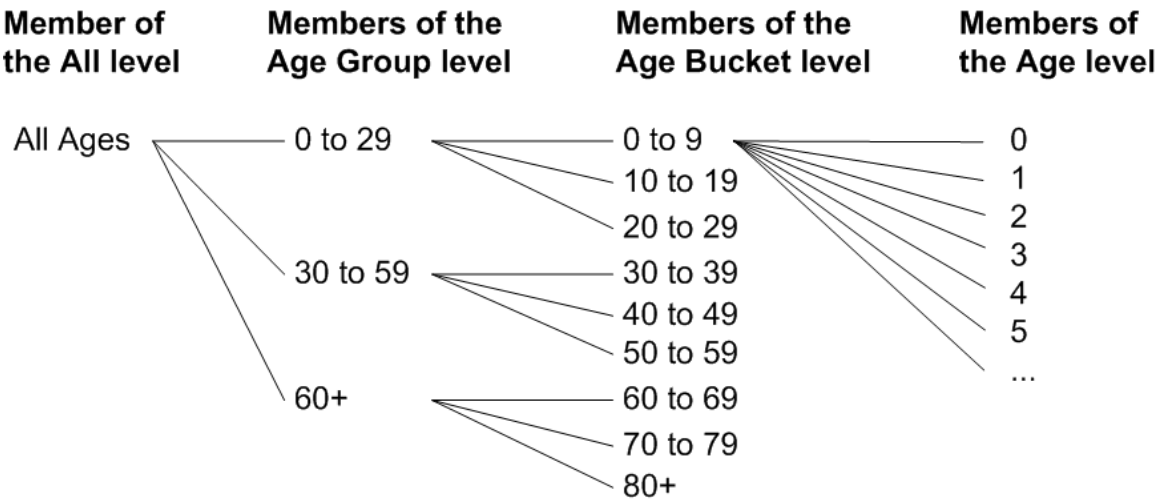
Finally, consider the following tuple:

```
(homed.juniper,homed.centerville)
```

To determine the value of this tuple, DeepSee counts the patients whose home city is Juniper and whose home city is Centerville. The value of this tuple is null, because each patient has one home city.

6.3 Example Tuple Expressions

A tuple expression can refer to a member at any level in any hierarchy of a dimension. Consider the following dimension (from the `Patients` cube) which includes one hierarchy with four levels:



You can create tuples that use members of any of these levels. For example, you can use any of the following tuple expressions:

```
(aged.[all patients])
(aged.[0 to 29])
(aged.5)
```

For another example, let us create variations of the preceding expressions. In this case, let us include members of other dimensions in the tuple expressions:

```
(aged.[all patients],gend.male)
(aged.[0 to 29],diagd.asthma)
(aged.5,allerd.soy,colord.red)
```

6.4 Using Sets of Tuples as Axes of a Query

You can use sets of tuples as axes of a query. The following example shows the simplest case, a set that consists of one tuple:

```
SELECT MEASURES.[%COUNT] ON 0, (homed.juniper,allerd.wheat,aged.[20 to 29]) ON 1 FROM demomdx

                                    %COUNT
Juniper->wheat->20 to 29                  1
```

The following example shows a set of tuples used as a query axis:

```
WITH SET myset AS
'{(homed.[cedar falls],allerd.soy,colord.red),(homed.magnolia,allerd.soy,colord.green),
(homed.34577,allerd.eggs,colord.green)}'
SELECT MEASURES.[%COUNT] ON 0, myset ON 1 FROM demomdx

                                    %COUNT
1 Cedar Falls->soy->Red                   *
2 Magnolia->soy->Green                    1
3 34577->eggs->Green                      *
```

For another example, the following is a valid query in DeepSee MDX:

```
WITH SET myset AS
'{(homed.[cedar falls],allerd.soy,colord.green),(colord.red,allerd.soy,homed.pine,gend.male)}'
SELECT MEASURES.[%COUNT] ON 0, myset ON 1 FROM demomdx

                                    %COUNT
1 Cedar Falls->soy->Green                 *
2 Red->soy->Pine->Male                    *
```

Finally, the following example uses tuples that refer multiple times to a single dimension:

```
SELECT MEASURES.[%COUNT] ON 0,
{(allerd.soy,allerd.wheat),(homed.juniper,homed.centerville)} ON 1 FROM demomdx
```

```
                                %COUNT
1 soy->wheat                         4
2 Juniper->Centerville               *
```

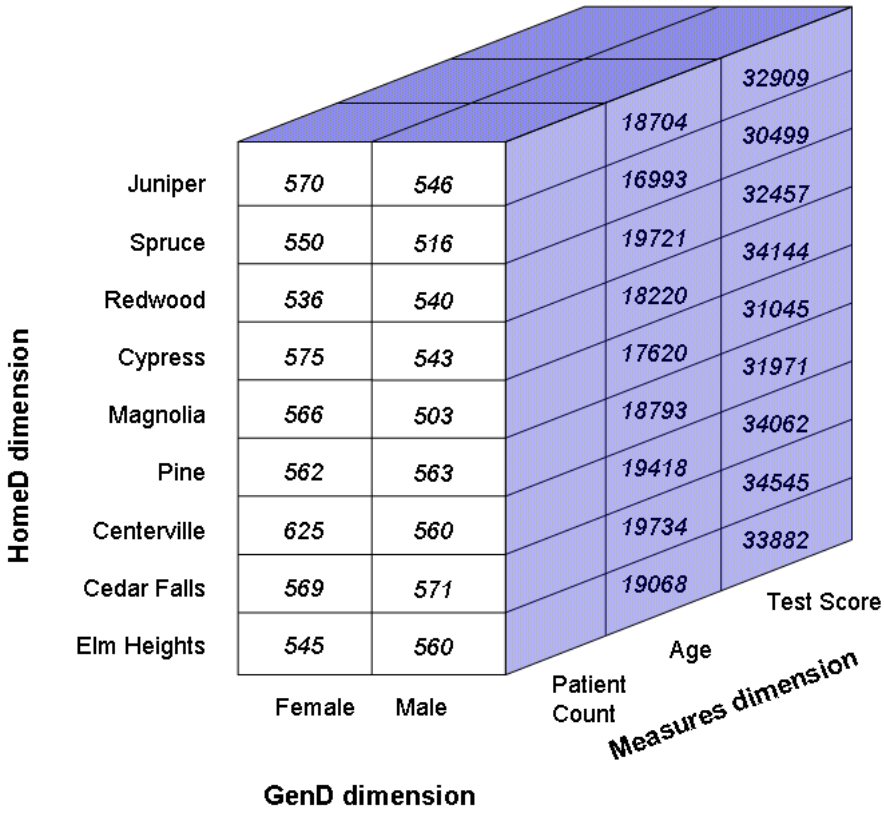
There are 4 patients who are allergic to both soy and wheat.

There are no patients with two home cities.

6.5 Introduction to Cubes

A cube is an n-dimensional structure that contains one *axis* (or *edge*) for each dimension. The cells of this cube are tuples. An MDX query retrieves specific tuples from the cube.

It is useful to visualize this cube, at least in simple cases. The `DemoMDX` cube has 10 dimensions (including the `Measures` dimension). For the sake of simplicity, the following figure shows three of those dimensions (`HomeD`, `GenD`, and `Measures`). Note that only three measures are actually shown.



Each axis of the cube is divided into segments, with one segment for each of the lowest-level members of the corresponding dimension. For the `HomeD` axis, these segments are the members of the `City` level.

Each cell in the cube is a fully qualified tuple. Each tuple has a value, as shown in the figure.

An MDX query is a request for a set of tuples, each of which has a value. Consider the following query:

```
SELECT CROSSJOIN(MEASURES.[%COUNT],gend.gender.MEMBERS) ON 0, homed.city.MEMBERS ON 1 FROM demomdx
```

	Female	Male
1 Cedar Falls	569	571
2 Centerville	625	560
3 Cypress	575	543
4 Elm Heights	545	560
5 Juniper	570	546
6 Magnolia	566	503
7 Pine	562	563
8 Redwood	536	540
9 Spruce	550	516

For this query, DeepSee finds the relevant tuples in the cube and obtains their values. For example, the first tuple is `(homed.[cedar falls],gend.female,measures.[%COUNT])`. The value of this tuple is 569.

Each measure that is aggregated by addition (such as `Age`) is contained directly in the cube. For other measures, MDX uses values from the cube and aggregates them as specified in the measure definition.

For example, the `Avg Age` measure is not contained directly in the cube, but the `Age` measure is; the `Age` measure contains the cumulative age of all the patients represented in a tuple. To calculate the `Avg Age` measure, MDX divides `Age` by `%COUNT`. Consider the following query:

```
SELECT CROSSJOIN(MEASURES.[avg age],gend.gender.MEMBERS) ON 0, homed.city.members ON 1 FROM demomdx
```

	Female	Male
1 Cedar Falls	36.90	34.56
2 Centerville	35.98	34.68
3 Cypress	37.02	33.55
4 Elm Heights	36.87	34.05
5 Juniper	38.09	34.26
6 Magnolia	35.64	35.03
7 Pine	36.64	33.38
8 Redwood	36.70	36.52
9 Spruce	37.90	32.93

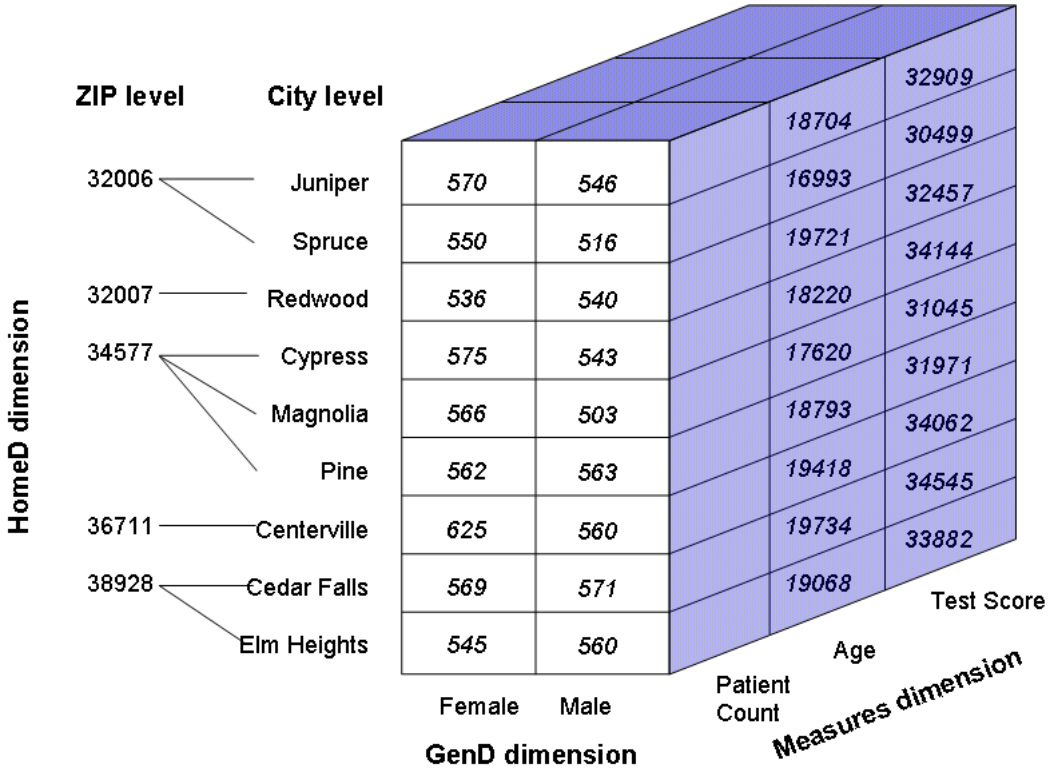
In this example, the second tuple is `(homed.[cedar falls],gend.male,measures.[avg age])`. To obtain this value, MDX divides the value of `(homed.[cedar falls],gend.male,measures.[age])` by the value of `(homed.[cedar falls],gend.male,measures.[%COUNT])` — 19734 divided by 571 is 34.56, as shown in the preceding results.

6.6 Higher Levels and a Cube Dimension

For now, we consider only dimensions that contain a single hierarchy.

For any dimension, only the lowest level is represented directly on the corresponding cube axis.

For example, the following figure shows all the levels of the `HomeD` dimension:



Notice that the `HomeD` axis includes only the leaf members of this dimension — that is, only the members of its lowest level. The higher levels consist of combinations of lower members. For example, each member of the `ZIP` level consists of one or more members of the `City` dimension.

Now consider the following query:

```
SELECT CROSSJOIN(MEASURES.[%COUNT],gend.gender.MEMBERS) ON 0, homed.zip.members ON 1 FROM demomdx
```

```
                              Female                Male
1 32006                        1,120               1,062
2 32007                          536                 540
3 34577                        1,703               1,609
4 36711                          625                 560
5 38928                        1,114               1,131
```

For this query, DeepSee finds the relevant tuples of the cube and obtains their values.

For example, the first tuple is `(homed.[32006],gend.female,measures.[%COUNT])`. The member 32006 consists of the cities Juniper and Spruce. This means that the tuple `(homed.[32006],gend.female,measures.[%COUNT])` consists of the combination of the following tuples:

- `(homed.[juniper],gend.female,measures.[%COUNT])`
- `(homed.[spruce],gend.female,measures.[%COUNT])`

These tuples have the values 570 and 550, respectively. The `%COUNT` measure is aggregated by adding, so the value for `(homed.[32006],gend.female,measures.[%COUNT])` is 1120.

6.7 Multiple Hierarchies in a Cube Dimension

A dimension can have multiple hierarchies. For a dimension that includes multiple hierarchies, the corresponding axis of the cube contains one segment for each member of the lowest level in each hierarchy.

Consider the following theoretical cube:

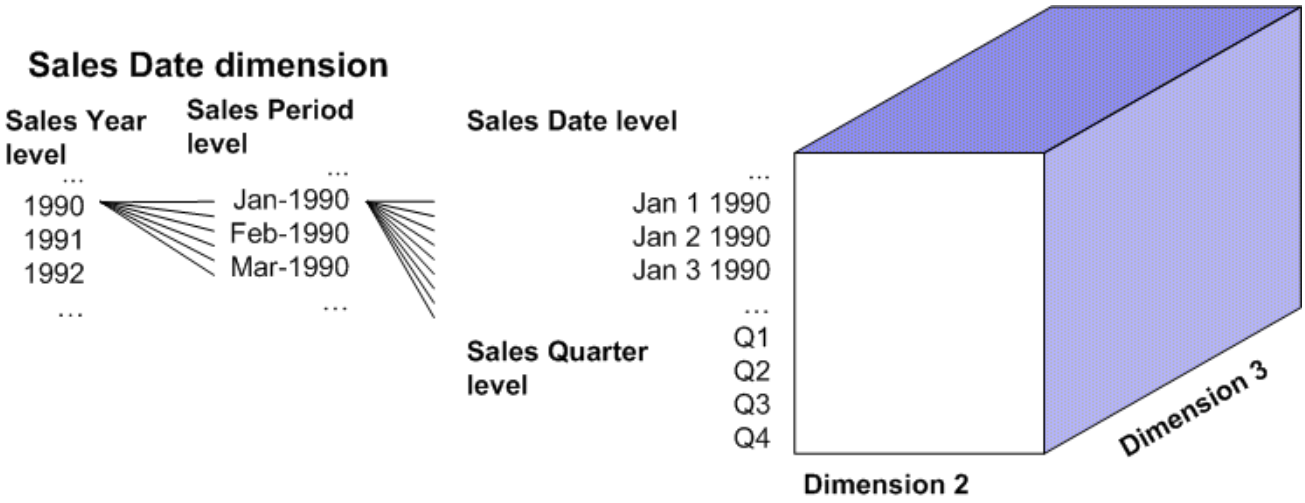
```
Members of cube 'theoretical':
---------------------------------------
...
Dimensions
...
  Sales Date
    H1
      Sales Year
      Sales Period
      Sales Date
    H2
      Sales Quarter
...
```

The `Sales Date` dimension contains two hierarchies. The `H1` hierarchy has three levels:

- The `Sales Year` level. For example, a members of this level is `1990`.
- The `Sales Period` level. For example, a members of this level is `Jan–1990`.
- The `Sales Date` level. For example, a members of this level is `Jan 3 1990`.

The other hierarchy contains only one level.

In this case, the `Sales Date` axis contains one segment for each member of `Sales Date` and one segment for each member of `Sales Quarter`. For example:



(For reasons of space, the picture of the cube is not divided into tuples.)

When a query uses, for example, the `Sales Quarter` level, DeepSee uses the appropriate part of this axis and accesses the requested tuples.

7

Filtering a Query

This chapter describes ways to filter data in MDX queries. It discusses the following topics:

- How to use the WHERE clause
- How to use the %NOT optimization
- How to use the %OR optimization

7.1 Introduction to the WHERE Clause

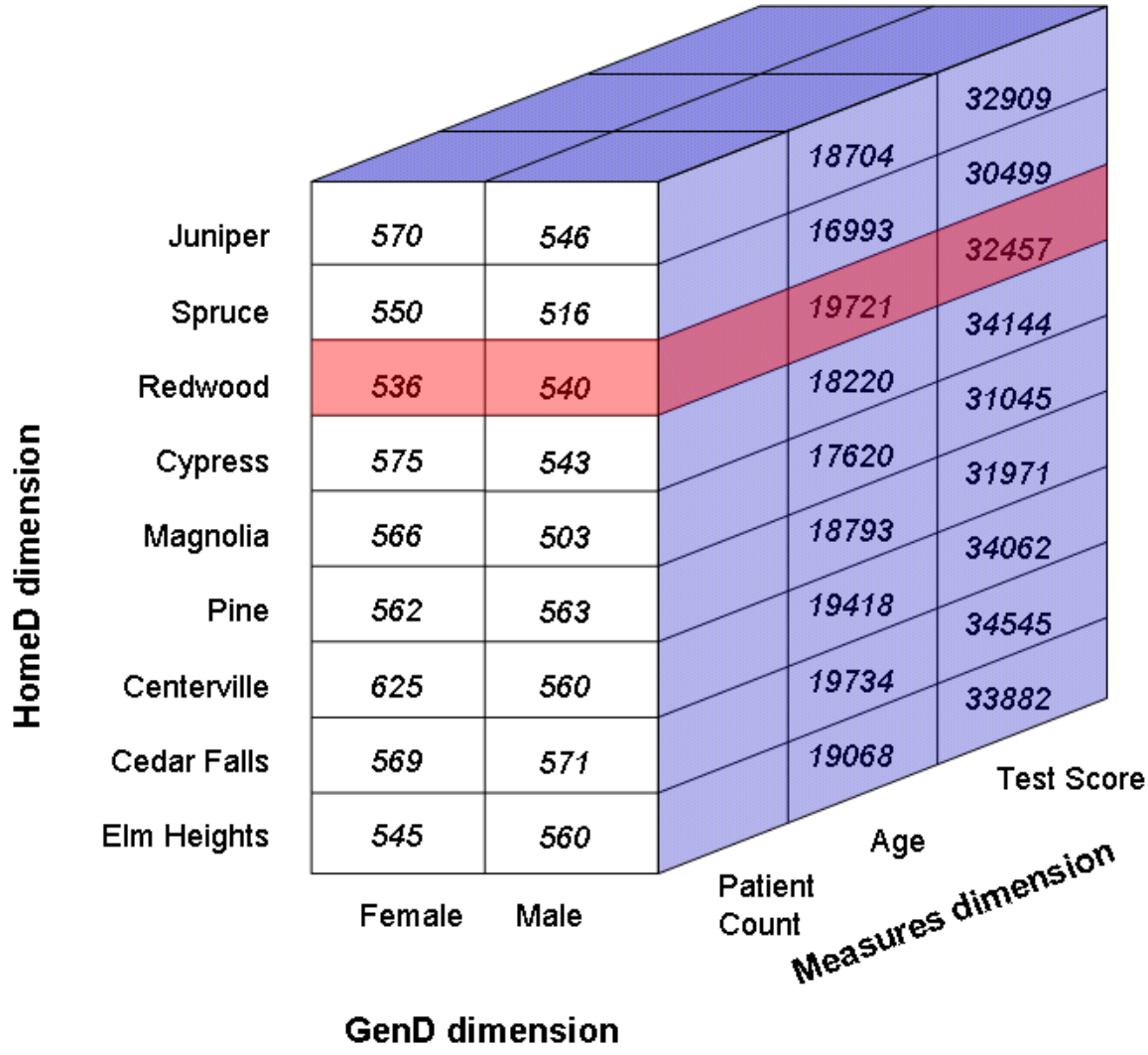
As noted in the section "Including a Simple Filter in a Query," earlier in this book, an MDX query itself can also include a filter (the WHERE clause). The WHERE clause of an MDX query is commonly referred to as the *slicer axis*. If the WHERE clause contains only one member, the system accesses only a slice of the cube.

For example, consider the following query:

```
SELECT {MEASURES.[%COUNT],MEASURES.[avg age]} ON 0, gend.gender.MEMBERS ON 1 FROM demomdx WHERE
homed.redwood

                        Patient Count              Avg Age
1 Female                          536                36.70
2 Male                            540                36.52
```

This query accesses only one slice of the cube, the slice of patients whose home city is Redwood. For example:



In this case, the Redwood slice is the only part of the cube that the query considers.

If the WHERE clause uses a set or a tuple, however, the phrase *slicer axis* is less useful, because in these cases, the cube is not truly being sliced.

7.1.1 Using a Set in the WHERE Clause

More generally, the WHERE clause can contain a set expression instead of a single member expression. In this case MDX combines the records with logical AND. For example, the following query uses only patients whose favorite color is red and patients who are male:

```
SELECT MEASURES.[%COUNT] ON 0, homed.city.MEMBERS ON 1 FROM demomdx WHERE{colord.red,gend.male}
```

```
                                  %COUNT
1 Cedar Falls                         66
2 Centerville                         72
3 Cypress                             76
4 Elm Heights                         81
5 Juniper                             74
6 Magnolia                            63
7 Pine                                71
8 Redwood                             72
9 Spruce                              58
```

In this case, the query uses the set `{colord.red,gend.male}`, which consists of two members. When DeepSee accesses the fact table, it finds the records associated with `colord.red` and the records associated with `gend.male` and it uses all those records.

Important: Each set element is used as a separate slicer axis, and the results of all the slicer axes (of all %FILTER clauses) are aggregated together. This is the process of *axis folding* (a filter is considered to be a query axis). Axis folding means that if a given source record has a non-null result for each slicer axis, that record is counted multiple times.

In axis folding, values are combined according to the aggregation method for that measure, as specified in the cube definition. (In the examples here, `%COUNT` is added.)

For more details, see "Axis Folding" in the appendix "How the DeepSee Query Engine Works" in the *DeepSee Implementation Guide*.

The next section discusses how to filter queries in yet another way.

7.1.2 Using Tuples in the WHERE Clause

In the WHERE clause, you can instead specify a single tuple or a set of tuples. For example:

```
SELECT MEASURES.[%COUNT] ON 0, NON EMPTY homed.city.MEMBERS ON 1 FROM demomdx
WHERE (aged.[age group].[60 +],gend.male)

                                    %COUNT
1 Cedar Falls                            7
2 Centerville                            9
3 Cypress
      12
4 Elm Heights                           14
5 Juniper                                8
6 Magnolia                               9
7 Pine                                   7
8 Redwood                                6
9 Spruce                                 2
```

For another example:

```
WITH SET myset as '{(aged.[age group].[60 +],diagd.chd),(aged.[age group].[60+],diagd.asthma)}'
SELECT MEASURES.[%COUNT] ON 0, NON EMPTY homed.city.MEMBERS ON 1 FROM demomdx WHERE myset

                                    %COUNT
1 Cedar Falls                            5
2 Centerville                            5
3 Cypress                                8
4 Elm Heights                            3
5 Juniper                                3
6 Magnolia                               5
7 Pine                                   2
8 Redwood                                5
```

When you filter the query itself, it is often useful to use the NON EMPTY keyword, so that the query returns only the non-null values. Include this keyword at the start of any set expression that might return a null value. For example:

```
SELECT MEASURES.[%COUNT] ON 0, NON EMPTY homed.city.MEMBERS ON 1 FROM demomdx
WHERE (aged.[age bucket].[30 to 39],diagd.chd)

                                    %COUNT
1 Elm Heights                            1
2 Magnolia                               1
```

In contrast, if we did not use NON EMPTY, the result would be as follows:

```
SELECT MEASURES.[%COUNT] ON 0, homed.city.MEMBERS ON 1 FROM demomdx
WHERE (aged.[age bucket].[30 to 39],diagd.chd)
```

```
                                    %COUNT
1 Cedar Falls                            *
2 Centerville                            *
3 Cypress                                *
4 Elm Heights                            1
5 Juniper                                *
6 Magnolia                               1
7 Pine                                   *
8 Redwood                                *
9 Spruce                                 *
```

7.2 The %NOT Optimization

It is often necessary to exclude a single member of a level. To do this easily, you can use the %NOT function, which is an InterSystems extension:

```
SELECT aged.[age bucket].MEMBERS ON 1 FROM patients WHERE aged.[age group].[0 to 29].%NOT
```

```
 1 0 to 9                                 *
2 10 to 19                               *
3 20 to 29                               *
4 30 to 39                             166
5 40 to 49                             139
6 50 to 59                             106
7 60 to 69                              86
8 70 to 79                              62
9 80+                                   41
```

Queries that use the %NOT function run more quickly than equivalent queries that use EXCEPT.

7.3 The %OR Optimization

Often it is necessary for the WHERE clause to refer to multiple members. For example:

```
SELECT gend.MEMBERS ON 1 FROM patients WHERE {allerd.[ant bites],allerd.soy,allerd.wheat}
```

```
1 Female                                56
2 Male                                  59
```

This query construction, however, means that DeepSee evaluates the query results multiple times (once for each item in the WHERE clause) and then combines them. This can be undesirably slow and can double-count items. (In this example, a given patient can be counted as many as three times, once for each allergy in the WHERE clause.)

With the %OR function, you can rewrite the query as follows:

```
SELECT gend.MEMBERS ON 1 FROM patients WHERE %OR({allerd.[ant bites],allerd.soy,allerd.wheat})
```

```
1 Female                                55
2 Male                                  57
```

Note the numbers are lower, because this query does not double-count any patients. Also, this query is faster than the preceding.

As of release 2014.1, you can use %OR with a set that contains members of different levels (or even that contains tuples). For example:

```
SELECT NON EMPTY [Measures].[%COUNT] ON 0 FROM [Patients]
WHERE %OR({[AgeD].[H1].[Age Bucket].&[80+],[DiagD].[H1].[Diagnoses].&[CHD]})
```

```
                              Patient Count
                                         71
```

8

Adding Summaries

This chapter describes how to add summaries (such as averages and totalsxt) to your MDX queries. It discusses the following topics:

- Introduction to the main functions you use to add summaries
- How to add a summary line to your query

8.1 Introduction to Summary Functions

MDX includes functions that summarize a given value, across a given set. For each function, the arguments are a set and an optional numeric expression (such as a reference to a measure). DeepSee evaluates the expression for each member of the set and then returns a single value. If no numeric expression is given, DeepSee instead evaluates the measure used in the query (possibly %COUNT).

The functions are as follows:

- SUM, which returns the sum of the values.
- AVG, which returns the average value. This function ignores members for which the expression is null.
- MAX, which returns the maximum value.
- MIN, which returns the minimum value.
- MEDIAN, which returns the value from the set that is closest to the median value.
- STDDEV, which returns the standard deviation of the values.
- STDDEVP, which returns the population standard deviation of the values.
- VAR, which returns the variance of the values.
- VARP, which returns the population variance of the values.

For example:

```
SELECT MAX(diagd.diagnoses.MEMBERS,MEASURES.[%COUNT]) ON 0 FROM demomdx
```

```
                                        MAX
                                         828
```

This query shows the maximum value of the `%COUNT` measure for the members of the `Diagnoses` level.

For another example, use the same function without specifying its second argument. In this case, the query displays the %COUNT measure as a column:

```
SELECT MEASURES.[%COUNT] ON 0, MAX(diagd.diagnoses.MEMBERS) ON 1 FROM demomdx
```

```
                                    %COUNT
MAX                                     828
```

For another example, use the same function without specifying any measure in the query at all:

```
SELECT MAX(diagd.diagnoses.MEMBERS) ON 0 FROM demomdx
```

```
                                      MAX
                                       828
```

In this case, DeepSee uses %COUNT.

8.2 Adding a Summary Line

More typically, rather than displaying the summary value by itself, you include it in a query that shows all the values of the set. This process is analogous to adding a summary line (as a row or column) in a spreadsheet.

The following example shows the %COUNT measure for each diagnosis, followed by the maximum value for this measure across this set:

```
SELECT MEASURES.[%COUNT] ON 0,
{diagd.diagnoses.MEMBERS, MAX(diagd.diagnoses.MEMBERS,MEASURES.[%COUNT])} ON 1 FROM demomdx
```

```
                                    %COUNT
1 None                                  828
2 asthma                                 90
3 CHD                                    37
4 diabetes                               45
5 osteoporosis                           22
6 MAX                                   828
```

Notice that the system first computes the %COUNT measure for each member using the aggregation method defined for that measure. In this case, the patients are counted. The asthma member, for example, has a total %COUNT value of 90. The MAX function then obtains the largest value for this measure, across the set of diagnoses.

For another example:

```
SELECT {gend.gender.MEMBERS, AVG(gend.gender.MEMBERS,MEASURES.[%COUNT])} ON 0,
MEASURES.[%COUNT] ON 1 FROM demomdx
```

```
                            Female              Male               AVG
%COUNT                          488                512               500
```

When using the summary functions, you might find it convenient to use named sets, as described in the chapter "Working with Sets." For example, the following query is equivalent to the preceding one:

```
WITH SET genders AS 'gend.gender.MEMBERS'
SELECT {genders, AVG(genders,MEASURES.[%COUNT])} ON 0, MEASURES.[%COUNT] ON 1 FROM demomdx
```

Another way to add a summary line is to define a summary member that combines the displayed members. See "Adding a Summary Member," in the chapter "Creating and Using Calculated Measures and Members."

9

Creating Calculated Measures and Members

This chapter describes how to create and use calculated measures and members. It discusses the following topics:

- Overview
- How to create calculated members within a query
- MDX recipes for commonly needed calculated measures
- MDX recipes for commonly needed non-measure members

Note: Your cubes might contain additional calculated members that you can use in all queries; see *Defining DeepSee Models*.

9.1 Overview of Calculated Measures and Members

In MDX, you can create a calculated member, which is a member based on other members. You can define two kinds of calculated members: ones that are measures and ones that are not. (Remember that a measure is considered to be a member of the MEASURES dimension.)

- A *calculated measure* is based on other measures. For example, one measure might be defined as a second measure divided by a third measure.

  The phrase *calculated measure* is not a standard MDX phrase. This book uses the phrase for brevity.

- A *non-measure calculated member* typically aggregates together other non-measure members. Like other non-measure members, this calculated member is a group of records in the fact table.

  For example, suppose member A refers to 150 records in the fact table, and member B refers to 300 records in the fact table. Suppose that you create a member C that aggregates A and B together. Then member C refers to the relevant 450 records in the fact table.

9.2 Creating a Calculated Member

To create one or more calculated members within a query, use syntax as follows:

```
WITH with_clause1 with_clause2 ... SELECT query_details
```

Tip: Notice that you do not include commas between the clauses.

Where:

- Each expression *with_clause1*, *with_clause2*, and so on has the following syntax:

  ```
  MEMBER MEASURES.[new_measure_name] AS 'value_expression'
  ```

  Later sections of this chapter discuss *value_expression*.
- *query_details* is your MDX query.

Then your query can refer the calculated member by name in all the places where you can use other members.

For example:

```
WITH MEMBER MEASURES.avgage AS 'MEASURES.[age]/MEASURES.[%COUNT]'
SELECT MEASURES.avgage ON 0, diagd.diagnoses.members ON 1 FROM demomdx
```

```
                                    avgage
1 None                                33.24
2 asthma                              34.79
3 CHD                                 67.49
4 diabetes                            57.24
5 osteoporosis                        79.46
```

Note: This calculated member is a *query-scoped* calculated member; its scope is the query. For information on session-scoped calculated members, see "CREATE MEMBER Statement," in the *DeepSee MDX Reference*.

9.3 MDX Recipes for Calculated Measures

This section describes how to create MDX expressions for some commonly needed calculated measures:

- General combinations of other measures
- Percentages of aggregate values
- Count of distinct members
- Semi-additive measures
- Filtered measures
- Measures that refer to other time periods
- Measures that refer to other cells in a pivot table

9.3.1 Combinations of Other Measures

For a calculated measure, the value expression often has the form of a mathematical formula that combines measure expressions. For example:

```
(MEASURES.[measure A] + MEASURES.[measure B]) * 100
```

Or:

```
(MEASURES.[measure A] + MEASURES.[measure B])/MEASURES.[measure C]
```

More formally, in this expression, you can use the following elements:

- References to measures.
- Numeric literals. For example: `37`
- Percentage literals. For example: `10%`

  There must be no space between the number and the percent sign.
- Mathematical operators. DeepSee supports the standard mathematical operators: `+` (addition), `-` (subtraction), `/` (division), and `*` (multiplication). It also supports the standard unary operators: `+` (positive) and `-` (negative).

  You can also use parentheses to control precedence.

  For example: `MEASURES.[%COUNT] / 100`
- MDX functions that return numeric values, such as AVG, MAX, COUNT, and others.

  In addition to the functions already discussed, DeepSee supports several scalar functions: SQRT, LOG, and POWER.

Tip: The MDX function IIF is often useful in such expressions. It evaluates a condition and returns one of two values, depending on the condition. You can use this to avoid dividing by zero, for example.

9.3.2 Percentages of Aggregate Values

It is often necessary to calculate percentages of the total record count or percentages of other aggregate values. In such cases, you can use the %MDX function, which is an InterSystems extension. This function executes an MDX query, which should return a single value, and returns that value, which is unaffected by the context in which you execute the function. This means that you can calculate percentages with measures defined by value expressions like this:

```
100 * MEASURES.[measure A] / %MDX("SELECT FROM mycube")
```

For example:

```
WITH MEMBER MEASURES.PercentOfAll AS '100 * MEASURES.[%COUNT]/%MDX("SELECT FROM demomdx")'
SELECT MEASURES.PercentOfAll ON 0, diagd.MEMBERS ON 1 FROM demomdx
```

```
                            PercentOfAll
1 None                             84.56
2 asthma                            6.85
3 CHD                               3.18
4 diabetes                          4.89
5 osteoporosis                      2.21
```

9.3.3 Distinct Member Count

In some cases, for a given cell, you want to count the number of distinct members of some particular level. For example, the `DocD` dimension includes the level `Doctor`. We could count the number of unique doctors who are primary care physicians for any given set of patients. To do so, we define a calculated measure that uses the following *value_expression*:

```
COUNT([docd].[h1].[doctor].MEMBERS,EXCLUDEEMPTY)
```

We can use this measure in a query as follows:

```
WITH MEMBER MEASURES.[distinct doctor count] AS 'COUNT(docd.doctor.MEMBERS,EXCLUDEEMPTY)'
SELECT MEASURES.[distinct doctor count] ON 0, aged.[age bucket].MEMBERS ON 1 FROM demomdx
```

```
                          distinct doctor co
1 0 to 9                                  38
2 10 to 19                                38
3 20 to 29                                38
4 30 to 39                                40
5 40 to 49                                41
6 50 to 59                                40
7 60 to 69                                33
8 70 to 79                                31
9 80+                                     28
```

9.3.4 Semi-Additive Measures

A *semi-additive measure* is a measure that is aggregated across most but not all dimensions. For example, customers' bank balances cannot be added across time, because a bank balance is a snapshot in time. To create such measures, you can use the %LAST function, an InterSystems extension to MDX.

Consider the following measures:

- `Balance` is based on the source property `CurrentBalance` and is aggregated by summing.

  You would avoid aggregating this measure over time, because it would give incorrect results; that is, you should use this measure only in pivot tables that include a time level for rows or columns.

- `Transactions` is based on the source property `TxCount` and is aggregated by summing.

You can define a calculated measure called `LastBalance` and use the following *value_expression*:

```
%LAST(Date.Day.Members,Measures.Balance)
```

The %LAST function returns the last non-missing value for a measure evaluated for each member of the given set. In this case, it finds the last day that has a value and returns that value.

9.3.5 Filtered Measures (Tuple Measures)

A normal measure considers all records in the fact table for which the source value is not null. In some cases, you may want to define a filtered measure, which has the following behavior:

- The measure is null for certain records.
- For the other records, the measure has a value.

For a filtered measure (also informally called a *tuple measure*), use a *value_expression* like the following:

```
([MEASURES].[my measure],[DIMD].[HIER].[LEVEL].[member name])
```

In this case, *value_expression* is a tuple expression where:

- [MEASURES].[my measure] is the measure to use as a basis.
- [DIMD].[HIER].[LEVEL].[member name] is the member for which the measure value should be non-null.

For example, the Avg Test Score measure is the average test score considering all patients who have a non-null value for the test. Suppose that in addition to the Avg Test Score measure, your customers would like to see another column that just shows the average test scores for patients with coronary heart disease (the CHD diagnosis). That is, the customers would like to have the measure Avg Test Score - CHD. In this case, you can create a calculated measure that has the following *value_expression*:

```
(MEASURES.[avg test score],diagd.h1.diagnoses.chd)
```

For example:

```
WITH MEMBER MEASURES.[avg test score - chd] AS
'(MEASURES.[avg test score],diagd.h1.diagnoses.chd)'
SELECT MEASURES.[avg test score - chd] ON 0, aged.[age bucket].MEMBERS ON 1 FROM demomdx
```

```
                        avg test score - c
1 0 to 9                                  *
2 10 to 19                                *
3 20 to 29                                *
4 30 to 39                                *
5 40 to 49                            78.00
6 50 to 59                            75.75
7 60 to 69                            80.71
8 70 to 79                            83.33
9 80+                                 55.25
```

9.3.6 Measures for Another Time Period

It is often useful to view the value of a given measure for an earlier time period, while viewing a later time period. As an example, you can define a calculated measure called UnitsSoldPreviousPeriod and use the following *value_expression*:

```
([DateOfsale].[Actual].CurrentMember.PrevMember ,MEASURES.[units sold])
```

Because of how this measure is defined, it is meaningful only if you use the DateOfSale dimension on the other axis of the query. For example:

```
WITH MEMBER [MEASURES].[UnitsSoldPreviousPeriod] AS
'([DateOfsale].[Actual].CurrentMember.PrevMember ,MEASURES.[units sold])'
SELECT {[Measures].[Units Sold],[MEASURES].[UNITSSOLDPREVIOUSPERIOD]} ON 0,
[DateOfSale].[Actual].[MonthSold].Members ON 1 FROM [HoleFoods]
```

```
                          Units Sold          DateOfSale
 1 Jan-2009                       15                   *
 2 Feb-2009                       10                  15
 3 Mar-2009                       13                  10
 4 Apr-2009                       15                  13
 5 May-2009                       22                  15
...
```

Notice that the caption of the second column is based on the dimension used within the value expression, rather than the name of the calculated member that we defined. We can use the %LABEL function to provide a more suitable caption. For example:

```
WITH MEMBER [MEASURES].[UnitsSoldPreviousPeriod] AS
'([DateOfsale].[Actual].CurrentMember.PrevMember ,MEASURES.[units sold])'
SELECT {[Measures].[Units Sold],%LABEL([MEASURES].[UNITSSOLDPREVIOUSPERIOD],"Units (Prv Pd)","")} ON
0,
[DateOfSale].[Actual].[MonthSold].Members ON 1 FROM [HoleFoods]
```

```
                          Units Sold       Units (Prv Pd)
 1 Jan-2009                       15                    *
 2 Feb-2009                       10                   15
 3 Mar-2009                       13                   10
 4 Apr-2009                       15                   13
 5 May-2009                       22                   15
 6 Jun-2009                       17                   22
 7 Jul-2009                       24                   17
 8 Aug-2009                       30                   24
...
```

These examples use a time-based level, because this kind of analysis is common for time levels. You can, however, use the same technique for data levels.

9.3.7 Measures That Refer to Other Cells

It is often useful to refer to the value in a different cell of the pivot table. To do so, you can use the %CELL and %CELLZERO functions. Each of these functions returns the value of another cell of the pivot table, by position. If the given call has no value, %CELL returns null; in contrast, %CELLZERO returns zero.

These functions have many uses. For one example, you can use %CELL to calculate a running total (in this case, the cumulative inches of rainfall):

```
SELECT {MEASURES.[Rainfall Inches],%CELL(-1,0)+%CELL(0,-1)} ON 0, {dated.year.1960:1970} ON 1 FROM
cityrainfall
```

```
                     Rainfall Inches           Expression
 1 1960                       177.83                177.83
 2 1961                       173.42                351.25
 3 1962                       168.11                519.36
 4 1963                       188.30                707.66
 5 1964                       167.58                875.24
 6 1965                       175.23              1,050.47
 7 1966                       182.50              1,232.97
 8 1967                       154.44              1,387.41
 9 1968                       163.97              1,551.38
10 1969                       184.84              1,736.22
11 1970                       178.31              1,914.53
```

9.4 MDX Recipes for Non-Measure Calculated Members

This section provides recipes for non-measure calculated members for some common scenarios:

- Age members
- Members that combine a hardcoded set of members
- Members that combine members specified in a term list
- Members that combine ranges of dates
- Members that intersect other members

9.4.1 Defining Age Members

It is often useful to have members that group records by age. To define such members, use an existing time level and the special NOW member. For example, consider the `MonthSold` level in the HoleFoods sample. You could define a calculated member named `3 Months Ago` with the following *value_expression*:

```
[dateofsale].[actual].[monthsold].[now-3]
```

For example:

```
WITH MEMBER CalcD.[3 months ago] as '[dateofsale].[actual].[monthsold].[now-3]'
SELECT calcd.[3 months ago] ON 0, {MEASURES.[units sold], MEASURES.target} ON1 FROM holefoods
```

```
                            3 months ago
1 Units Sold                          37
2 Target                          254.00
```

9.4.2 Defining a Hardcoded Combination of Members

In many cases, it is useful to define a coarser grouping that combines multiple members of the same level. To do so, create a non-measure calculated member that has a *value_expression* of the following form:

```
%OR({member_expression, member_expression,...})
```

For example:

```
%OR({colord.red,colord.blue,colord.yellow})
```

Each non-measure member refers to a set of records. When you create a member that uses the %OR function, you create a new member that refers to all the records that its component members use.

For example:

```
WITH MEMBER CalcD.[primary colors] as '%OR({colord.red,colord.blue,colord.yellow})'
SELECT calcd.[primary colors] ON 0,
{MEASURES.[%COUNT], MEASURES.[avg test score]} ON 1 FROM demomdx
```

9.4.3 Defining a Combination of Members Defined by a Term List

Term lists provide a way to customize a DeepSee model without programming. A term list is a simple (but extendable) list of key and value pairs. You can use term lists in the multiple ways; one is to build a set of members, typically for use in a filter.

In this case, you use the %TERMLIST function and the %OR function; create a non-measure calculated member that has a *value_expression* of the following form:

```
%OR(%TERMLIST(term_list_name))
```

Where *term_list_name* is a string that evaluates to the name of a term list.

For example:

```
%OR(%TERMLIST("My Term List"))
```

This expression refers to all records that belong to any of the members indicated by the term list (recall that %OR combines the members into a single member).

The %TERMLIST function has an optional second argument; if you specify "EXCLUDE" for this argument, the function returns the set of all members of the level that are not in the term list.

9.4.4 Aggregating Ranges of Dates

Another useful form uses a range of members aggregated by %OR:

```
%OR(member_expression_1:member_expression_n)
```

The expression member_expression_1:member_expression_n returns all members from member_expression_1 to member_expression_n, inclusive. This form is particularly useful with time levels, because you can use it to express a range of dates in a compact form.

For time levels, you can also use the special NOW member. The following expression aggregates sales records from 90 days ago through today:

```
%OR(DateOfSale.DaySold.[NOW-90]:DateOfSale.DaySold.[NOW])
```

Or use the following equivalent form:

```
%OR(DateOfSale.DaySold.[NOW-90]:[NOW])
```

You can also use the PERIODSTODATE function to get a range of dates. For example, the following expression gets the range of days from the start of the current year to today and aggregates these days together:

```
%OR(PERIODSTODATE(DateOfSale.YearSold,DateOfSale.DaySold.[NOW]))
```

9.4.5 Defining a Member as an Intersection of Other Members

In some cases, typically when you define a filter, it is useful to define a member that is an intersection of members. Suppose that you need a filter like the following (which does not show literal syntax):

```
Status = "discharged" and ERvisit = "yes" and PatientClass="infant"
```

Also suppose that you need to use this filter in many places.

Rather than defining the filter expression repeatedly, you could define and use a calculated member. For this calculated member, specify **Expression** as follows:

```
%OR({member_expression,member_expression,...}
```

For example:

```
%OR({birthd.year.NOW,allersevd.[003 LIFE-THREATENING]}
```

The expression (birthd.year.NOW,allersevd.[003 LIFE-THREATENING]) is a tuple expression, which is the intersection of the member birthd.year.NOW and the member allersevd.[003 LIFE-THREATENING] — that is, all patients who were born in the current year and who have a life-threatening allergy.