Defining DeepSee Models

Version 2012.2
13 September 2017
Defining DeepSee Models
Caché Version 2012.2  13 September 2017
Copyright © 2017 InterSystems Corporation
All rights reserved.

InterSystems, InterSystems Caché, InterSystems Ensemble, InterSystems HealthShare, HealthShare, InterSystems TrakCare, TrakCare,
InterSystems DeepSee, and DeepSee are registered trademarks of InterSystems Corporation.

InterSystems iKnow, Zen, and Caché Server Pages are trademarks of InterSystems Corporation.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive,
Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems
Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed,
transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part,
without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent
set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation.
InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such
standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising
out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS
CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION
REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION,
COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion
and without notice, to make substitutions and modifications in the products and practices described in this document.

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** ................................................................................................................................. 1

**1 Introduction** ........................................................................................................................................ 3

1.1 Purpose of DeepSee ............................................................................................................................ 3
1.2 Introduction to DeepSee Models ......................................................................................................... 4
1.3 Introduction to the Model Development Process ............................................................................... 5
1.4 Introduction to the DeepSee Architect
   1.4.1 Class Viewer .......................................................................................................................... 6
   1.4.2 Model Viewer ......................................................................................................................... 7
   1.4.3 Details Area .......................................................................................................................... 8

2 Concepts .................................................................................................................................................. 11

2.1 Introduction to Cubes
   2.1.1 The Source Class of a Cube ............................................................................................... 12
2.2 Dimensions, Hierarchies, and Levels
   2.2.1 Levels and Members .............................................................................................................. 12
   2.2.2 Member Names and Keys ..................................................................................................... 13
   2.2.3 Source Values ..................................................................................................................... 13
   2.2.4 Hierarchies and Dimensions ............................................................................................ 13
   2.2.5 The All Level and the All Member ...................................................................................... 14
   2.2.6 List-based Levels .............................................................................................................. 14
   2.2.7 See Also............................................................................................................................ 175
2.3 Properties ........................................................................................................................................... 14
2.4 Measures ............................................................................................................................................ 15
2.5 Listings .............................................................................................................................................. 16
2.6 Calculated Members
   2.6.1 Calculated Measures ........................................................................................................... 16
   2.6.2 Non-measure Calculated Members ................................................................................... 17
   2.6.3 See Also............................................................................................................................ 175
2.7 Subject Areas ...................................................................................................................................... 17
2.8 Filters ................................................................................................................................................ 18
   2.8.1 Filter Mechanisms .............................................................................................................. 18
   2.8.2 Using Filters ...................................................................................................................... 21
2.9 How DeepSee Builds and Uses Fact Tables
   2.9.1 Structure of a Fact Table .................................................................................................... 21
   2.9.2 Populating the Fact Table .................................................................................................. 22
   2.9.3 Using a Fact Table ............................................................................................................ 23
2.10 How DeepSee Generates Listings ................................................................................................. 24
2.11 Options Related to Multiple Cubes ............................................................................................... 25

3 Principles and Recommendations ........................................................................................................... 27

3.1 Choosing a Base Table ..................................................................................................................... 27
3.2 Defining Measures Appropriately
   3.2.1 Measures from Parent Tables ............................................................................................. 28
   3.2.2 Measures from Child Tables .............................................................................................. 29
3.3 Defining Hierarchies Appropriately
   3.3.1 What Happens if a Hierarchy Is Inverted ........................................................................... 29
   3.3.2 Time Hierarchies .............................................................................................................. 30
3.4 Defining Member Keys and Names Appropriately ............................................................................ 30
14 Using Unstructured Data in Cubes ................................................................. 129
14.1 Overview of iKnow/Cube Integration ......................................................... 129
14.1.1 iKnow Terminology ............................................................................. 130
14.1.2 About iKnow Measures and Dimensions ........................................... 130
14.2 Setting Up the Aviation Events Demo ......................................................... 131
14.3 Loading and Updating iKnow Dictionaries .............................................. 132
14.3.1 Loading iKnow Dictionaries ............................................................... 132
14.3.2 Updating iKnow Dictionaries .............................................................. 134
14.4 Defining an iKnow Measure ..................................................................... 134
14.4.1 Alternative Technique: Using an Existing iKnow Domain ................. 135
14.5 Defining an iKnow Entity Dimension ....................................................... 136
14.6 Defining an iKnow Dictionary Dimension .............................................. 137
14.7 Adding Member Overrides to an Item Level ........................................... 138
14.8 Including an iKnow Summary Field in a Listing ..................................... 139
14.9 Including a Link from a Listing to the Full Unstructured Text ............... 139
14.10 Creating a Listing for Use in iKnow Content Analysis Plugin .............. 139
14.11 Generating Secondary Cubes ................................................................ 140
14.11.1 Entity Occurrence Cube .................................................................... 140
14.11.2 Matching Results Cube .................................................................... 141
14.12 iKnow Domain Management in DeepSee ............................................. 142

15 Defining Worksheets ..................................................................................... 145
15.1 Overview of Worksheets ........................................................................ 145
15.2 Accessing the Worksheet Builder ............................................................ 145
15.3 Defining a Worksheet ............................................................................. 146
15.4 Specifying Worksheet Options ............................................................... 147
15.5 Specifying Details for Columns and Rows ............................................ 148
15.6 Editing an Existing Worksheet ............................................................... 148
15.7 Accessing a Worksheet Cell Programmatically ..................................... 148

16 Defining Quality Measures ........................................................................ 151
16.1 Overview of Quality Measures ............................................................... 151
16.2 Introduction to the Quality Measure Manager ....................................... 152
16.3 Creating a Quality Measure .................................................................. 153
16.4 Specifying the Expression for a Quality Measure ............................... 154
16.4.1 How Groups and Elements Are Combined ....................................... 156
16.5 Editing Other Information for a Quality Measure ............................... 156
16.6 Checking the Expression for a Quality Measure .................................. 157
16.7 Deleting a Quality Measure .................................................................. 157

17 Defining Basic KPIs .................................................................................... 159
17.1 Introduction to KPIs .............................................................................. 159
17.1.1 Comparison to Pivot Tables .............................................................. 159
17.1.2 Requirements for KPI Queries .......................................................... 160
17.2 Choosing Between MDX and SQL ......................................................... 161
17.3 Structure of a KPI Result Set ................................................................. 161
17.4 Defining a KPI with a Hardcoded Query .............................................. 162
17.5 Specifying Class Parameters ................................................................. 164
17.6 Specifying Ranges and Thresholds for Speedometers ....................... 165
17.7 Disabling the %CONTEXT Filter ............................................................. 166

18 Defining KPIs with Filters and Listings ...................................................... 167
18.1 Introduction to Filters ............................................................................ 167
18.2 Creating Interoperable Filters ................................................................. 168
  18.2.1 Filter Syntax for Pivot Tables ........................................................... 168
  18.2.2 Ways to Create Interoperable Filters .............................................. 168
18.3 Defining KPI Filters .............................................................................. 169
  18.3.1 Declaring the Filters in a KPI ............................................................ 169
  18.3.2 Specifying the List Items in a KPI Filter .......................................... 170
18.4 Overriding a Query at Runtime ............................................................ 172
  18.4.1 Overriding an MDX Query ............................................................... 172
  18.4.2 Overriding an SQL Query ............................................................... 173
18.5 Accessing Filter Values ....................................................................... 173
  18.5.1 Property to Access Filter Values ..................................................... 173
  18.5.2 Methods to Access Filter Values ..................................................... 174
18.6 Combining Filter Values in an MDX Query ........................................ 175
18.7 MDX KPI Examples ............................................................................. 176
  18.7.1 MDX KPI Example 1 ......................................................................... 176
  18.7.2 MDX KPI Example 2 ......................................................................... 177
  18.7.3 MDX KPI Example 3 ......................................................................... 178
18.8 SQL KPI Examples .............................................................................. 179
  18.8.1 KPI Example 1 .................................................................................. 179
  18.8.2 KPI Example 2 .................................................................................. 180
18.9 Defining a Listing for a KPI ................................................................. 180
  18.9.1 Example .......................................................................................... 181

19 Defining Advanced KPIs ......................................................................... 183
  19.1 Defining Manual KPIs ......................................................................... 183
    19.1.1 Available Properties ....................................................................... 183
    19.1.2 Overriding KPI Properties ............................................................... 184
    19.1.3 Defining a Manual Query ............................................................... 184
  19.2 Defining Cacheable KPIs .................................................................... 185
  19.3 Defining Asynchronous KPIs ............................................................. 185

20 Defining Plugins ...................................................................................... 187
  20.1 Introduction ....................................................................................... 187
    20.1.1 How Plugins Can Be Used .............................................................. 187
  20.2 Requirements for a Simple Plugin ...................................................... 188
  20.3 Implementing %OnCompute() ............................................................. 189
  20.4 Indicating State of Completion ......................................................... 190
  20.5 Creating a Plugin for Multiple Cubes ................................................. 190
    20.5.1 Determining the Listing Fields Programmatically ....................... 191
  20.6 Defining a Calculated Member That Uses a Plugin ............................ 192

21 Using Advanced Features of Cubes and Subject Areas .......................... 193
  21.1 Specifying maxFacts in the Cube Definition ....................................... 193
  21.2 Restricting the Records Used in the Cube .......................................... 194
    21.2.1 buildRestriction Attribute .......................................................... 194
    21.2.2 %OnProcessFact() Callback ......................................................... 195
  21.3 Defining and Using Intermediate Expressions ................................... 195
    21.3.1 Example ....................................................................................... 196
  21.4 Manually Specifying the Members for a Level .................................. 197
    21.4.1 XML Reserved Characters ............................................................ 198
  21.5 Defining Computed Dimensions ....................................................... 199
    21.5.1 Example Computed Dimension .................................................... 199
About This Book

This book describes, for developers, how to define all the elements used in DeepSee queries: DeepSee cubes, subject areas, worksheets, quality measures, KPIs, and plugins. It includes the following sections:

- Introduction
- Concepts
- Principles and Recommendations
- Defining Cubes
- Compiling and Building Cubes
- Defining Dimensions, Hierarchies, and Levels
- Defining Properties
- Defining Measures
- Defining Listings
- Defining Calculated Members and Named Sets
- Defining Subject Areas
- Defining Shared Dimensions and Compound Cubes
- Defining Cube-Cube Relationships
- Using Unstructured Data in Cubes
- Defining Worksheets
- Defining Quality Measures
- Defining Basic KPIs
- Defining KPIs with Filters and Listings
- Defining Advanced KPIs
- Defining Plugins
- Using Advanced Features of Cubes and Subject Areas
- Reference Information for Cube Classes
- Reference Information for Subject Area Classes
- Reference Information for KPI and Plugin Classes
- Details for the Fact and Dimension Tables

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.
About This Book

- *Using MDX with DeepSee* introduces MDX and describes how to write MDX queries manually for use with DeepSee cubes.
- *DeepSee MDX Reference* provides reference information on MDX as supported by DeepSee.

The following books are for both developers and users:
- *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.

For general information, see the *InterSystems Documentation Guide*. 
This chapter introduces DeepSee models.

1.1 Purpose of DeepSee

InterSystems DeepSee enables 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 like the following example:
The widgets on a dashboard are driven by pivot tables and KPIs (key performance indicators). For a pivot table, a user can display a listing, which displays source values.

Pivot tables, KPIs, and listings are queries and are executed at runtime:

- A pivot table can respond to runtime input such as filter selections made by the user. Internally it uses an MDX (MultiDimensional eXpressions) query that communicates with a DeepSee cube.

  A cube consists of a fact table and its indices. A fact table consists of a set of facts (rows), and each fact corresponds to a base record. For example, the facts could represent patients or departments.

  Depending on your configuration and implementation, DeepSee detects changes in your transactional tables and propagates them to the fact tables as appropriate.

  DeepSee generates an MDX query automatically when a user creates the pivot table in the Analyzer.

- A KPI can also respond to runtime user input. Internally, it uses either an MDX query (with a DeepSee cube) or an SQL query (with any table or tables).

  In either case, you create the query manually or copy it from elsewhere.

- A listing displays selected values from the source records used for the rows of the pivot table that the user has selected. Internally, a listing is an SQL query.

  You can specify the fields to use and let DeepSee generate the actual query. Or you can specify the entire query.

### 1.2 Introduction to DeepSee Models

A model includes some or all of the following elements:

- At least one cube definition. A cube describes ways that you can query a set of specific base elements (such as patients or transactions). A cube includes levels, which enable you to group records from the base set, and measures, which show aggregate values of those records. It also defines listings and other items.

  When you create pivot tables, you use levels and measures. Consider the following pivot table:

  ![Pivot Table Example]

  In this pivot table, the rows correspond to the members of the Patient Group level; each member is shown as one row. The data column displays the aggregate value of the Avg Test Score measure for each of these members; for this measure, the system computes the average value. Notice that the Avg Test Score is null for the None patient group.

- Any number of subject areas. A subject area is a subcube that enables users to focus on smaller sets of data without the need for multiple cubes. A subject area also enables you to customize captions and defaults of the cube.

- Any number of KPIs.

  In DeepSee, a KPI is an interactive dataset that can be displayed on a dashboard. The KPI can define actions, which a user can launch and which execute your custom code.

You use these elements to create dashboards as follows:

- Within the DeepSee Analyzer, you create pivot tables.
Each pivot table is a query that you create by drag and drop actions. The query runs against a cube or subject area.

- Within the Dashboard Designer, you add pivot tables and KPIs to dashboards, along with any filter controls or other needed controls.

### 1.3 Introduction to the Model Development Process

The model development process typically is as follows:

1. Create, compile, and build a basic cube with only a few items.
2. Use the Analyzer or the DeepSee shell to examine the results and to identify changes to make.
   - For information on the Analyzer, see *Using the DeepSee Analyzer*.
   - For information on the shell, see *Getting Started with DeepSee*.
3. Repeat the preceding steps as necessary.
4. When the cube is finalized or nearly finalized, define subject areas based on the cube.

Note that you might need multiple cubes.

### 1.4 Introduction to the DeepSee Architect

You use the Architect to create cubes and subject areas.

To access the Architect, do the following:

1. Click the InterSystems Launcher and then click *Management Portal*.
   - Depending on your security, you may be prompted to log in with a Caché username and password.
2. Switch to the appropriate namespace as follows:
   - Click *Switch*.
   - Click the namespace.
   - Click *OK*.
3. Go to the page [System] > [DeepSee] > [Architect].
   - When you first display the Architect, you see the following:
4. Click **Open**, click **Cubes**, click **Patients**, and then click **OK**.

Now the system displays the following:

The top area contains navigation links and buttons to perform different tasks.

Below that, the page consists of the following areas, when the Architect is displaying a cube.

1.4.1 **Class Viewer**

The left area is the Class Viewer, and it shows the properties in the base class used by the cube; this area is not shown for a subject area. For example:
The following rules control the display of a class in the Architect:

- All properties are shown except for relationship properties.
- This display is recursive; that is, properties of properties are shown.
- If a property is a collection (a list or an array) or a relationship, it is shown as a folder that displays the properties of the class used in the collection or relationship.
- If a property is of type %List (which is the object equivalent of $LISTBUILD), it is not shown as a folder.
  For example, see the DiagnosesAsLB property, which is included in the Patients sample to illustrate this point.
- If a class is not accessible from the base class via cascading dot syntax, it is not shown.
  For example, the Architect does not display DeepSee.Study.PatientDetails or DeepSee.Study.PatientEncounter.
- The Architect displays properties inherited from superclasses. (The sample does not demonstrate this.)

All core cube elements are based either on a source property or on a source expression (which is a Caché ObjectScript expression) that can use properties of any class.

**Important:** The Architect provides a useful view of the class properties, which makes it very easy to create DeepSee elements based on those properties. It is important, however, to know that although this view provides a convenient way to access some properties, you can also use a source expression to access any data. These source expressions are evaluated when the cube is built and thus do not affect your runtime performance.

### 1.4.2 Model Viewer

The center area is the *Model Viewer*, and it shows the current contents of the cube. For example:
Here you can select items for editing, including the cube itself in the first row. You can also delete an item by clicking the Delete button in the row for that item.

1.4.3 Details Area

The right area is the Details Area, and it shows details for the element that is currently selected in Model Viewer (if any), or for the cube (if nothing is selected).

For example:
To hide this area, click the Hide Details button ▶. If you do so, the Model Viewer is widened.

Then, to display this area again, click the Show Details button ◀.

In this area, you primarily work on the Details tab. The Tools tab provides quick access to other tools, not documented this book. This tab looks like this:
2

Concepts

This chapter explains the most important concepts in DeepSee models (apart from KPIs and plugins, which are discussed later in this book). It discusses the following topics:

- Cubes
- Dimensions, hierarchies, and levels
- Properties
- Measures
- Listings
- Calculated members
- Subject areas
- Filters
- How DeepSee builds and uses fact tables
- How DeepSee generates listings
- Options applicable when you have multiple cubes

2.1 Introduction to Cubes

A cube is an MDX concept and it defines MDX elements for use in the Analyzer. These elements determine how you can query the data, specifically, a set of specific records (such as patient records or transaction records). The set of records is determined by the source class for the cube.

A cube can contain all the following definitions:

- Levels, which enable you to group records
- Hierarchies, which contain levels
- Dimensions, which contain hierarchies
- Level properties, which are values specific to the members of a level
- Measures, which show aggregate values of those records
- Listings, which are queries that enable you to access the source data
• Calculated members, which are members based on other members
• Named sets, which are reusable sets of members or other MDX elements

The following sections discuss most of these items. For information on named sets, see Using MDX with DeepSee.

### 2.1.1 The Source Class of a Cube

In most cases, the source class for a cube is a persistent class.

The source class can also be a data connector, which is a class that extends %DeepSee.DataConnector. A data connector maps the results of an arbitrary SQL query into an object that can be used as the source of a cube. Typically, a data connector accesses external non-Caché data, but you can also use it to specify an SQL query against Caché, including an SQL query on a view. See “Defining and Using Data Connectors” in the DeepSee Implementation Guide.

The source class can also be a child collection class.

### 2.2 Dimensions, Hierarchies, and Levels

This section discusses dimensions, hierarchies, and levels.

#### 2.2.1 Levels and Members

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

Most pivot tables simply display data for the members of one or more levels. For example, the Age Group level has the members 0 to 29, 30 to 59, and 60+. The following pivot table shows data for these members:

<table>
<thead>
<tr>
<th>Age Group</th>
<th>Patient Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 29</td>
<td>4,255</td>
</tr>
<tr>
<td>30 to 59</td>
<td>4,119</td>
</tr>
<tr>
<td>60+</td>
<td>1,826</td>
</tr>
</tbody>
</table>

For another example, the following pivot table shows data for the Age Group and Gender levels (shown as the first and second columns respectively).

<table>
<thead>
<tr>
<th>Age Group</th>
<th>Gender</th>
<th>Patient Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 29</td>
<td>Female</td>
<td>2,073</td>
</tr>
<tr>
<td></td>
<td>Male</td>
<td>2,182</td>
</tr>
<tr>
<td>30 to 59</td>
<td>Female</td>
<td>2,030</td>
</tr>
<tr>
<td></td>
<td>Male</td>
<td>2,089</td>
</tr>
<tr>
<td>60+</td>
<td>Female</td>
<td>639</td>
</tr>
<tr>
<td></td>
<td>Male</td>
<td>687</td>
</tr>
</tbody>
</table>

You can also drag and drop individual members for use as rows or columns. For example:
For more details on the options, see *Using the DeepSee Analyzer*.

### 2.2.2 Member Names and Keys

Each member has both a name and an internal key, neither of which is required to be unique, even within the same level. Duplicate member names are legitimate in some cases. For example, different doctors can have the same name, and you would not want to combine them into a single member.

When a user drags and drops a member, DeepSee uses the member key, rather than the name, in its generated query. This means that duplicate member names do not necessarily pose a problem. Duplicate member keys, however, do make it difficult to refer to all the individual members.

For information on the scenarios in which you might have duplicate member names and duplicate member keys, see “Defining Member Keys and Names Appropriately,” in the next chapter.

### 2.2.3 Source Values

Each level is based on a *source value*, which is either a class property or a Caché ObjectScript expression. For example, the *Gender* level is based on the *Gender* property of the patient. For another example, the *Age Group* level is based on an expression that converts the patient’s *Age* property to a string (0 to 29, 30 to 59, or 60+), depending on the age.

### 2.2.4 Hierarchies and Dimensions

In DeepSee, 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 postal codes, and postal codes into countries.

There are three practical reasons to define hierarchies in DeepSee:

- DeepSee has optimizations that make use of them. For example, if you are displaying periods (year plus month) as rows or columns, and you then filter to a specific year, the query runs more quickly if your model defines years as the parent of periods.
- You can use hierarchies within a pivot table as follows: If you double-click a member of a level, DeepSee performs a drilldown to show the child members of that member, if any. For example, if you double-click a year, DeepSee drills down to the periods within that year; for details, see *Using the DeepSee Analyzer*.
- MDX provides functions that enable you to work with hierarchies. For example, you can query for the child postal codes of a given country, or query for the other postal codes in the same country.

You can use these functions in handwritten queries; the Analyzer does not provide a way to create such queries via drag and drop.

A *dimension* contains one or more parent-child hierarchies that organize the records in a similar manner; for example, a single dimension might contain multiple hierarchies related to allergies. There is no formal relationship between two different hierarchies or between the levels of one hierarchy and the levels of another hierarchy. The practical purpose of a dimension
is to define the default behavior of the levels that it contains — specifically the All level, which is discussed in the next subsection.

2.2.5 The All Level and the All Member

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

For example, the AgeD dimension includes one hierarchy with levels as follows:

<table>
<thead>
<tr>
<th>Member of the All level</th>
<th>Members of the Age Group level</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Patients</td>
<td>0 to 29</td>
</tr>
<tr>
<td></td>
<td>30 to 59</td>
</tr>
<tr>
<td></td>
<td>60+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Members of the Age Bucket level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 9</td>
</tr>
<tr>
<td>10 to 19</td>
</tr>
<tr>
<td>20 to 29</td>
</tr>
<tr>
<td>30 to 39</td>
</tr>
<tr>
<td>40 to 49</td>
</tr>
<tr>
<td>50 to 59</td>
</tr>
<tr>
<td>60 to 69</td>
</tr>
<tr>
<td>70 to 79</td>
</tr>
<tr>
<td>80+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Members of the Age level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

For a given dimension, you specify whether the All member exists, as well as its logical name and its display name. Within this dimension, the All member is named All Patients.

2.2.6 List-based Levels

In DeepSee, unlike many other BI tools, you can base a level upon a list value. For example, a patient can have multiple diagnoses. The Diagnoses level groups patients by diagnosis. For example:

<table>
<thead>
<tr>
<th>Diagnoses</th>
<th>Patient Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>8,480</td>
</tr>
<tr>
<td>asthma</td>
<td>660</td>
</tr>
<tr>
<td>CHD</td>
<td>319</td>
</tr>
<tr>
<td>diabetes</td>
<td>508</td>
</tr>
<tr>
<td>osteoporosis</td>
<td>217</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>10,184</strong></td>
</tr>
</tbody>
</table>

For a list-based level, any given source record can belong to multiple members. The pivot table shown here includes some patients multiple times.

2.2.7 See Also

See the chapter “Defining Dimensions, Hierarchies, and Levels.”

2.3 Properties

Each level may define any number of properties. If a level has a property, then each member of that level has a value for that property; other levels do not have values for the property.
In the Patients sample, the City level includes the properties Population and Principal Export.

Each property is based on a source value, which is either a class property or a Caché ObjectScript expression. For the City level, the properties Population and Principal Export are based directly on class properties.

You can use properties in queries in much the same way that you use measures. For example, in the Analyzer, you can use properties as columns (this example shows two properties):

<table>
<thead>
<tr>
<th>City</th>
<th>Population</th>
<th>Principal Export</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cedar Falls</td>
<td>90,000</td>
<td>iron</td>
</tr>
<tr>
<td>Centerville</td>
<td>49,000</td>
<td>video games</td>
</tr>
<tr>
<td>Cypress</td>
<td>3,000</td>
<td>gravel</td>
</tr>
<tr>
<td>Elm Heights</td>
<td>33,194</td>
<td>lettuce</td>
</tr>
<tr>
<td>Juniper</td>
<td>10,333</td>
<td>wheat</td>
</tr>
<tr>
<td>Magnolia</td>
<td>4,503</td>
<td>bundt cake</td>
</tr>
<tr>
<td>Pine</td>
<td>15,060</td>
<td>spaghetti</td>
</tr>
<tr>
<td>Redwood</td>
<td>29,152</td>
<td>peaches</td>
</tr>
<tr>
<td>Spruce</td>
<td>5,900</td>
<td>mud</td>
</tr>
</tbody>
</table>

In contrast to measures, properties cannot be aggregated. A property is null for all levels except for the level to which it belongs.

See the chapter “Defining Properties.”

2.4 Measures

A cube also defines measures, which show aggregate values in the data cells of a pivot table.

Each measure is based on a source value, which is either a class property or a Caché ObjectScript expression. For example, the Avg Test Score measure is based on the patient’s TestScore property.

The definition of a measure also includes an aggregation function, which specifies how to aggregate values for this measure. Functions include SUM and AVG.

For example, the following pivot table shows the Patient Count measure and the Avg Test Score measure. The Patient Count measure counts the patients used in any context, and the Avg Test Score measure shows the average test score for the patients used in any context. This pivot table shows the value for these measures for the members of the Age Group level:

<table>
<thead>
<tr>
<th>Age Group</th>
<th>Patient Count</th>
<th>Avg Test Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 29</td>
<td>4,255</td>
<td>59.79</td>
</tr>
<tr>
<td>30 to 59</td>
<td>4,119</td>
<td>59.83</td>
</tr>
<tr>
<td>60+</td>
<td>1,626</td>
<td>60.09</td>
</tr>
</tbody>
</table>

See the chapter “Defining Measures.”
2.5 Listings

A cube can also contain listings. Each listing has a name and specifies the fields to display when the user requests that listing. The following shows an example:

<table>
<thead>
<tr>
<th>#</th>
<th>PatientID</th>
<th>Age</th>
<th>Gender</th>
<th>Home City</th>
<th>Test Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SUBJ_100624</td>
<td>30</td>
<td>F</td>
<td>Elm Heights</td>
<td>73</td>
</tr>
<tr>
<td>2</td>
<td>SUBJ_101001</td>
<td>30</td>
<td>F</td>
<td>Juniper</td>
<td>91</td>
</tr>
<tr>
<td>3</td>
<td>SUBJ_101014</td>
<td>30</td>
<td>F</td>
<td>Elm Heights</td>
<td>87</td>
</tr>
<tr>
<td>4</td>
<td>SUBJ_101072</td>
<td>30</td>
<td>F</td>
<td>Redwood</td>
<td>70</td>
</tr>
</tbody>
</table>

The records shown depend on the context from which the user requested the listing.

If you do not define any listings in a cube, then if a user requests a listing, the Analyzer displays the following message:

Error $5001: %ExecuteListing: this cube does not support drill through

See the chapter “Defining Listings.”

2.6 Calculated Members

A calculated member is based on other members. You can define two kinds of calculated members:

- A calculated measure is a measure is based on other measures. (In MDX, each measure is a member of the Measures dimension.)
  
  For example, one measure might be defined as a second measure divided by a third measure.

  The phrase calculated measure is not standard in MDX, but this documentation uses it 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.

Calculated members are evaluated after the members on which they are based.

You can create calculated members of both kinds within the cube definition, and users can create additional calculated members of both kinds within the Analyzer.

2.6.1 Calculated Measures

It is very useful to define new measures based on other measures. For example, in the Patients sample, the Avg Test Score measure is defined as the Test Score measure divided by the Count measure. Consider the following pivot table:
When this pivot table is run, DeepSee determines the values for the Count and Test Score measures for each member of the Allergy Seversities level; a later section of this chapter describes how DeepSee does this. Then, for the Avg Test Score value for each member, DeepSee divides the Test Score value by the Count value.

2.6.2 Non-measure Calculated Members

For a non-measure calculated member, you use an MDX aggregation function to combine other non-measure members. The most useful functions are AGGREGATE and %OR, which are similar to each other.

Remember that each non-measure member refers to a set of records. When you combine multiple members into a new member, you create a member that refers to all the records that its component members use.

For a simple example, consider the ColorD dimension, which includes the members Red, Yellow, and Blue. These members access the patients whose favorite color is red, yellow, or blue, respectively. You can use %OR to create a single new member that accesses all three groups of patients.

For example:

You can use AGGREGATE to combine any members; for %OR, the members must belong to the same level.

2.6.3 See Also

See the chapter “Defining Calculated Members and Named Sets.”

2.7 Subject Areas

A subject area is a subcube with optional overrides to names of items. You define a subject area to enable users to focus on smaller sets of data without having to build multiple cubes. In a subject area, you can do the following:

• Specify a filter that restricts the data available in the subject area. For information on filters, see the next section.
  You can hardcode this filter, or you specify it programmatically, which means that you can specify it based on the $roles of the user, for example.

• Hide elements defined in the cube so that the Analyzer displays a subset of them.

• Define new names, captions, and descriptions for the visible elements.

• Specify the default listing for the subject area.
• Redefine or hide listings defined in the cube.
• Define new listings.

You can then use the subject area in all the same places where you can use a cube. For example, you can use it in the Analyzer, and you can execute MDX queries on it in the shell or via the API.

See the chapter “Defining Subject Areas.”

2.8 Filters

In BI applications, it is critical to be able to filter data in pivot tables and in other locations. This section discusses the filter mechanisms in DeepSee and how you can use them in your application.

2.8.1 Filter Mechanisms

DeepSee provides two simple ways to filter data: member-based filters and measure-based filters. You can combine these, and more complex filters are also possible, especially if you write MDX queries directly.

2.8.1.1 Member-based Filters

A member is a set of records. In the simplest member-based filter, you use a member to filter the pivot table (for example, other contexts are possible, as this section describes later). This means that the pivot table accesses only the records that belong to that member.

For example, consider the following pivot table, as seen in the Analyzer:

<table>
<thead>
<tr>
<th>Age Group</th>
<th>Patient Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 29</td>
<td>4,364</td>
</tr>
<tr>
<td>30 to 59</td>
<td>4,036</td>
</tr>
<tr>
<td>60+</td>
<td>1,600</td>
</tr>
<tr>
<td>Total</td>
<td>10,000</td>
</tr>
</tbody>
</table>

Suppose that we apply a filter that uses the 0 to 29 member of the Age Group level. The resulting pivot table looks like this:

<table>
<thead>
<tr>
<th>Age Group</th>
<th>Patient Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 29</td>
<td>4,364</td>
</tr>
</tbody>
</table>

The Analyzer provides options to display null rows and columns. If we display null rows, the pivot table looks like this:
We can use the same filter in any pivot table. For example, consider the following unfiltered pivot table:

<table>
<thead>
<tr>
<th>Favorite Color</th>
<th>Female</th>
<th>Male</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>1,187</td>
<td>1,220</td>
<td>2,407</td>
</tr>
<tr>
<td>Blue</td>
<td>669</td>
<td>566</td>
<td>1,235</td>
</tr>
<tr>
<td>Green</td>
<td>675</td>
<td>642</td>
<td>1,317</td>
</tr>
<tr>
<td>Orange</td>
<td>658</td>
<td>598</td>
<td>1,256</td>
</tr>
<tr>
<td>Purple</td>
<td>639</td>
<td>616</td>
<td>1,255</td>
</tr>
<tr>
<td>Red</td>
<td>665</td>
<td>606</td>
<td>1,271</td>
</tr>
<tr>
<td>Yellow</td>
<td>622</td>
<td>637</td>
<td>1,259</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5,115</strong></td>
<td><strong>4,885</strong></td>
<td><strong>10,000</strong></td>
</tr>
</tbody>
</table>

This pivot table shows the Patient Count measure although the headings do not indicate this. If we filter this pivot table in the same way as the previous one, we see this:

<table>
<thead>
<tr>
<th>Age Group</th>
<th>Favorite Color</th>
<th>Female</th>
<th>Male</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 29</td>
<td>None</td>
<td>494</td>
<td>579</td>
<td>1,073</td>
</tr>
<tr>
<td></td>
<td>Blue</td>
<td>263</td>
<td>257</td>
<td>520</td>
</tr>
<tr>
<td></td>
<td>Green</td>
<td>261</td>
<td>294</td>
<td>555</td>
</tr>
<tr>
<td></td>
<td>Orange</td>
<td>260</td>
<td>256</td>
<td>516</td>
</tr>
<tr>
<td></td>
<td>Purple</td>
<td>270</td>
<td>273</td>
<td>543</td>
</tr>
<tr>
<td></td>
<td>Red</td>
<td>310</td>
<td>280</td>
<td>590</td>
</tr>
<tr>
<td></td>
<td>Yellow</td>
<td>268</td>
<td>299</td>
<td>567</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>2,126</strong></td>
<td><strong>2,238</strong></td>
<td><strong>4,364</strong></td>
</tr>
</tbody>
</table>

Notice that the total record count is the same in both cases; in both cases, we are accessing only patients that belong to the 0 to 29 member.

You can also use multiple members together in a filter, and you can combine filters that refer to members of different levels. Also, rather than choosing the members to include, you can choose the members to exclude.

**Tip:** Member-based filters are so easy to create and so powerful that it is worthwhile to create levels whose sole purpose is for use in filters.

### 2.8.1.2 Measure-based Filters

DeepSee supports *searchable measures*. With such a measure, you can apply a filter that considers the values at the level of the source record itself.

For the Patients sample, you can have a filter that accesses only the patients who have an encounter count of 10 or higher. If we use this filter in a pivot table, we might see this:

<table>
<thead>
<tr>
<th>Age Group</th>
<th>Patient Count</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 29</td>
<td>3,246</td>
<td>3,246</td>
</tr>
<tr>
<td>30 to 59</td>
<td>3,305</td>
<td>3,305</td>
</tr>
<tr>
<td>60+</td>
<td>1,330</td>
<td>1,330</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>7,881</strong></td>
<td><strong>7,881</strong></td>
</tr>
</tbody>
</table>
If we use the same filter in a different pivot table, we might see this:

<table>
<thead>
<tr>
<th>Favorite Color</th>
<th>Female</th>
<th>Male</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>926</td>
<td>973</td>
<td>1,899</td>
</tr>
<tr>
<td>Blue</td>
<td>537</td>
<td>451</td>
<td>988</td>
</tr>
<tr>
<td>Green</td>
<td>530</td>
<td>523</td>
<td>1,053</td>
</tr>
<tr>
<td>Orange</td>
<td>515</td>
<td>454</td>
<td>969</td>
</tr>
<tr>
<td>Purple</td>
<td>495</td>
<td>486</td>
<td>981</td>
</tr>
<tr>
<td>Red</td>
<td>533</td>
<td>485</td>
<td>1,018</td>
</tr>
<tr>
<td>Yellow</td>
<td>469</td>
<td>504</td>
<td>973</td>
</tr>
<tr>
<td>Total</td>
<td>4,005</td>
<td>3,876</td>
<td>7,881</td>
</tr>
</tbody>
</table>

In both cases, the total patient count is the same, because in both cases, the pivot table uses only the patients who have at least 10 encounters.

### 2.8.1.3 More Complex Filters

It is possible to create more complex filters that combine member- and measure-based filters. The following shows an example of such a filter, as created in the Analyzer:

```
AND
Age Group IS 30 to 59
Measures.Encounter Count >= 10
```

Internally, the query does not use AND and OR, but instead uses MDX syntax. All DeepSee filters use MDX syntax.

You can also create filters that use MDX functions. For example:

- The FILTER function uses the aggregate values of a measure, rather than the lowest-level values which a measure-based filter uses. For example, you can use this to filter out patients who belong to cities that have fewer than 1000 patients.
  
  In the Analyzer, the Levels option for a row or column uses this function internally.

- The EXCEPT function can be used to remove specific members. DeepSee uses this function when you create a member-based filter that excludes your selected members.

  DeepSee MDX provides many other functions that perform set operations.

- The TOPCOUNT and other functions access members based on their ranking.

For an introduction to MDX and a survey of your options, see *Using MDX with DeepSee*. Also see *DeepSee MDX Reference*. 


2.8.2 Using Filters

When you define a pivot table, you can specify how it is filtered. In practice, however, it is undesirable to create multiple similar pivot tables with different filters, because the pivot tables can become difficult to maintain. Instead, you can use any or all of the following tools:

- In the Analyzer, you can define named filters, which you can then use in multiple pivot tables. A named filter is available in the Analyzer along with the contents of the cube or subject area (see the next item).

- In the Architect, you can define subject areas that are filtered views of a base cube. Then when you create pivot tables, you start with a subject area rather than with the cube itself. These pivot tables are always filtered by the subject area filter, in addition to any filters that are specific to the pivot tables themselves.

    In a subject area, you can specify a hardcoded filter, or you can customize a callback method to specify the filter at runtime (to base it on a value such as $roles$, for example).

- In the User Portal, when you create dashboards, you can include filter controls in them (this applies only to simple, member-based filters). Then the user can select the member or members to include or exclude.

Filters are always cumulative.

2.9 How DeepSee Builds and Uses Fact Tables

When you compile a cube, DeepSee generates a fact table and related tables. When you build a cube, DeepSee populates these tables and generates their indices. At runtime, DeepSee uses the fact table. This section describes this process. It includes the following topics:

- Structure of a fact table
- How DeepSee populates a fact table
- How DeepSee uses a fact table

DeepSee does not generate tables for subject areas. A subject area uses the tables that have been generated for the cube on which the subject area is based.

2.9.1 Structure of a Fact Table

A fact table typically has one record for each record of the base table; this row is a fact. The fact contains one field for each level and one field for each measure. The following shows a sketch:
The field for a given level might contain no values, a single value, or multiple values. Each distinct value corresponds to a member of this level. The field for any given measure contains either null or a single value.

When DeepSee builds this fact table, it also generates indices for it.

The fact table does not contain information about hierarchies and dimensions. In the fact table, regardless of relationships among levels, each level is treated in the same way: the fact table contains one column for each level, and that column contains the value or values that apply to each source record.

**Tip:** By default, the fact table has the same number of rows as the base table. When you edit a cube class in Studio, you can override its `OnProcessFact()` callback, which enables you to ignore selected rows of the base table. If you do so, the fact table has fewer rows than the base table.

### 2.9.2 Populating the Fact Table

When you build a cube, the system iterates through the records of the base table. For each record, the system does the following:

- Examines the definition of each level and obtains either no value, a single value, or multiple values.
  
  In this step, the system determines how to categorize the record.

- Examines the definition of each measure and obtains either no value or a single value.

The system then writes this data to the corresponding row in the fact table and updates the indices appropriately.

#### 2.9.2.1 Determining the Values for a Level

Each level is specified as either a source property or a source expression. Most source expressions return a single value for given record, but if the level is of type list, its value is a Caché list of multiple values.

For a given record in the base table, the system evaluates that property or expression at build time, stores the corresponding value or values in the fact table, and updates the indices appropriately.

For example, the **Age Bucket** level is defined as an expression that returns one of the following strings: 0–9, 10–19, 20–29, and so on. The value returned depends upon the patient’s age. The system writes the returned value to the fact table, within the field that corresponds to the **Age Bucket** level.

For another example, the **Allergy** level is a list of multiple allergies of the patient.

#### 2.9.2.2 Determining the Value for a Measure

When DeepSee builds the fact table, it also determines and stores values for measures. Each measure is specified as either a source property or a Caché ObjectScript source expression.

For a given row in the base table, the system looks at the measure definition, evaluates it, and stores that value (if any) in the appropriate measure field.

For example, the **Test Score** measure is based on the **TestScore** property of the patients.

#### 2.9.2.3 Determining the Value for a Property

When DeepSee builds the fact table, it also determines values for properties, but it does not store these values in the fact table. In addition to the fact table, the system generates a table for each level (except for age and time levels). When the system builds the fact table, it stores values for properties in the appropriate dimension tables.
2.9.3 Using a Fact Table

Consider the following pivot table:

<table>
<thead>
<tr>
<th>Age Bucket</th>
<th>Count</th>
<th>Test Score</th>
<th>Avg Test Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 9</td>
<td>1,410</td>
<td>82,860</td>
<td>58.77</td>
</tr>
<tr>
<td>10 to 19</td>
<td>1,471</td>
<td>88,229</td>
<td>59.98</td>
</tr>
<tr>
<td>20 to 29</td>
<td>1,374</td>
<td>83,297</td>
<td>60.62</td>
</tr>
<tr>
<td>30 to 39</td>
<td>1,529</td>
<td>90,769</td>
<td>59.36</td>
</tr>
<tr>
<td>40 to 49</td>
<td>1,487</td>
<td>89,755</td>
<td>60.36</td>
</tr>
<tr>
<td>50 to 59</td>
<td>1,103</td>
<td>65,909</td>
<td>59.75</td>
</tr>
<tr>
<td>60 to 69</td>
<td>745</td>
<td>44,056</td>
<td>59.14</td>
</tr>
<tr>
<td>70 to 79</td>
<td>569</td>
<td>34,698</td>
<td>60.98</td>
</tr>
<tr>
<td>80+</td>
<td>312</td>
<td>18,960</td>
<td>60.77</td>
</tr>
</tbody>
</table>

The first column displays the names of the members of the Age Bucket level. The first data column shows the Patient Count measure, the second data column shows the Test Score measure, and the last column shows the Avg Test Score measure. The Avg Test Score measure is a calculated member.

The system determines these values as follows:

1. The first row refers to the 0–9 member of the Age Bucket level. The system uses the indices to find all the relevant patients (shown here with red highlighting) in the fact table:

<table>
<thead>
<tr>
<th>Age Bucket</th>
<th>Allergies</th>
<th>level C</th>
<th>...</th>
<th>Test Score</th>
<th>meas B</th>
<th>meas C</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 to 49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 to 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 to 59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 to 19</td>
<td>art, bites, bee stings, soy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 to 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 to 19</td>
<td>dairy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30 to 39</td>
<td>art, bites, dairy products</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 to 29</td>
<td>art, bites, bee stings, eggs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80+</td>
<td>art, bites</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 to 29</td>
<td>soy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. In the pivot table, the Patient Count column shows the count of patients used in a given context.

For the first cell in this column, DeepSee counts the number of records in the fact table that it has found for the 0–9 member.

3. In the pivot table, the Test Score column shows the cumulative test score for the patients in a given context.

For the first cell in this column, DeepSee first finds the values for the Test Score in the fact table that it has found for the 0–9 member:
Then it aggregates those numbers together, in this case by adding them.

4. In the pivot table, the Avg Test Score column is meant to display the average test score for the patients in a given context.

The Avg Test Score measure is a calculated member, computed by dividing Test Score with Patient Count.

The system repeats these steps for all cells in the result set.

## 2.10 How DeepSee Generates Listings

This section describes how the system uses the listings defined in the cube.

Within a pivot table, a user selects one or more cells.

The user then clicks the Listing button, and the system displays a listing, which shows the values for the lowest-level records associated with the selected cells (also considering all filters that affect this cell):

To generate this display, the system:

1. Creates an temporary listing table that contains the set of source ID values that correspond to the facts used in the selected cells.
2. Generates an SQL query that uses this listing table along with the definition of your listing.
3. Executes this SQL query and displays the results.
Your cube can contain multiple listings (to show different fields for different purposes). When you create a pivot table in the Analyzer, you can specify which listing to use for that pivot table.

The listing query is executed at runtime and uses the source data rather than the fact table. Because of this, if the fact table is not completely current, it is possible for the listing to show a different set of records than you see in the fact table.

### 2.11 Options Related to Multiple Cubes

In many cases, it is useful to have multiple cubes. In such cases, you can also define shared dimensions, compound cubes, and cube relationships. For details, see the chapters “Defining Shared Dimensions and Compound Cubes” and “Defining Cube-Cube Relationships” and
This chapter discusses core principles and other recommendations for your DeepSee models:

- Choosing a base table
- Defining measures appropriately
- Defining hierarchies appropriately
- Defining member keys and names appropriately
- Avoiding very granular levels
- Using list-based levels carefully
- Handling null values appropriately
- Considerations when you define multiple cubes
- Other recommendations

### 3.1 Choosing a Base Table

When defining a cube, the first step is to choose the class to use as the base class for that cube. The key point to remember is this: Within the cube, any record count refers to a count of records in this class (as opposed to some other class referenced within the cube). Similarly, the selection of the base class determines the meaning of all measures in the cube.

For example:

- If the base class is **Transactions**, transactions are counted. In this cube, you can have measures like the following:
  - Average broker fee per transaction (or average for any group of transactions)
  - Average transaction value per transaction (or average for any group of transactions)

- If the base class is **Customers**, customers are counted. In this cube, you can have measures like the following:
  - Average broker fee per customer (or average for any group of customer)
  - Average transaction value per customer (or average for any group of customers)

You can have multiple cubes, each using a different base class, and you can use them together in dashboards.
3.2 Defining Measures Appropriately

The value on which a measure is based must be one-to-one with the records in the base table. Otherwise, DeepSee would not aggregate that measure in the way that you intend it to. This section demonstrates this principle.

3.2.1 Measures from Parent Tables

Do not base a measure on a field in a parent table. For example, consider the following two tables:

- **Order** — Each row represents an order submitted by a customer. The field SaleTotal represents the total monetary value of the order.
- **OrderItem** — Each row represents an item in that order. In this table, the field OrderItemSubtotal represents the monetary value of this part of the order.

Suppose that we use **OrderItem** as the base table. Also suppose that we define the measure **Sale Total**, based on the parent’s SaleTotal field. The goal for this measure is to display the total sale amount for all the sales of the selected order items.

Let us consider the contents of the fact table. The following shows an example:

![Table Example](image)

The first four rows represent the items in the same order. The next two rows represent the items of another order, and so on.

Suppose that this model has a dimension called **Item Type**. Let us examine what happens when DeepSee retrieves records for all items of type R:

<table>
<thead>
<tr>
<th>Item Type</th>
<th>dim B</th>
<th>...</th>
<th>Sale Total</th>
<th>meas B</th>
<th>meas C</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>type Q</td>
<td>nnnnn</td>
<td>nnnn</td>
<td>279.07</td>
<td>nnnnn</td>
<td>nnnnn</td>
<td>nnnnn</td>
</tr>
<tr>
<td>type R</td>
<td>nnnnn</td>
<td>nnnn</td>
<td>279.07</td>
<td>nnnnn</td>
<td>nnnnn</td>
<td>nnnnn</td>
</tr>
<tr>
<td>type R</td>
<td>nnnnn</td>
<td>nnnn</td>
<td>279.07</td>
<td>nnnnn</td>
<td>nnnnn</td>
<td>nnnnn</td>
</tr>
<tr>
<td>type S</td>
<td>nnnnn</td>
<td>nnnn</td>
<td>279.07</td>
<td>nnnnn</td>
<td>nnnnn</td>
<td>nnnnn</td>
</tr>
<tr>
<td>type R</td>
<td>nnnnn</td>
<td>nnnn</td>
<td>52.14</td>
<td>nnnnn</td>
<td>nnnnn</td>
<td>nnnnn</td>
</tr>
<tr>
<td>type T</td>
<td>nnnnn</td>
<td>nnnn</td>
<td>52.14</td>
<td>nnnnn</td>
<td>nnnnn</td>
<td>nnnnn</td>
</tr>
</tbody>
</table>

To compute the value of the **Sale Total** measure for type R, DeepSee adds together the three values shown here: 279.07, 279.07, and 52.14. But this action double-counts one of the orders.

Depending on the case, the **Sale Total** measure might aggregate correctly; that is, it might show the correct total sales figure for the selected order items. But you cannot ensure that the measure does this, because you cannot prevent double-counting as shown as in this example.
3.2.2 Measures from Child Tables

You can use a value in a child table as the basis of a measure, but to do so, you must aggregate that value across the relevant rows of the child table.

Consider the following two tables:

- **Customer** — Each row represents a customer.
- **Order** — Each row represents a customer order. The field **SaleTotal** represents the total monetary value of the order.

Suppose that we use **Customer** as the base table, and that we want to create a measure based on the **SaleTotal** field. Because a customer potentially has multiple orders, there are multiple values for **SaleTotal** for a given customer. To use this field as a measure, we must aggregate those values together. The most likely options are to add the values or to average them, depending on the purpose of this measure.

3.3 Defining Hierarchies Appropriately

In any hierarchy, pay attention to the order of the levels. A level is automatically a child of the previously listed level in that hierarchy. For example, consider the **HomeD** dimension from the Patients sample:

```
<table>
<thead>
<tr>
<th>HomeD</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
</tr>
<tr>
<td>ZIP</td>
</tr>
<tr>
<td>City</td>
</tr>
</tbody>
</table>
```

The **ZIP** level is the parent of the **City** level. More precisely, each member of the **ZIP** level is the parent of one or more members of the **City** level. (In reality, there is a many-to-many relationship between ZIP codes and cities, but the Patients sample is simplistic, and in this sample, ZIP codes represent larger areas than cities.)

In your hierarchies, the first level should be the least granular, and the last one should be the most granular.

Hierarchies in MDX are parent-child hierarchies. To enforce this rule, DeepSee considers the order of the levels in each hierarchy when it builds the cube, as follows:

- For the first level defined in a hierarchy, DeepSee creates a separate member for each unique source value.
- For the levels after that, DeepSee considers the source value for the level in combination with the parent member.

For example, suppose that the first level is **State**, and the second level is **City**. When it creates members of the **City** level, DeepSee considers both the city name and the state to which that city belongs.

The internal logic is slightly different for time hierarchies, but the intent is the same.

3.3.1 What Happens if a Hierarchy Is Inverted

In contrast, suppose that we move the **City** level so that it is before the **ZIP** level in the cube, recompile, and rebuild. If we use the **ZIP** level for rows in a pivot table, we see something like the following:
In this case, DeepSee has created more than one member with the same name, because it assumes that (for example) there are two ZIP 32006 codes, which belong to different cities.

In this case, it is obviously incorrect to have multiple members with the same name. In other scenarios, however, it is legitimate to have multiple members with the same name. For example, different countries can have cities with the same name.

### 3.3.2 Time Hierarchies

It is important to remember that MDX hierarchies are parent-child hierarchies. All the records for a given child member must be contained in the parent member as well.

For example, consider the January member of the Birth Month level; this includes all patients born in January of any year — patients born in 1920, 1921, and so on. No single year contains all these records; thus the Birth Month level cannot be the child of the Year level.

“Time Levels and Hierarchies,” later in this book, gives guidelines on appropriate time hierarchies.

### 3.4 Defining Member Keys and Names Appropriately

Each member has both a name and an internal key. By default, these are the same, except for time levels. Both of these identifiers are strings. When you define your model, you should consider the following items:

- It can be correct to have duplicate member names. Different people, for example, can have the same name. When a user drags and drops a member, DeepSee uses the member key, rather than the name, in its generated query.

- InterSystems recommends that you ensure that member keys are unique in each level. Duplicate member keys make it difficult to refer to all the individual members.

  In particular, if you have duplicate member keys, users cannot reliably drill down by double-clicking.

- To help users distinguish the members, if a level has multiple members with the same name, InterSystems also recommends that you add a property to the level whose value is the same as the key. To do so, simply base the property on the same source property or source expression that the level uses.

  Then the users can display the property to help distinguish the members.

The following sections describe the scenarios in which you might have duplicate member keys and names.
3.4.1 Ways to Generate Duplicate Member Keys

Duplicate member keys are undesirable. If you have duplicate member keys, users cannot reliably drill down by double-clicking.

Except for time levels, each unique source value for a level becomes a member key. (For time levels, the system uses slightly different logic to generate unique keys for all members.)

If there is a higher level in the same hierarchy, it is possible for DeepSee to generate multiple members with the same key. See “Ensuring Uniqueness of Member Keys,” later in this book.

3.4.2 Ways to Generate Duplicate Member Names

Duplicate member names may or may not be undesirable, depending on your business needs.

Except for time levels, each unique source value for a level becomes a member name, by default. (For time levels, the system generates unique names for all members.)

There are only two ways in which DeepSee can generate multiple members with the same name:

• There is a higher level in the same hierarchy. This higher level might or might not be suitable. See “Defining Hierarchies Appropriately,” earlier in this chapter.

• You have defined a level as follows:
  – The member names are defined separately from the level definition itself.
  – The member names are based on something that is not unique.

  See “Using Property Values as Member Names,” later in this book.

Note that Caché does not automatically trim leading spaces from string values as it builds the fact table. If the source data contains leading spaces, you should use a source expression that removes those. For example:

$ZSTRIP(%source.myproperty, "<W")

Otherwise, you will create multiple members with names that appear to be the same (because some of the names have extra spaces at the start).

3.5 Avoiding Very Granular Levels

For users new to DeepSee, it is common to define a level that has a one-to-one relationship with the base table. Such a level is valid but is not particularly useful, unless it is also combined with filters to restrict the number of members that are seen.

A very granular level has a huge number of members (possibly hundreds of thousands or millions), and DeepSee is not designed for this scenario.

For example, suppose that we modified the Patient sample to have a Patient level. Then we could have a pivot table like this:
This is valid but an SQL query would produce the same results more efficiently. Consider the processing that is described in “How DeepSee Builds and Uses Fact Tables,” in the previous chapter. (That description gives the conceptual flow rather than the actual processing, but the overall idea is the same.) That processing is intended to aggregate values together as quickly as possible. The pivot table shown above has no aggregation.

If you need a pivot table like the one shown here, create it as an SQL-based KPI.

### 3.6 Using List-based Levels Carefully

In DeepSee, unlike many other BI tools, you can base a level upon a list value. Such levels are useful, but it is important to understand their behavior.

For example, a patient can have multiple allergies. Each allergy has an allergen and a severity. Suppose that the base table is Patients and the model includes the Allergies and Allergy Severity levels. We could create a pivot table that looks like this:

<table>
<thead>
<tr>
<th>Allergies</th>
<th>Nil known allergies</th>
<th>Minor</th>
<th>Moderate</th>
</tr>
</thead>
<tbody>
<tr>
<td>additive/coloring agent</td>
<td></td>
<td>137</td>
<td>121</td>
</tr>
<tr>
<td>animal dander</td>
<td></td>
<td>132</td>
<td>136</td>
</tr>
<tr>
<td>ant bites</td>
<td></td>
<td>126</td>
<td>131</td>
</tr>
<tr>
<td>bee stings</td>
<td></td>
<td>124</td>
<td>141</td>
</tr>
<tr>
<td>dairy products</td>
<td></td>
<td>128</td>
<td>114</td>
</tr>
<tr>
<td>dust mites</td>
<td></td>
<td>126</td>
<td>103</td>
</tr>
<tr>
<td>eggs</td>
<td></td>
<td>132</td>
<td>125</td>
</tr>
<tr>
<td>fish</td>
<td></td>
<td>137</td>
<td>135</td>
</tr>
<tr>
<td>mold</td>
<td></td>
<td>116</td>
<td>136</td>
</tr>
<tr>
<td>nil known allergies</td>
<td>1,578</td>
<td></td>
<td></td>
</tr>
<tr>
<td>peanuts</td>
<td></td>
<td>129</td>
<td>114</td>
</tr>
<tr>
<td>pollen</td>
<td></td>
<td>131</td>
<td>135</td>
</tr>
</tbody>
</table>

Upon first seeing this pivot table, the user might think that this pivot table shows correlations between different sets of patient allergies. It does not.

This pivot table, as with all other pivot tables in this cube, shows sets of patients. For example, the ant bites row represents patients who have an allergy to ant bites. The Minor column represents patients who have at least one allergy that is marked
as minor. There are 129 patients who have an allergy to ant bites and at least one allergy that is marked as minor. This does not mean that there are 129 patients with minor allergies to ant bites.

It is possible to create a pivot table that does show correlations between different sets of patient allergies. To do so, however, you would have to define a model based on the patient allergy, rather than the patient.

When you use a list-based level in a filter, the results require careful thought. Consider the following pivot table:

<table>
<thead>
<tr>
<th>Allergies</th>
<th>Patient Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Data Available</td>
<td>3,943</td>
</tr>
<tr>
<td>additive/coloring agent</td>
<td>413</td>
</tr>
<tr>
<td>animal dander</td>
<td>418</td>
</tr>
<tr>
<td>ant bites</td>
<td>436</td>
</tr>
<tr>
<td>bee stings</td>
<td>446</td>
</tr>
<tr>
<td>dairy products</td>
<td>445</td>
</tr>
<tr>
<td>dust mites</td>
<td>416</td>
</tr>
<tr>
<td>eggs</td>
<td>389</td>
</tr>
<tr>
<td>fish</td>
<td>419</td>
</tr>
<tr>
<td>mold</td>
<td>397</td>
</tr>
<tr>
<td>nil known allergies</td>
<td>1,427</td>
</tr>
<tr>
<td>peanuts</td>
<td>444</td>
</tr>
<tr>
<td>pollen</td>
<td>426</td>
</tr>
<tr>
<td>shellfish</td>
<td>442</td>
</tr>
<tr>
<td>soy</td>
<td>449</td>
</tr>
<tr>
<td>tree nuts</td>
<td>453</td>
</tr>
<tr>
<td>wheat</td>
<td>435</td>
</tr>
</tbody>
</table>

This pivot table shows patients, grouped by allergy. Now suppose that we apply a filter to this pivot table, and the filter selects the fish member:

<table>
<thead>
<tr>
<th>Allergies</th>
<th>Patient Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>additive/coloring agent</td>
<td>13</td>
</tr>
<tr>
<td>animal dander</td>
<td>19</td>
</tr>
<tr>
<td>ant bites</td>
<td>16</td>
</tr>
<tr>
<td>bee stings</td>
<td>23</td>
</tr>
<tr>
<td>dairy products</td>
<td>24</td>
</tr>
<tr>
<td>dust mites</td>
<td>22</td>
</tr>
<tr>
<td>eggs</td>
<td>12</td>
</tr>
<tr>
<td>fish</td>
<td>419</td>
</tr>
<tr>
<td>mold</td>
<td>13</td>
</tr>
<tr>
<td>peanuts</td>
<td>22</td>
</tr>
<tr>
<td>pollen</td>
<td>22</td>
</tr>
<tr>
<td>shellfish</td>
<td>23</td>
</tr>
<tr>
<td>soy</td>
<td>20</td>
</tr>
<tr>
<td>tree nuts</td>
<td>24</td>
</tr>
<tr>
<td>wheat</td>
<td>19</td>
</tr>
</tbody>
</table>
Now we are viewing only patients who have an allergy to fish. Note the following:

- The pivot table shows the fish member. For this member, the patient count is the same as in the previous pivot table.
- It also shows some other members of the Allergies level. For these members, the patient count is lower than in the previous pivot table.
- The pivot table does not include the No Data Available member of the Diagnoses level. Nor does it include the Nil Known Allergies member.

To understand these results, remember that we are viewing only patients who are allergic to fish, and this is not the same as “viewing only the fish allergy.” The patients who are allergic to fish are also allergic to other things. For example, 13 of the patients who are allergic to fish are also allergic to additives/coloring agents.

If we change the filter to select only the No Data Available member, we see this:

```
In this case, we are viewing only the patients who do not have any recorded allergy. By definition (because of how this level is defined), there is no overlap of these patients with the patients who have specific allergies.

Note: List-based levels cannot have properties and cannot have child or parent levels.
```

### 3.7 Handling Null Values Correctly

For any measure or level, the source value could potentially be null in some cases. It is important to understand how DeepSee handles nulls and to adjust your model according to your business needs and usability requirements.

#### 3.7.1 Null Values in a Measure

For a measure, if the source value is missing for a given record, DeepSee does not write any value into the measure column of the fact table. Also, DeepSee ignores that record when aggregating measure values. In most scenarios, this is appropriate behavior. If it is not, you should use a source expression that detects null values and replaces them with a suitable value such as 0.

#### 3.7.2 Null Values in a Level

For a level, if the source value is missing for a given record in the base class, DeepSee automatically creates a member to contain the null values (with two exceptions). You specify the null replacement string to use as the member name; otherwise, the member is named **Null**.

The first exception is **levels that use range expressions**. The second is **computed dimensions**, which are discussed in the chapter “Using Advanced Features of Cubes and Subject Areas.” The replacement string has no effect in these cases.
3.8 Usability Considerations

It is also useful to consider how users see and use the model elements. This section explains how the Analyzer and pivot tables represent model elements and concludes with some suggestions for your models.

3.8.1 Dimensions, Hierarchies, and Levels

The Model Contents area of the Analyzer displays each dimension and the levels in it, but does not display the hierarchies (for reasons of space). For example:

```
  ▼BirthD
    ▼Decade
      ▼Year
        ▼Quarter Year
          ▼Period
            ▼Date
```

Therefore, a user working in the Analyzer does not necessarily know which levels are related via hierarchies.

If a level is used for rows, then the name of the level appears as the column title. For example:

```
Decade | Patient Count
-------|-------------
1910s  | 79
1920s  | 203
1930s  | 532
1940s  | 771
```

If a dimension is used for rows, the name of the dimension appears as the column title. Also, the system uses an MDX function that gets the All member for the dimension, as well as all members of the first level defined in that dimension:

```
  ▼Aged
    ▼All Patients
    ▼0 to 29
    ▼30 to 59
    ▼60+
```

3.8.2 All Members

The Model Contents area displays every All member. For example:

```
  ▼Dimensions
    ▼Aged
```

In the Analyzer, users can drag and drop the All member, in the same way they can drag any other member.
3.8.3 Considerations with Multiple Cubes

If you find that you need to define multiple cubes to analyze one area of your business, consider the following points:

- If you have a large number of cubes, relationships are often quite useful. Rather than defining the same dimension repeatedly in different cubes (with different definitions), you can define it in a single place. This is more convenient for development and lessens the amount of disk space that is needed.
  
  The slight disadvantage of relationships is that if you rebuild the independent cube, you must also rebuild all the dependent cubes in the appropriate order, as discussed in the chapter “Defining Cube-Cube Relationships.”

- If you define relationships or shared dimensions, define a utility method or routine that compiles the cubes in the appropriate order.
  
  It is easier to maintain such a method as you add cubes than it is to manually rebuild them in the correct order. Building them in the wrong order can cause problems that are difficult to troubleshoot.

- If you need pivot tables that display measures from multiple cubes, you must define shared dimensions and a compound cube. A compound cube is the only way to use measures together that belong to different cubes.

3.8.4 Recommendations

The following recommendations may also be useful to you, depending on your business needs:

- Decide whether you will use dimensions directly in pivot tables. If so, assign user-friendly names to them (at least for display names).
  
  If not, keep their names short and omit spaces, to enable you to write MDX queries and expressions more easily.

  Also, use a different name for the dimension than for any level in the dimension, in order to keep the syntax clear, if you need to view or create MDX queries or expressions.

- Hierarchy names are not visible in the Analyzer or in pivot tables. If you use short names (such as H1), your MDX queries and expressions are shorter and easier to read.

- Define only one hierarchy in any dimension; this convention gives the users an easy way to know if the levels in a dimension are associated with each other.

- Define an All member in only one dimension. Give this All member a suitably generic name, such as All Patients. Depending on how you intend to use the All member, another suitable name for it might be Total or Aggregate Value.

- Use user-friendly names for levels, which are visible in pivot tables.
• You can have multiple levels with the same name in different hierarchies. Pivot tables and filters, however, show only the level name, so it is best to use unique level names.

• Specify a value for the **Field name in fact table** option for each applicable level and measure; this option does not apply to time levels and iKnow measures. Take care to use unique names.

  Troubleshooting is much easier when you can identify the field in which a given level or measure is stored, within the fact table.

  For information, see “Details for the Fact and Dimension Tables.”
4

Defining Cubes

This chapter describes the basics of defining cubes. It includes the following topics:

• How to define a new, minimal cube
• Possible source classes for a cube
• How to specify the other cube options
• How to add items to a cube
• Rules for names of items in a cube
• Other common options for items in a cube
• How to compile and build a cube
• How to display a cube in the Analyzer
• How to delete a cube

Important: DeepSee uses SQL to access data while building the cube, and also when executing detail listings. If your model refers to any class properties that are SQL reserved words, you must enable support for delimited identifiers so that DeepSee can escape the property names. For a list of reserved words, see “Reserved Words” in the Caché SQL Reference. For information on enabling support for delimited identifiers, see “Identifiers” in Using Caché SQL.

4.1 Defining a Cube

To define a cube:

1. In the Architect, click New.

   The system displays a dialog box where you can enter details for the new cube.

2. Click Cube.

3. Enter the following information at a minimum:
   
   • **Cube Name** — Logical name of the cube to use in queries.
   • **Class Name for the Cube** — Complete package and class name for the cube class.
• **Source Class** — Complete package and class name of the base class for this cube. See the subsection “Possible Source Classes.”

You can either type the class name or click **Browse** and select the class.

The other options are discussed later in this chapter.

Apart from the class name of the cube class, you can edit all cube options after creating the cube.

4. Click **OK**.

5. Optionally save the cube. To do so:
   a. Click **Save**.
   b. Click **OK**.

The system creates the class.

Another option is to use a utility to generate the cube class, as discussed in the following subsection.

Or manually create the class as described in the appendix “Reference Information for Cube Classes.”

### 4.1.1 Generating the Cube Class

The class `%DeepSee.WizardUtils` provides a method that you can use to generate a cube class. The method is as follows:

```lisp
classmethod %GenerateCubeDefinition(pSourceClass As %Library.String(MAXLEN="")="", pCubeName As %Library.String(MAXLEN="")="", pCubeClass As %Library.String(MAXLEN="")="", pAutoDelete As %Library.Integer = 0)
```

Where:

- **pSourceClass** is the full name of the source class for the cube.
- **pCubeName** is the logical name of the cube.
- **pCubeClass** is the full name of the cube class.
- **pAutoDelete** controls whether the cube class is deleted, if it already exists. If this argument is nonzero, the class is deleted; otherwise it is not.

This method generates a cube definition as follows:

- It has one measure for each numeric property in the source class.
- It has one date dimension for each date property in the source class. This dimension contains one hierarchy with three levels. The levels are year, year and month, and date.
- It has one data dimension for each other property in the source class. This dimension contains one hierarchy with one level.
- It has one listing that uses all properties in the source class.

The method ignores transient and multidimensional properties.

### 4.1.2 Changing the Base Class for a Cube

On rare occasions, you might need to change the base class for a cube. To do so, you can do either of the following:

- Edit the **Source Class** option for the cube in the Details Area.
Click the Change link next to Source Values at the top of the Class Viewer.

If you do so, the system displays a dialog box where you can choose a source class; this is the same dialog box as is shown in “Possible Source Classes.”

After you do this, be sure to modify the source property or source expression appropriately for all parts of the model.

### 4.2 Possible Source Classes for a Cube

If you click Browse next to Source Class, the system displays a dialog box like the following:

![Dialog box showing possible source classes](image)

Here you can select any class that can be used as the source of a cube in this namespace. There are three types of classes you can use this way:

- **Persistent classes** — Classes that extend %Library.Persistent.
- **Data connector classes** — Classes that extend %DeepSee.DataConnector. A data connector maps the results of an arbitrary SQL query into an object that can be used as the source of a cube. Typically, a data connector accesses external non-Caché data, but you can also use it to specify an SQL query against Caché, including an SQL query on a view.

For a cube that is based on a data connector, all listings in that cube must also be based on data connectors. Also, all these data connectors must have the same property marked as idkey="true", because the underlying mechanism uses the same ID values in all cases.


- **Collection classes**.
4.3 Other Cube Options

You can specify the following options for a cube:

- **Cube name** — Logical name of the cube to use in queries.
- **Display name** — Localizable name of the cube. If you do not specify this, the user interface instead displays the logical name.
- **Description** — (Optional) Comments to add to the cube class definition. Each line is saved as a separate comment line at the start of the class definition.
- **Caption** — (Optional) Specify the caption to display in the Analyzer and other utilities when working with this cube.
- **Domain** — (Optional) Specify the name of the domain to contain the localized strings of this cube. You might find it convenient to use a single domain for all your cubes; in other cases, it might be appropriate to have a separate domain for each cube. See the chapter “Performing Localization” in the *DeepSee Implementation Guide*.
- **Source class** — Complete package and class name of the base class for this cube.
- **Null replacement string** — (Optional) Specifies the string (for example, None) to use as the member name if the source data for a level is null.
  
  This option can overridden for levels, by a level option of the same name.
- **Default listing** — (Optional) Logical name of the default listing for this cube. This listing must be defined in the cube.
- **Resource** — (Optional) Specify the Caché resource that secures the cube.
- **Owner** — (Optional) Specify the owner of the cube. Specify a Caché user name.
- **Count measure name** — (Optional) Specify an alternative name for the Count measure. The default is %COUNT. It is useful to rename the Count measure if you create a compound cube; see the chapter “Defining Shared Dimensions and Compound Cubes.”
- **Count measure caption** — (Optional) Specify an alternative caption for the Count measure. The default is COUNT.
- **Initial build order** — (Optional) Specifies an optional ORDER BY clause for use when building the entire cube; does not affect incremental updates. Specify a comma-separated list of fields in the source table. You can use the SQL keywords ASC and DESC. For example: Age DESC, Gender
  
  For the implications of this option, see the section “Controlling the Fact Order” in the *DeepSee Implementation Guide*.
- **Build restriction** — (Optional) Specifies an optional WHERE clause to use when building or updating the cube. Specify an SQL comparison expression that uses fields in the source table. For example: Gender='F'
- **Depends On** — (Optional) Specifies the class or classes that must be runnable before this class can be compiled. This option controls how the Architect sets the DependsOn compiler keyword.
  
  By default, this option is blank, and DeepSee automatically sets the DependsOn keyword equal to the name of the source class for the cube. In some cases (for example with cube relationships), you might need to specify an additional class.
  
  If you need to specify this option, specify a comma-separated list of classes and specify the full package and class name for each class in the list. Your list should include the source class for the cube.
  
  For information on relationships between cubes, see the chapter “Defining Cube-Cube Relationships.”
4.4 Adding Items to a Cube

To add an item (such as a level or measure) to a cube, use the following general procedure:

1. For some types of items, first click an existing item in the Model Viewer, to indicate where the new item is to be added. The details are given in later sections of this book.

2. Click Add.

   The system displays a dialog box on which you can choose the item to add.

3. Click the item type and then click OK.

   The system adds the item, displays it in the Model Viewer, and shows its details in the Details Area.

4. Edit the details in the Details Area.

The following table indicates where the different types of items are discussed:

<table>
<thead>
<tr>
<th>For information on ...</th>
<th>See the chapter ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Dimension</td>
<td>“Defining Dimensions, Hierarchies, and Levels”</td>
</tr>
<tr>
<td>Time Dimension</td>
<td></td>
</tr>
<tr>
<td>Age Dimension</td>
<td></td>
</tr>
<tr>
<td>Hierarchy</td>
<td></td>
</tr>
<tr>
<td>Level</td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>“Defining Properties”</td>
</tr>
<tr>
<td>Measure</td>
<td>“Defining Measures”</td>
</tr>
<tr>
<td>Listing</td>
<td>“Defining Listings”</td>
</tr>
<tr>
<td>Calculated Member and Named Set</td>
<td>“Defining Calculated Members and Named Sets”</td>
</tr>
</tbody>
</table>

4.5 Names for Model Elements

When you define a model element, you specify a logical name for it (the Name field). This name is used in MDX queries and also is the default display name for that element. This section discusses requirements and suggestions for these names.

The logical names must follow these rules:

- The first character must be either a letter (in the Latin-1 character set), a number, or the underscore character (_).
- The other characters must be either letters, numbers, spaces, or underscore characters.

   Note that if you use spaces in a name, you must enclose the name within square brackets when writing MDX queries.
- The name must not be an MDX reserved keyword. Reserved keywords are not case-sensitive in MDX.

The logical names must also follow these additional rules:

- Within a given Caché namespace, each cube name must be unique.
- Within a given cube, each dimension name must be unique.
• Within a given dimension, each hierarchy name must be unique.
• Within a given hierarchy, each level name must be unique.
• Within a given level, each property name must be unique.
• Within a given cube, each measure name must be unique.

4.6 Other Common Options

When you define a model element, you can also specify the following options for it:

• **Display name** — (Optional) Localized name of this element for use in user interfaces. If you do not specify this, the user interface instead displays the logical name.
• **Description** — (Optional) Description of this element.
• **Disabled** — (Optional) If you select this check box, the element is disabled (not seen by the compiler). When you recompile the cube, this element is ignored.

4.7 Compiling and Building a Cube

As you develop your cubes, you will probably recompile and rebuild them multiple times. Briefly:

• To compile a cube, click **Compile**. The system starts to compile the class and displays a dialog box that shows progress. If you have made changes that you have not yet saved, the system saves them. Then click **OK**.
• To build a cube, click **Build**. The system displays a dialog box. Click **Build**. DeepSee starts to build the cube and displays progress as it does so. Then click **OK**. The cube is then available for use in the Analyzer.

For more information, see the chapter “Compiling and Building Cubes,” later in this book.

4.8 Opening a Cube in the Analyzer

As you develop your cubes, you should periodically use the Analyzer and examine the results. To open a cube in the Analyzer:

1. Click **DeepSee**, click **Analyzer**, and then click **Go**.
   
   **Tip:** If the Analyzer is already open, just click the **Analyzer** link at the top of the page.
2. If the left area is not currently displaying the cube you are validating, click **Open** and choose the cube.

For specific tips on validating levels, see “Validating Your Levels,” later in this book. For general information on using the Analyzer, see *Using the DeepSee Analyzer*. 
4.9 Deleting a Cube

To delete a cube, do the following:

1. In the Terminal, execute the following command:

   ```
   do ##class(%DeepSee.Utils).%KillCube(cubeName)
   ```

   Where `cubeName` is the name of the cube to remove. This command removes the cube cache and indices.

2. Delete the cube class (and its generated classes and their data) in either of the following ways:
   - In the Terminal, execute the following command:
     ```
     do $system.OBJ.Delete(classname)
     ```

     Where `classname` is the full package and class name of the cube class. For example:

     ```
     do $system.OBJ.Delete("Mypackage.Myclass")
     ```

   - In Studio, right-click the cube class and select **Delete**.
This chapter describes how to compile and build cubes. It includes the following topics:

- When to recompile and rebuild
- How to compile a cube
- How to build a cube in the Architect
- How to build a cube in the Terminal
- How to minimize cube size during development
- How to use parallel processing during a cube build
- How to recognize build errors

Note: This chapter describes how to build a cube during model development; while you build a cube as described here, users cannot execute queries. (However, if a query is currently running, you can build the cube.) For information on keeping a deployed cube current while permitting queries to be executed, see the DeepSee Implementation Guide.

5.1 When to Recompile and Rebuild

If you make any change to a cube class or a subject area class, you must recompile that class before those changes take effect. For many changes to a cube, you must also rebuild the cube before those changes take effect.

The following table lists the required actions after changes:

<table>
<thead>
<tr>
<th>Element Type</th>
<th>Type of Change</th>
<th>Required Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cube (root element)</td>
<td>Edits to Name or Source class</td>
<td>Recompile and rebuild</td>
</tr>
<tr>
<td></td>
<td>Other changes that apply to the cube but not to specific elements in the cube</td>
<td>Recompile</td>
</tr>
<tr>
<td>Element Type</td>
<td>Type of Change</td>
<td>Required Actions</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>Measure</td>
<td>Edits to the following options of an existing measure (many other elements have some or all of these common options).</td>
<td>Recompile</td>
</tr>
<tr>
<td></td>
<td>- Disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Hidden</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Display name</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Description</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Format string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>All other changes, including adding and deleting measures</td>
<td>Recompile and rebuild</td>
</tr>
<tr>
<td>Dimension (not a computed dimension)</td>
<td>Edits to the following options of an existing dimension:</td>
<td>Recompile</td>
</tr>
<tr>
<td></td>
<td>- Common options as listed in “Measure”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Enable the All level for this dimension</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Caption for All member</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Display name for All member</td>
<td></td>
</tr>
<tr>
<td></td>
<td>All other changes, including adding and deleting dimensions</td>
<td>Recompile and rebuild</td>
</tr>
<tr>
<td>Computed dimension</td>
<td>All changes</td>
<td>Recompile</td>
</tr>
<tr>
<td>iKnow dimension</td>
<td>All changes</td>
<td>Recompile</td>
</tr>
<tr>
<td>Hierarchy</td>
<td>Edits to the common options of an existing hierarchy (as listed in “Measure”)</td>
<td>Recompile</td>
</tr>
<tr>
<td></td>
<td>All other changes, including adding and deleting hierarchies</td>
<td>Recompile and rebuild</td>
</tr>
<tr>
<td>Level</td>
<td>Edits to the following options of an existing level:</td>
<td>Recompile</td>
</tr>
<tr>
<td></td>
<td>- Common options as listed in “Measure”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Null replacement string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Sort option</td>
<td></td>
</tr>
<tr>
<td></td>
<td>All other changes, including adding and deleting levels</td>
<td>Recompile and rebuild</td>
</tr>
<tr>
<td>Property</td>
<td>Edits to the following options of an existing property:</td>
<td>Recompile</td>
</tr>
<tr>
<td></td>
<td>- Common options as listed in “Measure”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Sort members by property value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>All other changes, including adding and deleting properties</td>
<td>Recompile and rebuild</td>
</tr>
<tr>
<td>Listing</td>
<td>All changes</td>
<td>Recompile</td>
</tr>
<tr>
<td>Calculated member</td>
<td>All changes</td>
<td>Recompile</td>
</tr>
<tr>
<td>Named set</td>
<td>All changes</td>
<td>Recompile</td>
</tr>
</tbody>
</table>
### 5.2 Compiling a Cube

To compile a cube class in the Architect:

1. Click **Compile**.
   
   The system starts to compile the class and displays a dialog box that shows progress.
   
   If you have made changes that you have not yet saved, the system saves them before compiling the cube.

2. Click **OK**.

Or open the cube class in Studio and compile it in the same way that you compile other classes.

When you compile a cube class, the system automatically generates the fact table and all related classes if needed. If the fact table already exists, the system regenerates it only if it is necessary to make a structural change.

If there are any cached results for this cube, the system purges them.

### 5.3 Building a Cube

The phrase *building a cube* refers to two tasks: adding data to the fact table and other tables and building the indices used to access this data.

To build a cube in the Architect:

1. Click **Build**.
   
   The system displays a dialog box.

2. Click **Build**.
   
   DeepSee starts to build the cube and displays progress as it does so.

3. Click **OK**.

The cube is then available for use as described in *Using the DeepSee Analyzer*.
5.4 Building the Cube in the Terminal

To build the cube in the Terminal, execute the `%BuildCube()` class method of the `%DeepSee.Utils` class. This method has the following signature:

```plaintext
classmethod %BuildCube(pCubeName As %String,
    pAsync As %Boolean = 1,
    pVerbose As %Boolean = 1,
    pIndexOnly As %Boolean = 1,
    pMaxFacts As %Boolean = 0) as %Status
```

Where:

- `pCubeName` is the logical name of the cube as given in its XData block; this is not case-sensitive.
- `pAsync` controls whether the process is asynchronous. If this argument is true, the system uses background tasks.
- `pVerbose` controls whether the method writes status information. If this argument is 1, the system writes status updates to the command line. In all cases, this method also writes log files as described in the following subsection.
- `pIndexOnly` controls whether the method only updates the indices. If this argument is 1, the system only updates the indices of the fact table.
- `pMaxFacts` specifies the maximum number of rows in the fact table. This determines the number of rows of the base table that the system uses when building the cube.

While you are developing a cube, you typically recompile and rebuild it frequently. If you are using a large data set, you might want to limit the number of facts in the fact table, in order to force the cube to be rebuilt more quickly.

If `pMaxFacts` is 0, the default, all rows of the base table are processed.

For example:

```plaintext
do ##class(%DeepSee.Utils).%BuildCube("patients")
```

This method writes output that indicates the following information:

- Number of processors used.
- Total elapsed time taken by the build process.
- Total amount of time spent evaluating source expressions, summed across all processors.

For example:

```plaintext
Building cube [patients]
Existing cube deleted.
Fact table built: 1,000 fact(s) (2 core(s) used)
Fact indices built: 1,000 fact(s) (2 core(s) used)
Complete
Elapsed time: 1.791514s
Source expression time: 0.798949s
```

If `Source expression time` seems too high, you should re-examine your source expressions to be sure that they are as efficient as possible; in particular, if the expressions use SQL queries, double-check that you have the appropriate indices on the tables that the queries use.
5.5 Minimizing Cube Size During Development

While you are developing a cube, you typically recompile and rebuild it frequently. If you are using a large data set, you might want to limit the number of facts in the fact table, in order to force the cube to be rebuilt more quickly. To do this, do one of the following:

- Edit the cube class in Studio and add the `maxFacts` attribute to the `<cube>` element. See the appendix “Reference Information for Cube Classes.”
  
  If you do so, be sure to remove this attribute before deployment.
- Build the cube in the Terminal and specify the `pMaxFacts` argument. See “Building a Cube in the Terminal” earlier in this chapter.

5.6 Using Parallel Processing During a Cube Build

If all the following items are true, DeepSee uses multiple cores to perform the build:

- You build the cube asynchronously (which you can do from the Terminal).
- The source for a cube is a persistent class (rather than a data connector). Data connectors are described in the DeepSee Implementation Guide.
- The persistent class is bitmap-friendly.
- The Initial build order and Build restriction options of the cube have not been set. These options are described in “Specifying Cube Options,” earlier in this book.

5.6.1 Specifying the Agent Count

When you build a cube asynchronously, DeepSee sets up a pool of agents to do the work, if it is possible to use parallel processing. This pool consists of a set of agents with high priority and the same number of agents with low priority. By default, the total number of high priority agents (or low priority agents) is four times the number of cores detected on the machine where DeepSee is running.

**Note:** These agents are also used to execute queries.

To specify a different agent count, call the class method `%SetAgentCount()` in the class `%DeepSee.Utils`. The argument should be one of the following:

- The desired number of high priority agents — that is, half of the desired total agent count.
- Null — to restore the default behavior.

The method returns the current number of high priority agents — that is, half of the total agent count. Or it can return null, which means that the default behavior is in effect.

For example, the following command sets the total agent count to 10:

```d
#class(%DeepSee.Utils).%SetAgentCount(5)
```

To see the current number of agents, use the `%GetAgentCount()` method of the same class. This method returns the current number of high priority agents. Or it can return null, which means that the default behavior is in effect.
When the agents are running, they are visible in the Management Portal as separate processes. To see them, go to the page [System] > [System Operation] > [Processes] and look for instances of the %DeepSee.TaskMaster routine. They are not shut down automatically but are intended not to consume resources when idle.

On rare occasions, you might need to reset these agents. To do so, use the %Reset() method of %DeepSee.Utils. This method also clears any pending tasks and clears the result cache for the current namespace, which would have an immediate impact on any users. This method is intended for use only during development.

## 5.7 Build Errors

When you build a cube, it is important to check the fact count (as discussed in the first subsection), because this count is a useful indicator of any build problems.

In most cases, if errors occur while DeepSee builds a cube, DeepSee reports them to you, in the Architect or in the Terminal. It also records the same errors in a log file, as discussed in the second subsection. Note that in some cases, DeepSee does not report an error.

In the case of <STORE> errors, see the third subsection.

### 5.7.1 Checking the Fact Count

When you build a cube, pay attention to the number of facts that it builds and indexes. For example, in the Architect:

```plaintext
Status
Deleting facts 10,000 fact(s) 100% Complete
Building facts 10,000 fact(s) 100% Complete
Building indices 10,000 fact(s) 100% Complete
Complete
```

Or, in the Terminal:

```plaintext
SAMPLES>do ##class(%DeepSee.Utils).%BuildCube("patients")
Existing cube deleted.
Fact table built: 10,000 fact(s), 2 core(s) used.
Fact indices built: 10,000 fact(s), 2 core(s) used.
Complete
```

Each fact is a record in the fact table. The fact table should have the same as the number of records in the base table, except in the following cases:

- You limit the fact count as discussed earlier in this chapter.
- The cube class also defines the %OnProcessFact() callback, which you can use to exclude records from the cube.
  See the chapter “Using Advanced Features of Cubes and Subject Areas.”

If you see discrepancies of unknown cause, do the following:

1. Examine any levels that use range expressions, and verify that these levels do not drop records. See “Validating Your Levels,” later in this book.

   An error of this kind affects the index count but not the fact count.

2. Examine the log files (discussed next).

3. If the log files indicate a <STORE> error, see “<STORE> Errors.”
4. Try disabling selected dimensions or measures. Then recompile and rebuild to isolate the dimension or measure that is causing the problem.

Also, when DeepSee builds the indices, the fact count should equal the number of records in the fact table. For example, the Architect should show the same number for Building facts and for Building indices. If there is a discrepancy between these numbers, check the log files.

### 5.7.2 Log Files

After it builds the cube or tries to build the cube, DeepSee writes the following log files to the directory `install-dir/mgr`:

- **DeepSeeUpdate_cube_name_namespace.log**, where `cube_name` is the name of the cube, and `namespace` is the namespace in which this cube is defined. If errors occur during the build process, this file contains them.
  
  If there are no errors, the system does not write this file.

- **DeepSeeTasks_namespace.log**. This file contains information about the background agents that DeepSee used during the build process. For example:

  2009-11-04 16:53:35.648       2312 TaskMaster Background agents killed
  2009-11-04 16:53:35.663       2312 TaskMaster Create background agents..
  2009-11-04 16:53:35.739       6900 TaskMaster Agent started:1...
  ...
  2009-11-04 16:54:19.561       2312 TaskMaster Background agents killed

  To see this file from the Management Portal, go to the page `[System] > [DeepSee] > [DeepSee Logs].`

The time stamps in these files use `$NOW` to write the local date and time, ignoring daylight saving time.

### 5.7.3 <STORE> Errors

In some cases, DeepSee might report a `<STORE>` error like the following:

ERROR #5002: Cache error: `<STORE>%ConstructIndices+44^Cube.cube_name.Fact.1

This error can occur when a level has a very large number of members. By default, when DeepSee builds the indices, it uses local memory to store the indices in chunks and then write these to disk. If a level has a very large number of members, it is possible to run out of local memory, which causes the `<STORE>` errors.

To avoid such errors, try either of the following:

- Build the cube with a single process. To do so, use `%BuildCube()` in the Terminal, and use 0 for its second argument.

- In the `<cube>` element, specify `bitmapChunkInMemory="false"` (this is the default). When this cube is built using background processes, the system will use process-private globals instead of local variables (and will not be limited by local memory).
Defining Dimensions, Hierarchies, and Levels

This chapter describes how to define dimensions, hierarchies, and levels. It discusses the following topics:

- Overview
- How to create a new dimension, hierarchy, and level
- How to create a new hierarchy and level
- How to create a new level
- How to specify the source values for a dimension or level
- Details for source expressions
- How to ensure that member keys are unique
- How to override the default null replacement string for level
- How to define a level based on a list
- How to define a time level
- How to define a custom time level
- How to define an age level
- How to specify a range expression for a level
- How to configure a level to use the DISPLAYLIST values of a class property
- How to use property values as member names
- How to control the sort order for the members
- How to make member names localizable in the same way as other cube elements
- How to define dependencies between levels in different hierarchies
- How to change the order of the dimensions in a cube
- How to change the order of the levels in a hierarchy
- How to specify column names in the fact table
- How to validate a level
6.1 Overview

You can create model elements in either of two ways:

- By drag and drop. If you can drag a property from the Class Viewer to an appropriate target in the Model Viewer, the Architect creates a model element; as described in the table after this list. Then you can edit the definition in the Details Area, if needed.

  This technique makes it easy to define elements directly based on source properties. It is also useful as a starting point when you need to create elements based on source expressions.

- By using the Add Element link.

For reference, the following table describes the result of various drag-and-drop actions:

<table>
<thead>
<tr>
<th>If you drop the property XYZ here...</th>
<th>The Architect creates this...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measures heading or any existing measure</td>
<td>A measure named XYZ, XYZ1 or so on, as needed for uniqueness. The measure is based on the source property XYZ and is aggregated with SUM. See the chapter “Defining Measures.”</td>
</tr>
<tr>
<td>Dimensions heading</td>
<td>A new data dimension named XYZ, XYZ1, or so on, as needed for uniqueness. This dimension contains the hierarchy H1, which contains the level XYZ. The level is based on the source property XYZ.</td>
</tr>
<tr>
<td>Existing dimension</td>
<td>A new hierarchy (H2, for example), which contains the level XYZ. The level is based on the source property XYZ.</td>
</tr>
<tr>
<td>Existing hierarchy</td>
<td>A new level named XYZ, XYZ1, or so on, as needed for uniqueness within this hierarchy. The level is based on the source property XYZ.</td>
</tr>
<tr>
<td>Existing level</td>
<td>A new level property named XYZ, XYZ1, or so on, as needed for uniqueness within this level. The level property is based on the source property XYZ. See the chapter “Defining Properties.”</td>
</tr>
</tbody>
</table>

6.2 Creating a New Dimension, Hierarchy, and Level

To define a usable level, you must define the following, at a minimum:

- A dimension
- A hierarchy in that dimension
- A level in that hierarchy

You can use drag-and-drop actions as described in “Overview.” Or, you can do the following:

1. Add a dimension:
   a. Click Add Element.
      The system displays a dialog box.
   b. For Enter New Item Name, type a dimension name.
c. Click one of the following choices, depending on the type of dimension you want to create:

- **Data Dimension** — Click this for most dimensions.
- **Time Dimension** — Click this to create a dimension that groups data by a date or time value. For details, see “Defining a Time Level,” later in this chapter.
- **Age Dimension** — Click this to create a dimension that groups data by age, based on a date value. For details, see “Defining an Age Level,” later in this chapter.

For information on the **iKnow Dimension** option, see “Using Unstructured Data in Cubes,” later in this book.

d. Click **OK**.

The system creates the dimension and displays it in the **Model Viewer**. For example:

```
<table>
<thead>
<tr>
<th></th>
<th>data dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>hierarchy</td>
</tr>
<tr>
<td>New_Level1</td>
<td>level 1</td>
</tr>
</tbody>
</table>
```

Note that DeepSee has also created a hierarchy within this dimension and a level within that hierarchy.

2. Modify the automatically created level to use a more suitable name:

a. Click the level in the **Model Viewer**.

The system displays details in the **Details Area**.

b. Make the following changes:

- **Name** — Change this to the level name you want.
- **Display Name** — Specifies the localized display name for this level. Either clear this (so that the system uses the value given for **Name** instead) or specify a display name.

See also “Other Common Options,” earlier in this book.

3. Specify the source values for this level, as described in the section “Defining the Source Values for a Dimension or Level.”

4. Click **Save**.

5. When prompted, click **OK**.

**Note:** By default, the Analyzer does not display hierarchy names shown unless a dimension contains multiple hierarchies. Alternatively, a dimension can be defined so that its hierarchy names are always shown or never shown. See the reference for **showHierarchies** in “<dimension>” in the appendix “Reference Information for Cube Classes.”

### 6.3 Adding a Hierarchy and Level

To add a hierarchy and a level to an existing dimension, you can use drag-and-drop actions as described in “Overview.” Or, you can do the following:

1. Click the dimension in the **Model Viewer**.
2. Click Add Element.
   The system displays a dialog box.

3. For Enter New Item Name, type a hierarchy name.

4. Click Hierarchy.

5. Click OK.
   The system creates the hierarchy and displays it in the Model Viewer. It also creates one level within that hierarchy.

6. Optionally select the hierarchy in the Model Viewer and edit the details in the Details Area.

7. Select the level in the Model Viewer and edit the details in the Details Area.

### 6.4 Adding a Level

To add a level to an existing hierarchy, you can use drag-and-drop actions as described in “Overview.” Or, you can do the following:

1. Click either the hierarchy or an existing level within that hierarchy, in the Model Viewer.
   This action indicates where the level is to be added:
   • If you click the hierarchy, the new level will be added after all the other levels in this hierarchy.
   • If you click a level, the new level will be added immediately before that level.

2. Click Add Element.
   The system displays a dialog box.

3. For Enter New Item Name, type a level name.

4. Click Level.

5. Click OK.
   The system creates the level and displays it in the Model Viewer.

6. Optionally select the level in the Model Viewer and edit the details shown in the Details Area.

Alternatively, drag a class property from the Class Viewer and drop it onto a hierarchy. The system adds a level based on this property; this level is added after any other levels in this hierarchy.

**Important:** The order of the levels in a hierarchy determines the structure of the hierarchy, as noted in “Defining Hierarchies Appropriately,” earlier in this book. If you want to reorder the levels after defining them, see “Changing the Order of Levels in a Hierarchy.”

### 6.5 Defining the Source Values for a Dimension or Level

Each level must have a specified source value. You can specify the source values within the dimension or within the level. Typically:
For data dimensions, you specify the source values within the level.

For time and age dimensions, you specify the source values within the dimension.

For iKnow dimensions, see “Using Unstructured Data in Cubes,” later in this book. iKnow dimensions do not use the mechanism described here.

To specify a source value:

1. Select the dimension or the level in the Model Viewer.
2. In the Details Area, specify a value for one of the following:
   - **Property** — Specify the property name relative to the base class used by the cube; you can use dot syntax to refer to properties of properties. For example:
     
     Age
     
     For another example:
     
     HomeCity.PostalCode
     
     The property must have a projection to SQL.
     
     Also, you can refer to an object-valued property. When you do so, the numeric ID of the object is used as the source value.
     
     You cannot directly use a stream property (to use such a property, create an expression that returns the contents of the stream or a selected part of the contents).
   - **Expression** — Specify a Caché ObjectScript expression on which the level is based. For example:
     
     ```ObjectScript
     ##class(Cubes.StudyPatients).GetFavoriteColor(%source.PatientID)
     ```
     
     This expression is evaluated when the cube is built. For details, see the next section.

You can enter values into the **Property** and **Expression** fields in any of the following ways:

- By dragging a class property from the Class Viewer and dropping it into the field.
  
  The property that you drag and drop replaces any existing contents of the field.

- By clicking the Search button.
  
  - For **Property**, the system then displays a dialog box that shows the properties of this class. Click a property and then click OK.
  
  - For **Expression**, the system then displays a dialog box with a larger field you can type into. Type an expression and then click OK.

- By directly editing the value in the field.

The information on this section and in the next section also applies to level properties and to measures, which also require source values of some form.

### 6.5.1 Specifying a Source Value when Using a Data Connector

If the cube is based on a data connector, note the following restrictions:

- Specify the **Property** option but not the **Expression** option.

- You cannot use Caché dot syntax, because none of the properties of the data connector are object references.
For information on creating data connectors, see “Defining Data Connectors” in the DeepSee Implementation Guide.

### 6.6 Details for Source Expressions

As noted in the previous section, you can specify either a source value or a source expression to use as the basis of a dimension or level (or property or measure). You can create source expressions as follows:

- You can refer to a property in the source class. To do so, use the syntax `%source.PropertyName`, where `PropertyName` is the name of the property. When it builds the cube, DeepSee parses this expression and looks up the SqlFieldName of the given property.

- You can use dot syntax to refer to properties of properties.

- You can refer to an object-valued property. When you do so, the numeric ID of the object is used as the source value.

- You can use the variable `%cube` to refer to the cube class; this is useful if you have defined utility methods in the cube class that you want to use within source expressions.

- You can refer to the value of an `<expression>` element. To do so, use the following syntax:

  ```
  %expression.expressionName
  ```


- You can use the utility methods `ToUpper()`, `ToLower()`, and `Log()` as in the following example:

  ```
  ..ToUpper(%source.HomeCity.Name)
  ```

  DeepSee can use these methods in this way because the fact table class inherits from `%DeepSee.CubeFunctionSet`, which defines them. For details on these methods, see the class reference for that class.

- You can use the `%lookup()` method of `%DeepSee.CubeDefinition` to invoke a term list. See the subsection “Using a Term List.”

- You can use the `%rule()` method of `%DeepSee.CubeDefinition` to invoke an Ensemble business rule. See the subsection “Using an Ensemble Business Rule.”

**Note:** You cannot use a source expression if the cube is based on a data connector.

#### 6.6.1 Using 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. (See “Defining Term Lists” in the DeepSee Implementation Guide.)

You can invoke a term list within a source expression. To do so, use the `%lookup()` method of `%DeepSee.CubeDefinition`. This method has the following signature:

```
%Lookup(term_list_name, lookup_value, default, alternative_field)
```

Where the arguments are strings, and their values are as follows:

- `term_list_name` evaluates to the name of a term list.
- `lookup_value` evaluates to the string to look up in the term list.
- `default`, which is optional, evaluates to the value to return if `lookup_value` is not found in the term list.
- `alternative_field`, which is optional, is the name of the field to return. The default is "value".
This argument is not case-sensitive.

This function examines the given term list, finds the term whose "key" field equals the string given by lookup_value and then returns the value contained in the field identified by alternative_field.

All term lists have at least two fields: "key" and "value". You can add additional fields. For information, see “Defining Term Lists” in the DeepSee Implementation Guide.

Note: Because your cube definition class inherits from %DeepSee.CubeDefinition, which defines the %Lookup() method, you can use a source expression like the following:

%cube.%Lookup(term_list_name,lookup_value,default,alternative_field)

For example, suppose that you have the following term list, called LocalTeams:

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta</td>
<td>Braves</td>
</tr>
<tr>
<td>Boston</td>
<td>Red Sox</td>
</tr>
<tr>
<td>New York</td>
<td>Yankees</td>
</tr>
</tbody>
</table>

You could add a property to the City level in HoleFoods as follows:

<property name="Team" sourceExpression='%cube.%Lookup("LocalTeams",%source.Outlet.City,"No Team")'/>  

6.6.2 Using an Ensemble Business Rule

Business rules allow nontechnical users to change the behavior of Ensemble business processes. You can also use them in source expressions in DeepSee cubes. (For details on Ensemble business rules, see Using Business Rules with Ensemble.)

To access an Ensemble business rule within a source expression, use the %Rule() method of %DeepSee.CubeDefinition. This method has the following signature:

%Rule(rule_name)

Where rule_name is the name of an Ensemble business rule.

When this function is evaluated (for a given source record) during a cube build, the system passes to it an instance of the cube source class as the context object. The system uses this object, evaluates the rule, and then accesses the value returned by the rule.

Note: Because your cube definition class inherits from %DeepSee.CubeDefinition, which defines the %Rule() method, you can use a source expression like the following:

%cube.%Rule(rule_name)

6.7 Ensuring Uniqueness of Member Keys

If a level has a parent level, it is possible for member keys not to be unique. For details, see “Defining Member Keys and Names Appropriately,” in the previous chapter. InterSystems recommends that you ensure that member keys are unique for each level, so that you can easily access any member directly.
If it is possible, considering your source values and your hierarchies, to have duplicate member keys, do the following:

- Instead of using just the source value alone, concatenate the source value with some other value that is unique to the parent member. For example, suppose that you want to base the level on a source property called CityName, but you could have multiple cities with the same name in different countries. Instead of specifying Property, specify Expression. Use an expression like the following:

  %source.CountryName_%source.CityName

- If you do not want to use these concatenated strings as member names, add a property to the level and use its values as the member names. See “Using Property Values as the Member Names,” later in this chapter.

### 6.8 Specifying the Null Replacement String for a Level

For a level, the **Null replacement string** option specifies a string to use in place of any null values. This string takes precedence over the null replacement string that you specify for the cube.

### 6.9 Defining a List-based Level

You can define a level that is based on a list, an array, or a Caché class relationship. Each distinct list item is indexed as a separate member of the level. (For such levels, be sure to read “Using List-based Levels,” earlier in this book.)

DeepSee can directly use a source value that has the type %List, that is in the format returned by the $List function, or that is a character-delimited list. For other formats, you must convert the source value and then use it.

**Note:** A list-based level must be contained in its own hierarchy; there cannot be another level in this hierarchy.

#### 6.9.1 Defining a Level When the Source Value Is in a Standard Format

If the source value is available in $List format, in %List format, or in a character-delimited list, define the level as described earlier, with the following additional steps:

1. For **Source value list type**, select one of the following:
   - **$List structure** — Use this if the source value is in the format returned by the $List function or has the type %List.
   - **Comma delimited** — Use this if the source value is a comma-separated list.
   - **Other delimited** — Use this if the source value is a character-delimited list that uses some other character as the delimiter.

2. If you selected Other delimited, then for **List delimiter**, specify the character used to delimit the list.

#### 6.9.2 Defining a Level When the Source Value Is in Another Format

If the source value is not available in $List format or as a character-delimited list, use the following approach:

1. Create a utility method that converts it into one of the required formats. For example:
ClassMethod GetAllergies(ID As %Numeric) As %List
{
    Set allergies=#class(DeepSee.Study.Patient).%OpenId(ID,0).Allergies
    If (allergies.Count()==0) {Quit $LISTFROMSTRING("No Data Available")}
    Set list=""
    For i=1:1:allergies.Count() {
        Set $LI(list,i)=allergies.GetAt(i).Allergen.Description
    }
    Quit list
}

Notice the following points:

- The %List class is equivalent to the $List format.
- The second argument to %OpenId() specifies concurrency locking. If this argument is zero, you are opening the object without checking for other users of the object, for speed. Because you are only reading a value from the object, this technique is safe even in a multi-user environment.
- If there are no allergies, this method returns the string No Data Available, which becomes a member of the level, in exactly the same way as other returned strings.

Tip: You can place this utility method within the cube class. Also, you can easily test the method within the Terminal.

2. Within the Architect, when defining the level, do the following:
   a. For Expression, invoke the utility method and pass %source.%ID to it as an argument. For example:
      ##class(Cubes.StudyPatients).GetAllergies(%source.%ID)
   b. Select List.
   c. If the source value is a character-separated list, then for List delimiter, specify the character used to delimit the list. If this is null, DeepSee assumes that the source value is in $List format.

### 6.10 Defining a Time Level

A time level groups the data according to a date/time value in the data (such as a birth date, order date, response date, and so on). This section shows an example and describes how to create this type of level.

#### 6.10.1 Introduction and Example

A time level is a level defined within a time dimension. For example:

![Time Level Diagram]

Furthermore (not shown in the preceding picture), these items are defined as follows:
The BirthD dimension is based on the BirthDate property in the source class. This property contains the patient’s birth date.

Each level in this dimension uses an option called **Time function** to extract a specific part of the birth date. For example, the **Year** level extracts the year part of the date.

As a result:

- The **Decade** level has members such as 2010s, 2000s, 1990s, and so on.
- The **Year** level has members such as 2009, 2008, and so on.
- The **Quarter Year** level has members such as Q1 2009, Q2 2009, and so on.
- The **Period** level has members such as 2009–01, 2009–02, and so on.
- The **Date** level has members such as Jan 1 2009, Jan 2 2009, and so on.

The order of levels within the hierarchy H1 establishes that the **Decade** level is the parent of the **Year** level. For example, the member 1990s is the parent of the members 1990, 1991, 1992, and so on.

Similarly, the **Year** level is the parent of the **Quarter Year** level. For example, the member 2009 is the parent of the members Q1 2009, Q2 2009, and so on.

With time levels, you can use a special member called **NOW**, which uses the current date (runtime) and accesses the appropriate member of the level.

### 6.10.2 Defining a Time Level

To define a time level:

1. Define a dimension as described earlier, with two changes:
   - For **Type**, choose **time**.
   - Specify either **Property** or **Expression**.
     - If you specify **Property**, the ClientDataType of the property type class must be one of the following: DATE, FDATE, FTIMESTAMP, MVDATE, or TIMESTAMP. This means that the system can use any property of the following types: %Date, %MV.Date, %Time, %TimeStamp, %FilemanDate, %FilemanTimeStamp, a subtype of one of these, or your own data type that has one of the preceding ClientDataType values.
     - If you specify **Expression**, the source expression must return data in **$Horolog** format.

   When you define the dimension, the system creates the dimension and a hierarchy.

2. Define a level within this hierarchy.

   In this case, you will see different options than you saw for a data dimension.

3. For **Extract value with function**, choose a function that extracts the desired part of the date/time value. Use one of the following values:

<table>
<thead>
<tr>
<th>Time function</th>
<th>How Value Is Stored</th>
<th>How Value Is Displayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinuteNumber</td>
<td>60958,5083</td>
<td>01:24</td>
</tr>
<tr>
<td>HourNumber</td>
<td>60958,5083</td>
<td>1am</td>
</tr>
<tr>
<td>DayMonthYear</td>
<td>60958</td>
<td>Nov 24 2007</td>
</tr>
<tr>
<td>DayNumber</td>
<td>24</td>
<td>24</td>
</tr>
</tbody>
</table>
For the WeekYear and WeekNumber functions, DeepSee follows the ISO 8601 standard, which assigns a week number (from 1 to 53) for every date within a year. Because 7 is not a factor of 365 or 366, some years have 53 weeks. The first week of a year is defined as the week that contains the first Thursday of the year. That means for some years, January 1 is in the last week of the prior year. Similarly, December 31 may be in the first week of the next year.

6.10.3 Time Levels and Hierarchies

As noted earlier, the order of levels in a hierarchy affects how the members of the levels are created. For any two adjacent levels in the same hierarchy, the first level (level A) becomes the parent of the second level (level B). The hierarchy is a parent-child hierarchy, which means the following:

- Any member of level A is the parent of one or more members of level B.
- Any member of level B is the child of exactly one member of level A.
- Any record that belongs to a given member of level B must always belong to the same member of level A.

Therefore, a level based on Year cannot be the parent of a level based on Quarter, for example. Consider two patients, one born in Q3 2007 and one born in Q3 1982. These two patients both belong to the same member (Q3) of the Quarter level but belong to different members (2007 and 1982, respectively) of the Year level.

The following table lists time levels and their typical child levels. For reference, it also shows example members of each level:

<table>
<thead>
<tr>
<th>Time function</th>
<th>How Value Is Stored</th>
<th>How Value Is Displayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>WeekYear</td>
<td>2007W47</td>
<td>2007W47</td>
</tr>
<tr>
<td>WeekNumber</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>MonthNumber</td>
<td>11</td>
<td>November</td>
</tr>
<tr>
<td>MonthYear</td>
<td>200711</td>
<td>November 2007</td>
</tr>
<tr>
<td>QuarterNumber</td>
<td>4</td>
<td>Q4</td>
</tr>
<tr>
<td>QuarterYear</td>
<td>20074</td>
<td>Q4 2007</td>
</tr>
<tr>
<td>Year</td>
<td>2007</td>
<td>2007</td>
</tr>
<tr>
<td>Decade</td>
<td>1950</td>
<td>1950s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A Level Based on the Time Function...</th>
<th>Has Members Like This...</th>
<th>Child Level Is Typically Based on the Time Function...</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decade</td>
<td>1950s</td>
<td>Year, QuarterYear, MonthYear, or DayMonthYear</td>
<td>These levels use all parts of the date</td>
</tr>
<tr>
<td>Year</td>
<td>2007</td>
<td>QuarterYear, MonthYear, or DayMonthYear</td>
<td></td>
</tr>
<tr>
<td>QuarterYear</td>
<td>Q4 2007</td>
<td>MonthYear or DayMonthYear</td>
<td></td>
</tr>
<tr>
<td>MonthYear</td>
<td>November 2007</td>
<td>DayMonthYear</td>
<td></td>
</tr>
<tr>
<td>WeekYear</td>
<td>2007W47</td>
<td>WeekNumber</td>
<td></td>
</tr>
<tr>
<td>DayMonthYear</td>
<td>Nov 24 2007</td>
<td>No typical child level</td>
<td></td>
</tr>
<tr>
<td>A Level Based on the Time Function...</td>
<td>Has Members Like This...</td>
<td>Child Level Is Typically Based on the Time Function...</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>--------------------------</td>
<td>------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>QuarterNumber</td>
<td>Q4</td>
<td>MonthNumber</td>
<td>These levels are independent of the year</td>
</tr>
<tr>
<td>MonthNumber</td>
<td>November</td>
<td>No typical child level</td>
<td></td>
</tr>
<tr>
<td>WeekNumber</td>
<td>47</td>
<td>No typical child level</td>
<td></td>
</tr>
<tr>
<td>DayNumber</td>
<td>24</td>
<td>No typical child level</td>
<td>This level is independent of the year and independent of the part of the year</td>
</tr>
<tr>
<td>HourNumber</td>
<td>1am</td>
<td>MinuteNumber</td>
<td>These levels are independent of the day</td>
</tr>
<tr>
<td>MinuteNumber</td>
<td>01:24</td>
<td>No typical child level</td>
<td></td>
</tr>
</tbody>
</table>

### 6.10.4 Handling a Calendar That Has a Date Offset

In some cases, you may need a time level to match a financial calendar that includes a date offset. For example, in many companies, the financial year starts 1 Oct. Consider the following pivot table:

```
<table>
<thead>
<tr>
<th>FY 2002</th>
<th>Oct-2001</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>FY 2002-Q1</td>
<td>Nov-2001</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Dec-2001</td>
<td>7</td>
</tr>
<tr>
<td>FY 2002-Q2</td>
<td>Jan-2002</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Feb-2002</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Mar-2002</td>
<td>10</td>
</tr>
<tr>
<td>FY 2002-Q3</td>
<td>Apr-2002</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>May-2002</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Jun-2002</td>
<td>12</td>
</tr>
<tr>
<td>FY 2002-Q4</td>
<td>Jul-2002</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>Aug-2002</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Sep-2002</td>
<td>15</td>
</tr>
</tbody>
</table>
```

In the innermost grouping, this pivot table groups the records by the actual period (year and month, combined). For example, there are 14 records associated with the actual period October 2001. The periods are grouped into fiscal quarters, and the fiscal quarters are grouped into fiscal years. Note that the quarter FY 2002–Q1 includes October, November, and December of 2001.

For such a time level, edit the cube class in Studio and specify the `timeOffset` and `timeFormat` properties of the `<level>` element. For the example shown here:

- For the first two levels (the levels with members such as FY 2002 and FY 2002–Q1), `timeOffset` is "-3m" (which subtracts three months from the date values used by these levels).
- For the third level (the level with members such as Oct-2001), `timeOffset` is not specified, and thus this level uses the actual date values.

For information, see “Specifying a Date Offset” and “Specifying a Member Name Format,” in the appendix “Reference Information for Cube Classes.”
6.11 Defining Custom Time Levels

You can define custom levels that use time in other ways. To do so:

- Create a dimension whose **Type** is **data** (not **time**).
- For each level, specify a value for **Expression** that returns the desired value.
- See “Specifying the Sort Order for the Members,” later in this chapter.

By default, the members are sort alphabetically, but you can change the sort order.

For example, the Patients cube contains the **BirthWeekdayD** dimension, which is disabled by default. For this dimension, **Expression** is as follows:

```sql
$system.SQL.DAYNAME(%source.BirthDate)
```

This method executes the **DAYNAME()** method in the %SYSTEM.SQL class. That class provides a large set of methods that handle date values.

If you enable this dimension and rebuild the cube, you can create pivot tables like this:

<table>
<thead>
<tr>
<th>Weekday</th>
<th>Patient Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunday</td>
<td>147</td>
</tr>
<tr>
<td>Monday</td>
<td>140</td>
</tr>
<tr>
<td>Tuesday</td>
<td>141</td>
</tr>
<tr>
<td>Wednesday</td>
<td>141</td>
</tr>
<tr>
<td>Thursday</td>
<td>140</td>
</tr>
<tr>
<td>Friday</td>
<td>150</td>
</tr>
<tr>
<td>Saturday</td>
<td>141</td>
</tr>
</tbody>
</table>

This sample dimension defines only a single level. With the appropriate logic, you can define a hierarchy with weeks, months, and quarters in a 4–4–5 calendar, for example.

6.12 Defining an Age Level

An age level groups the records according to an age value in the data, relative to the current date. For example, it could group patients by age relative to today (that is, the date when the pivot table or query is run).

(Note that the Patients sample does not demonstrate this type of level; the **Age**, **Age Bucket**, and **Age Group** levels group patients in the fictitious study according to their ages at the date of the study; this is common practice with retrospective studies.)

To define an age level:

1. Define a dimension as described earlier, with two changes:
   - For **Type**, choose **age**.
   - Specify either **Property** or **Expression**.
**Important:** If you specify **Property**, the property must be in **$Horolog** format or must be of type `%TimeStamp` (or a subclass).

If you specify **Expression**, the source expression must return data in **$Horolog** format.

When you define the dimension, the system creates the dimension and a hierarchy.

2. Define a level within this hierarchy.

   In this case, you will see different options than you saw for a data dimension.

3. For **Extract value with function**, choose a function that extracts the desired part of the time value. Use one of the following values:
   - "Days" – Use this to determine the age in days.
   - "Months" – Use this to determine the age in months.
   - "Years" – Use this to determine the age in years.

### 6.13 Specifying a Range Expression

For any level in a data dimension, you can use the **Range expression** option. This option lets you use new values in the place of the actual source values. For numeric data, this transformation can replace source values with discrete bins. For any data, this transformation can specify replacement strings.

In either case, the new values become the names of the members of the level. The values also become the keys for the members.

**Note:** The null replacement option does not have any effect on levels that use a range expression.

To specify a range expression:

1. Click the Find button next to the **Range expression** field.

   The Architect displays a page like the following:

   ![Range Expression Page](image)

   - **Cube name:** HoleFoods
   - **Level name:** Discount Type

   **Enter a set of replacement values.**

   **Form of original values**
   - (Circle selected) **Numeric ranges (possibly open-ended)**
   - **Strings**

   **From** _(_
   **To** _(_
   **Replacement Value (Required)** _(_

   - **Add Replacement**
   - **Clear Changes**

2. For **Form of original values**, choose either **Numeric** or **Strings**.
This choice affects the form of this page.

3. Add a series of replacements as follows:
   - To add a row to the end of the table, click **Add Replacement**.
     Then enter details as given in the following subsections.
   - To remove a row, click the Remove button for that row.

   The order of the rows determines the default sort order of the members of this level.

4. Click **OK** to close this page.

   The **Range expression** field now shows something like this (for numeric bins):
   
   
   
   

   Or this (for string data):

   

   

   To make member names localizable, see “**Manually Specifying the Members of a Level,**” later in this book.

### 6.13.1 Defining Numeric Bins

When you define numeric bins, the Range Expression editor looks like this:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Replacement Value (Required)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>(</td>
<td>0.2</td>
<td>1-19%</td>
</tr>
<tr>
<td>[</td>
<td>0.2</td>
<td>20-49%</td>
</tr>
<tr>
<td>[</td>
<td>0.5</td>
<td>50%+</td>
</tr>
</tbody>
</table>

Each row in this table defines a numeric bin as follows:

- The button at the left end of the line indicates the form of the lower limit — inclusive or exclusive. When there is no lower limit, this option has no effect.

  If the lower limit is *inclusive*, that means that exact value of **From** is included in the range. If the lower limit is *exclusive*, that means that **From** is not in the range.

- **From** indicates the lower limit of the bin, if any.

- **To** indicates the upper limit of the bin, if any.

- The button at the right end of the line indicates the form of the upper limit — inclusive or exclusive. When there is no upper limit, this option has no effect.

- **Replacement Value** specifies the string to use as a member name for this level. Any record whose source value falls within the defined range is assigned to this member.
This value is also the key for the member.

For this level, DeepSee ignores records whose source values do not fall into any of the given bins.

### 6.13.2 Defining String Replacements

When you define string replacements, the Range Expression editor looks like this:

![Range Expression editor](image)

Each row in this table defines a string replacement as follows:

- **Original Value** specifies a possible value of the source property or source expression of this level.
- **Replacement Value** specifies the string to use as a member name for this level. Any record whose source value matches the given **Original Value** is assigned to this member.

This value is also the key for the member.

For this level, DeepSee ignores records whose source values do not match any of the given original values.

### 6.14 Configuring a Level to Use Display Values

A Caché class property can have both a stored value and a displayed value, via the `VALUELIST` and `DISPLAYLIST` property parameters. For example:

```
Property Gender As %String(VALUELIST = ",F,M", DISPLAYLIST = ",Female,Male");
```

When you use such a property as a level, by default, the system uses the value given in the `VALUELIST` parameter.

In the level definition, if you select the **Use value in DISPLAYLIST** option, the system uses the value given in the `DISPLAYLIST` parameter instead. For example:

<table>
<thead>
<tr>
<th>Gender</th>
<th>Patient Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Female</td>
<td>5,042</td>
</tr>
<tr>
<td>Male</td>
<td>4,958</td>
</tr>
</tbody>
</table>

### 6.15 Using Property Values as the Member Names

By default, the source values for a level become the names for the members of the level.

You can instead specify the member names by defining a property for this level and using the values of that property as the member names. This is useful in the following scenarios:
To enable member names to be accessed at runtime — Values of properties can be accessed at runtime (the same is not true for levels).

To provide user-friendly member names for members — In some cases, you must base a level on a unique value that is not user-friendly. For example, suppose that you have a level based on the patient’s primary care physician. People’s name are not reliably unique, so you would have to base the level on a unique doctor identifier instead, which might have no meaning for the users. In this case, you would also define a property that accesses the doctor’s name, and you would use that property as the name of the member.

To use property values as the names for members of a level:

1. Define a property for this level, as described in the next chapter.
   Each member of the level has a value for this property.
2. For this property, select the **Use as member names** option.
3. Optionally, to retrieve the property value at runtime, select the **Get value at run time** option.

   If you use this option, note the following requirement: For the parent level (the level that contains the property), the source property or source expression of that level must evaluate to an ID. The system assumes that (at least for this level), the source data is normalized. That is, for the level, the data is in a different table and the source table contains a link to that table.

For an example of **Use as member names**, see the DocD dimension in the Patients cube. Depending on the number of patients you generate, you may see multiple doctors with the same name.

### 6.16 Specifying the Sort Order for Members

This section describes how to control the sort order for members of a level.

#### 6.16.1 For a Data Level

For a data level, by default, members are sorted as follows:

- If the level does not use a range expression, the members are sorted in increasing order by name (using alphabetic sort, no matter what form the names have).
- If the level uses a range expression, the members are sorted in the order determined by the range expression. That is, the first member is determined by the first replacement in the range expression, the second member is determined by the second replacement, and so on.

To change the sort order for the members of a data level, use any of the techniques described in the following subsections.

#### 6.16.1.1 Specifying the Level Sort Option

The simplest way to specify the sort order for members of a level is to set the **Sort** option for the level. Select **asc**, **asc numeric**, **desc**, or **desc numeric**.

- **asc** and **desc** sort members **alphabetically**, in ascending or descending order, and **asc numeric** and **desc numeric** sort **numerically**, in ascending or descending order.

#### 6.16.1.2 Sorting Members by Property Values

To sort the members of a level in order by property values:
1. Define a property for this level, as described in the next chapter. Each member of the level has a value for this property.

2. For this property, select asc, asc numeric, desc, or desc numeric for Sort members by property value.

   asc and desc sort members alphabetically, in ascending or descending order; asc numeric and desc numeric sort numerically, in ascending or descending order.

   (By default, no value is selected, and the property does not affect the sort order of the members.)

If you specify Sort members by property value within more than one property of a level, the members are sorted by the first of those properties. Then they are subsorted by second of those properties, and so on.

### 6.16.1.3 Using Property Names as Member Names

An alternative approach, similar to the preceding, is as follows:

1. Use a source expression for the level that forces the members in the desired order, even if the member names are not desirable.

2. Define a property for this level, as described in the next chapter.

3. For this property, specify a source value or source expression that returns a better name for each. For this property, select the option Use as member names.

For an example, in the Patients cube, the Age level contains the patients’ ages in years. These members would be sorted as 0, 1, 10, 11, and so on by default. These members are, however, sorted numerically. This level is defined as follows:

- The Age level uses the following source expression:

  $CASE($LENGTH(%source.Age),2:%source.Age,:"0"_%source.Age)

- This level also includes the Age property, which directly gives the age in years. This property uses the option Use as member names.

### 6.16.1.4 Listing the Members in the Desired Order

To list the members in the desired order, edit the cube class to add <member> elements to the level. For information, see “Manually Specifying the Member Order for a Level,” later in this book.

### 6.16.2 For a Time Level

By default, members of a time level are sorted in increasing order by time.

To specify a different sort order for the members, specify Sort option. Choose either asc (ascending in time) or desc (descending in time).

### 6.17 Making the Member Names Localizable

For cubes as well as dimensions, hierarchies, levels, and other cube elements, you can specify the Display name value, for which you can later define translations in different languages. The same technique is available for members of a level, if you know all the members in advance and if you edit the cube class in Studio. See “Manually Specifying the Members of a Level,” later in this book.
6.18 Defining Dependencies Between Levels in Different Hierarchies

In some cases, there is a virtual dependency between levels that are not in a hierarchy with each other. For example, you might have a cube with a Country level and a Product level. These levels are logically independent from each other; theoretically any product could be sold in any country. It would not make sense to put these levels into a hierarchy in the same dimension.

But if specific products are sold only in specific countries, there is a virtual dependency between these levels. When a user selects a country, it is desirable to show only the products appropriate for that country. In such a case, you can add a dependency between the levels as follows:

1. In Studio, open the cube class.
2. Find the definition of the dependent level.
3. Add the following to this definition:

   ```
   dependsOn="[dim].[hier].[level]"
   ```

   Where `dim` is the name of the other dimension, `hier` is the name of the hierarchy, and `level` is the name of the level. `[dim].[hier].[level]` is the MDX identifier of the level.

   For example:

   ```
   level name="Product" displayName="Product"
   sourceProperty="SKU"
   dependsOn="[Region].[H1].[Country]" />
   ```

   If the level depends on multiple levels, specify a comma-separated list. For example:

   ```
   dependsOn="[dim1].[hier1].[level1],[dim2].[hier2].[level2]"
   ```

The `dependsOn` attribute adds another index to the fact table and consequently consumes more disk space. Use this attribute sparingly. Also, the `dependsOn` attribute is completely unrelated to the `DependsOn` compiler keyword.

6.18.1 Example

To see how this feature works, try the following demonstration:

1. In Studio, edit the Patients cube class (`DeepSee.Model.PatientsCube`). Add the following to the Patient Group level of the PatGrpD dimension:

   ```
   dependsOn="[HomeD].[H1].[ZIP]"
   ```

   For example:

   ```
   <level name="Patient Group" displayName="Patient Group"
   sourceExpression='$CASE(%source.PatientGroup,"A":"Group A","B":"Group B","None")'
   factName="DxPatGroup" description="Patients are assigned to Patient Groups, in this fictitious study."
   dependsOn="[HomeD].[H1].[ZIP]"/>
   ```

2. Save and compile the class.
3. In the Terminal, execute the `ReassignPatients()` method in `DeepSee.Populate`, in the `SAMPLES` namespace:

   ```
   d ##class(DeepSee.Populate).ReassignPatients()
   ```
This method modifies the data for this sample so that there is a virtual dependency between the Patient Group level and the ZIP code level. Patients in specific ZIP codes belong either to Patient Group A or no group; patients in the other ZIP codes belong either to Patient Group B or no group. The method synchronizes the cube, so there is no need to rebuild this cube after executing this method.

Then open the Basic Dashboard Demo and try the Home ZIP Code and Patient Group filters. When you choose a ZIP code, that affects the list in the Patient Group filter.

6.19 Changing the Order of Dimensions in the Cube

To change the order of dimensions in the cube:

1. Click Reorder.
   The system displays a dialog box.
2. Click Dimensions.
3. Optionally click Alphabetize to alphabetize them.
   This affects the list immediately. You can then reorganize the list further if needed. Also, when you add dimensions, they are not automatically alphabetized.
4. Click the name of a dimension and then click the Up button or the Down button as needed.
5. Repeat as needed for other dimensions.
6. Click OK.

The order of the dimensions in the cube affects how they are displayed in the Analyzer. It does not have any other effect. Some customers choose to alphabetize their dimensions for convenience; others put more-often used dimensions at the top of the list.

6.20 Changing the Order of Levels in a Hierarchy

To change the order of levels in a hierarchy:

- To move a level up in the hierarchy, click the Up button in the row for that level.
- To move a level down in the hierarchy, click the Down button in the row for that level.

Important: The order of the levels in a hierarchy determines the structure of the hierarchy, as noted in “Defining Hierarchies Appropriately,” earlier in this book.

6.21 Specifying the Column Names in the Fact Table

When you compile a cube class, DeepSee generates a fact table class and some related classes. When you build a cube, DeepSee populates these tables, which are described in “Details for the Fact and Dimension Tables.”
By default, DeepSee generates the names of the columns in the fact table, but you can specify the column names to use instead. To do so, specify a value for the **Field name in fact table** option for each applicable level; this option is not available for time levels. Take care to use unique names.

### 6.22 Validating Your Levels

After defining levels, you should build the cube and validate that the levels behave appropriately. For each level, use the Analyzer and create a new pivot table that displays the level as rows.

1. Click **DeepSee**, click **Analyzer**, and then click **Go**.
   
   **Tip:** If the Analyzer is already open, just click the **Analyzer** link at the top of the page.

2. If the left area is not currently displaying the cube you are validating, click **Open** and choose the cube.

3. In the left area, find the dimension that contains the level, and expand that dimension.

4. Drag and drop the level to the **Rows** box.

In this pivot table, look for the following items:

- Make sure that the level has the correct number of members.
  
  If the level has duplicate members with the same name, see the section “**Duplicate Member Names,**” earlier in this book.
  
  If a member is missing, use the Management Portal to find the source records that should have been associated with this member. Make sure that the source property is in the expected form needed by your source expression.

- Make sure that the member names are in the correct form, as needed by the business users.

- Make sure that the null member, if any, has a suitable name, as needed by the business users.

- Make sure that the members are sorted in the desired order.
  
  If not, see the preceding section.

- If the level uses ranges, do the following to validate that the level is not dropping any records because of errors with range ends:
  
  1. Click the Pivot Options button ![Pivot Options button](image)
  2. In the **Row Options** area, click **Show Totals** and then click **OK**.

The total line shows the total of the values above it. This total should equal the total number of records in the cube (unless you do not want it to do so). If it does not, examine the ranges carefully to find any gaps. For example, consider the following pivot table (for a cube that contains 10000 patients):

<table>
<thead>
<tr>
<th>Age Group</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 29</td>
<td>4,179</td>
</tr>
<tr>
<td>30 to 59</td>
<td>4,001</td>
</tr>
<tr>
<td>60+</td>
<td>1,573</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>9,753</strong></td>
</tr>
</tbody>
</table>

This version of the **Age Group** level uses the following ranges:
Patients who are 29, 30, 59, or 60 are not included in any member in this incorrectly defined level.

**Tip:** In the Analyzer, you can access the generated MDX query for a pivot table (to do so, click the Query Text button). Then you can save that query to a text file and rerun the query later programmatically. This technique enables you to revalidate the model again later, if the data has changed. Sometimes new data contains values that you did not consider when creating the model.

For information on running queries programmatically, see the DeepSee Implementation Guide.

Another useful tool is the `%AnalyzeMissing()` method of `%DeepSee.Utils`. This method analyzes every non-computed level, measure, and relationship within the given cube and return the number of facts that have no value for each. For example:

```plaintext
Do ##class(%DeepSee.Utils).%AnalyzeMissing("patients")
```

<table>
<thead>
<tr>
<th>Level</th>
<th>#Missing</th>
<th>%Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lvl [AgeD].[H1].[Age]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [AgeD].[H1].[Age Bucket]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [AgeD].[H1].[Age Group]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [AllerD].[H1].[Allergies]</td>
<td>388</td>
<td>38.80%</td>
</tr>
<tr>
<td>Lvl [AllerSevD].[H1].[Allergy Severities]</td>
<td>388</td>
<td>38.80%</td>
</tr>
<tr>
<td>Lvl [BirthD].[H1].[Date]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [BirthD].[H1].[Decade]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [BirthD].[H1].[Period]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [BirthD].[H1].[Quarter Year]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [BirthD].[H1].[Year]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [BirthQD].[H1].[Month]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [BirthQD].[H1].[Quarter]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [BirthTD].[H1].[Birth Time]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [ColorD].[H1].[Favorite Color]</td>
<td>221</td>
<td>22.10%</td>
</tr>
<tr>
<td>Lvl [DiagD].[H1].[Diagnoses]</td>
<td>831</td>
<td>83.10%</td>
</tr>
<tr>
<td>Lvl [DocD].[H1].[Doctor]</td>
<td>159</td>
<td>15.90%</td>
</tr>
<tr>
<td>Lvl [DocD].[H1].[Doctor Group]</td>
<td>271</td>
<td>27.10%</td>
</tr>
<tr>
<td>Lvl [GenD].[H1].[Gender]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [HomeD].[H1].[City]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [HomeD].[H1].[ZIP]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Mr [Measures].[Age]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Mr [Measures].[Allergy Count]</td>
<td>388</td>
<td>38.80%</td>
</tr>
<tr>
<td>Mr [Measures].[Avg Age]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Mr [Measures].[Avg Allergy Count]</td>
<td>388</td>
<td>38.80%</td>
</tr>
<tr>
<td>Mr [Measures].[Avg Enc Count]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Mr [Measures].[Avg Test Score]</td>
<td>200</td>
<td>20.00%</td>
</tr>
<tr>
<td>Mr [Measures].[Encounter Count]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Mr [Measures].[Test Score]</td>
<td>200</td>
<td>20.00%</td>
</tr>
<tr>
<td>Lvl [PatGrpD].[H1].[Patient Group]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [PatGrpD].[H1].[Tested]</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Lvl [ProfD].[H1].[Industry]</td>
<td>564</td>
<td>56.40%</td>
</tr>
<tr>
<td>Lvl [ProfD].[H1].[Profession]</td>
<td>564</td>
<td>56.40%</td>
</tr>
</tbody>
</table>
```

TOTAL FACTS 1,000
This chapter describes how to define properties. It discusses the following topics:

- How to create a property
- How to get the value of a property at runtime
- How to change the order of properties in a level
- How to specify column names in the dimension tables
- A summary of special uses for properties

### 7.1 Adding a Property

To add a property to an existing level:

1. Click either the level or an property within that level, in the Model Viewer.
   
   This action indicates where the property is to be added:
   
   - If you click the level, the new property will be added after all the other properties in this level.
   - If you click a property, the new property will be added immediately before that property.
   
   **Note:** The order of the properties in a level is important if the properties are used to sort members. See “Specifying the Sort Order for Members of a Data Level,” in the previous chapter.

2. Click **Add Element**.
   
   The system displays a dialog box.

3. For **Enter New Item Name**, type a property name.
   

4. Click **Property**.

5. Click **OK**.

6. Select the property in the Model Viewer.

7. In the **Details Area**, specify a value for **Property** or **Expression**. See “Defining the Source Values for a Dimension or Level” and “Details for Source Expressions,” earlier in this book.
Alternatively, drag a class property from the Class Viewer and drop it onto a level in the Model Viewer. Then make changes if needed in the Details Area.

Also see “Other Common Options,” earlier in this book.

### 7.1.1 Defining a Property for a List-based Level

If the associated level is list-based, you must define the property as follows:

- Specify a value for **Expression**.
- In the expression, refer to the `%value` variable. This variable contains the value of the associated list element.

For example, consider the Allergy Severities level in the Patients cube. This level has a property that controls the sort order of the members of this level. This property (SeveritySort) is defined by the following expression:

```plaintext
$cube.GetSeveritySort(%value)
```

This executes the `GetSeveritySort()` method in the cube class and the allergy severity (a string) as a argument. The method returns an integer.

### 7.1.2 Specifying a Format String

In Studio, you can specify a format string, which enables you to specify the display format for the property. You can override this formatting in the Analyzer (or in manually written MDX queries). For details, see “<property>” in the first appendix.

### 7.2 Getting a Property Value at Runtime

In some cases, you can define a property so that its value is retrieved at runtime from the appropriate source table. The requirements are as follows:

- For the property, select the **Get value at run time** option.
- For the parent level (the level that contains the property), the source property or source expression of that level must evaluate to an ID.

  The system assumes that (at least for this level), the source data is normalized. That is, for the level, the data is in a different table and the source table contains a link to that table.

  This requirement means that, by default, the member names for this level are IDs, which is not generally suitable for your users. In this case, also configure a property of this level for use as display names for the members. This can be the same property or a different property.

  See “Using Property Values as the Member Names,” in the previous chapter.

### 7.3 Changing the Order of Properties in a Level

To change the order of properties in a level:

- To move a property up in the level, click the Up button in the row for that level.
• To move a property down in the level, click the Down button in the row for that level.

The order of properties in a level can affect the sort order of the members of the level. See “Specifying the Sort Order for Members of a Data Level,” in the previous chapter.

### 7.4 Specifying the Column Names in the Dimension Tables

When you compile a cube class, DeepSee generates a fact table class and related classes for the dimensions. When you build a cube, DeepSee populates these tables, which are described in “Details for the Fact and Dimension Tables.” The properties for a level are stored in the corresponding dimension table.

By default, DeepSee generates the names of the columns in the fact table, but you can specify the column names to use instead. To do so, specify a value for the **Field name in fact table** option for each property. Take care to use unique names.

### 7.5 Special Uses for Properties

By default, properties have no effect on the members of the level to which they belong. You can, however, use properties to modify the members in either or both of the following ways:

• You can use the property values as the member names, overriding the default member names defined by the level definition.

  To do this, use the **Use as member names** option and (optionally) **Get value at run time** options; see “Using Property Values as the Member Names,” in the previous chapter.

  Note that if you select **Use as member names**, and if the source value is null, the system instead uses the appropriate null replacement string; this is either the null replacement string for the level (if defined) or for the cube (if defined). See “Specifying the Null Replacement String for a Level” and “Specifying Cube Options.”

• You can use the property values to specify the default sort order of the members within the level.

  To do this, use the **Sort members by property value** option; see “Specifying the Sort Order for Members of a Data Level,” in the previous chapter.

  The **Sort members by property value** option has no effect if the level uses a range expression.

**Tip:** Within a level, member names are not required to be unique, as noted in the section “Defining Member Keys and Names,” earlier in this book. In a well-defined cube, however, each member of a given level does have a unique key. When a user creates a query in the Analyzer, DeepSee automatically uses the member keys instead of the names. The user can expand the list of members and separately drag and drop different members that have the same names. For convenience of the users, InterSystems suggests that you also add a level property whose value is the key. For such a property, simply base the property on the same source property or source expression that the level uses. The users can then display this property in order to determine the unique identifier that DeepSee uses for the member.
This chapter describes how to define measures. It discusses the following topics:

- How to create a measure
- How to specify the type of measure
- How to specify the aggregation for a measure
- How to specify a searchable measure
- How to specify a format string
- How to specify a format string for a date measure
- How to change the order of measures in a cube
- How to specify column names in the fact table

DeepSee automatically creates a measure whose default name is Count. To override this default name, specify the **Count measure caption** option for the cube; see “Specifying Cube Options,” earlier in this book.

### 8.1 Adding a Measure

To add a measure, drag a class property from the Class Viewer and drop it onto the **Measures** label in the Model Viewer. Then make changes if needed in the Details Area.

Or do the following:

1. Click **Add Element**.
   - The system displays a dialog box.
2. For **Enter New Item Name**, type a measure name.
3. Click **Measure**.
4. Click **OK**.
5. Select the measure in the Model Viewer.
6. Specify the following options, at a minimum:
   - **Type** — Specifies the measure type, as described in “Specifying the Measure Type.”
8.2 Specifying the Measure Type

The Type option specifies the kind of data the measure expects in the source data, as well as the type used in the generated fact table, as follows:

<table>
<thead>
<tr>
<th>Measure Type</th>
<th>Expected Type of Source Data</th>
<th>Type Used by Measure</th>
<th>Measure Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Numeric data</td>
<td>%Double</td>
<td>Numeric value. Default aggregation is SUM.</td>
</tr>
<tr>
<td>integer</td>
<td>Numeric data (any fractional values are removed by truncation)</td>
<td>%Integer</td>
<td>Integer value. Default aggregation is SUM.</td>
</tr>
<tr>
<td>age</td>
<td>Date/time data in $Horolog format</td>
<td>%Integer</td>
<td>Age value in days. Can be used only with AVG (default), MIN, and MAX aggregations. Can be used only with AVG (default), MIN, and MAX aggregations.</td>
</tr>
<tr>
<td>date</td>
<td>Date/time data in $Horolog format</td>
<td>%DeepSee.Datatype.dateTime</td>
<td>Date value (in $Horolog format, with seconds removed). Can be used only with AVG, MIN, and MAX (default) aggregations.</td>
</tr>
<tr>
<td>boolean</td>
<td>0 or 1</td>
<td>%Boolean</td>
<td>Boolean value that can be aggregated. Default aggregation is COUNT.</td>
</tr>
<tr>
<td>string</td>
<td>Any</td>
<td>%String</td>
<td>String values to be stored in the fact table; these are not indexed. Can be used only with COUNT.</td>
</tr>
<tr>
<td>text</td>
<td>Any</td>
<td>%Text</td>
<td>String values to be stored in the fact table; these are indexed. Can be used only with COUNT.</td>
</tr>
<tr>
<td>iKnow*</td>
<td>Any, but particularly text values</td>
<td>%GlobalCharacterStream or %String, depending on the selected source</td>
<td>Text values to be processed and indexed using the iKnow Smart Indexing API. This measure cannot be dragged and dropped in the Analyzer.</td>
</tr>
</tbody>
</table>

*For information on the iKnow type, see “Using Unstructured Data in Cubes,” later in this book.
8.3 Specifying How to Aggregate a Measure

The Aggregate option specifies how to aggregate values for this measure, whenever combining multiple records. If you specify this, use one of the following values:

- **SUM** (the default) — Adds the values in the set.
- **COUNT** — Counts the records for which the source data has a non-null (and nonzero) value.
- **MAX** — Uses the largest value in the set.
- **MIN** — Uses the smallest value in the set.
- **AVG** — Calculates the average value for the set.

For a boolean or a string measure, select **COUNT**.

8.4 Specifying a Searchable Measure

You can specify that a measure is searchable; if so, you can filter records used in a pivot table by the value of that measure. To specify a measure as searchable, select the **Searchable** check box in the Details Area.

8.5 Specifying a Format String

**Note:** For date measures, see the next section.

The Format string option enables you to specify the display format for the data. You can override this formatting in the Analyzer (or in manually written MDX queries). To specify the formatting for a measure in the Architect, do the following while the measure is displayed:

1. Click the Find button.

The system displays a dialog box that includes the following fields:
Here:

- **Positive piece** specifies the format to use for positive values.
- **Negative piece** specifies the format to use for negative values.
- **Zero piece** specifies the format to use for zero.
- **Missing piece** specifies the format to use for missing values; this is not currently used.

In each of these, **Format string** specifies the numeric format, and **Color** specifies the color.

The details are different for date-type measures.

2. Specify values as needed (see the details after these steps).

3. Click OK.

### 8.5.1 Format String Field

The **Format string** field is a string that includes one of the following base units:
### Specifying a Format String for a Date Measure

To specify the format string for a date measure, enter a value like the following into the **Format string** field:

%date%^color
8.7 Changing the Order of Measures in the Cube

To change the order of measures in the cube:

1. Click **Reorder**.
   
   The system displays a dialog box.

2. Click **Measures**.

3. Optionally click **Alphabetize** to alphabetize them.
   
   This affects the list immediately. You can then reorganize the list further if needed. Also, when you add measures, they are not automatically alphabetized.

4. Click the name of a measure and then click the Up button or the Down button as needed.

5. Repeat as needed for other measures.

6. Click **OK**.

The order of the measures in the cube affects how they are displayed in the Analyzer. It does not have any other effect. Some customers choose to alphabetize their measures for convenience; others put more-often used measures at the top of the list.

8.8 Specifying the Column Names in the Fact Table

When you compile a cube class, DeepSee generates a fact table class and some related classes. When you build a cube, DeepSee populates these tables, which are described in “Details for the Fact and Dimension Tables.”

By default, DeepSee generates the names of the columns in the fact table, but you can specify the column names to use instead. To do so, specify a value for the **Field name in fact table** option for each property. Take care to use unique names.
Defining Listings

This chapter describes how to define listings. It discusses the following topics:

- How to add a listing
- Options for a simple listing
- Options for a data connector listing
- Options for a custom listing
- How to define a map-type listing

In the Studio, you can define formatting for a listing. See the details for "<listing>" in the appendix "Reference Information for Cube Classes."

9.1 Adding a Listing

To add a listing:

1. Click Add Element.
   The system displays a dialog box.
2. For Enter New Item Name, type a listing name.
3. Click Listing.
4. Click OK.
5. Select the listing name in the Model Viewer, in the Listings section.
6. Optionally specify the following details:
   - Display name — Localizable name of the listing. If you do not specify this, the user interface instead displays the logical name.
   - Description — (Optional) Optional description of the listing.
   - Resource — (Optional) Specify the Caché resource that secures the listing.
7. Define the listing as described in the following sections.
Note: When you define listings, it is important to consider whether the cube itself is based on a data connector. If it is, then all listings for that cube must be based on the same data connector on which the cube is based.

If the cube is not based on a data connector, however, its listings can be based on data connectors or on the same source class used by the cube.

9.2 Defining a Simple Listing

A simple listing uses fields in the source table that is used by the cube.

For a simple listing, specify the following options:

- **Field list** — Specifies a comma-separated list of fields in the source table to display. To specify this option, click the Search button next to Field list to display a dialog box where you can select the fields. If you do so, the Architect displays the following:

To use this dialog box:

- If necessary, expand the items in the **Source Class** tree.
– To add a property, double-click the property name in the **Source Class** tree. The property is then added to the end of the list shown in **Field List**. Note that the value shown is the SqlFieldName for the property, which might be different from the property name.

– To move an item up or down, click it in **Field List** and then use the Move Up ❯ and Move Down ❯ buttons as needed.

– To edit an item, click it in **Field List**, make changes in **Edit Field**: and then click **Update**. For example, you can add an SQL alias. See the subsection “**Additional Options**.”

– To delete an item, click it in **Field List** and then click the Delete ✗ button.

When you are done, click **OK**.

Or type directly into **Field list**. For example:

```
PatientID,Age,Gender,HomeCity->Name AS "Home City",TestScore AS "Test Score"
```

- **Order by** — Specifies a comma-separated list of fields in the source table by which to sort the listing (these do not need to be included in **Field list**). The overall sort is controlled by the first field in the list, the secondary sort is controlled by the second field, and so on.

To specify this option, click the Search button next to **Field list** to display a dialog box where you can select the fields. This dialog box is a simpler version of the one for **Field list**.

Or type directly into **Order By**. For example:

```
Age,Gender
```

After a field name, you can include the ASC or DESC keyword to sort in ascending or descending order, respectively.

- Ignore the **Data Connector** field and the **Custom SQL query** field.

### 9.2.1 Additional Options

Note the following points:

- You can use Caché arrow syntax to refer to a property in another table. See “**Special Features**” in *Using Caché SQL*.

  ```
PatientID,HomeCity,PrimaryCarePhysician->DoctorGroup
```

- You can include aliases.

  ```
PatientID,Age,Gender,HomeCity->Name AS "Home City",TestScore AS "Test Score"
```

  Or:

  ```
PatientID,Age,Gender,HomeCity->Name "Home City",TestScore "Test Score"
```

- You can use standard SQL and Caché SQL functions, if you enclose the function name within parentheses so that it is not interpreted as a field name.

  ```
(UCASE(PatientID)),%EXTERNAL(Gender)
```

- You can use more advanced SQL features if you use `source.field_name` rather than `field_name`.

  ```
%ID,'$'||source.Sales AS Sales
```

In this case, the Sales column displays the Sales field, preceded by a dollar sign ($).
9.3 Defining a Data Connector Listing

A data connector listing uses fields in a data connector.

For such a listing, specify the following options:

- **Data Connector** — Select the data connector class to use.
- **Field list** — Specify the fields to include. These must be fields in the data connector. You cannot include aliases, SQL functions, or Caché arrow syntax. By default, the listing displays only a property named %ID, if that exists.
- **Order by** — Not used.
- **Ignore the Custom SQL query field.**

**Note:** If a cube is based on a data connector, all its listings must also be based on that same data connector.

9.4 Defining a Custom Listing

A custom listing uses fields from some other table, not the source table used by the cube, and not a data connector.

For a custom listing, specify the following options:

- **Custom SQL query** — Select this.
- **Custom SQL** — Specify an SQL SELECT query.

For any listing, the system creates a temporary listing table that contains the set of source ID values that correspond to the facts used in the current context (the context in which the user requests the listing). Your custom SQL query selects fields from the source table (or possibly other tables) that match the given source IDs. Internally, the overall query might be as follows:

```
SELECT source.Field1, source.Field2
FROM DeepSee_Study.Patient source, internal-listing-table-name list
WHERE source.%ID=list.sourceID AND list.queryKey='2144874459'
```

When you specify the query, however, you specify tokens to replace some or all of these details.

For **Custom SQL**, provide a value with the following basic form:

```
SELECT list of field names FROM $$$SOURCE WHERE $$$RESTRICT
```

Where:

- For **list of field names**, use a comma-separated list of field names. For a field in the source table, use a reference of the form `source.field_name_in_source_table`. You can include Caché arrow syntax as well.
- The **$$SOURCE** token establishes `source` as the alias for that table. Internally this token is replaced by something like the following: `DeepSee_Study.Patient source, internal-listing-table-name list`
You can add other table names here with aliases; see the third example.

- The `$$$RESTRICT` token is replaced by a condition that joins the source table to the temporary listing table; internally this is replaced by something context-dependent like the following: `source.%ID=list.sourceID AND list.queryKey='2144874459'`

  You can add other conditions here; see the third example.

As a simple example:

```sql
SELECT source.PatientID, source.Age, source.HomeCity->Name FROM $$SOURCE WHERE $$RESTRICT
```

As a more complex example, you can also include the ORDER BY clause:

```sql
SELECT source.PatientID, source.Age, source.HomeCity->Name FROM $$SOURCE WHERE $$RESTRICT ORDER BY source.HomeCity->Name
```

You can also retrieve data from other tables by adding the table names to the FROM clause. For example:

```sql
```

### 9.5 Defining a Map-type Listing

By default, any listing displays a table of information. If you have suitable data, you can instead configure the listing to display a map with marked points that indicate geographic information that is relevant to the listing data. For example, the map could highlight sales locations or customer locations. For example:
To do this:

- Using Studio, editing the listing definition to include `listingType=\"map\"`
- Define the listing query so that it contains the fields `Latitude` and `Longitude` (case-sensitive). These fields should contain, respectively, the applicable latitude and longitude in decimal format (rather than degree/minute/second format).

The listing query can also contain other fields. Any additional fields are displayed in a balloon when the user clicks a map position. For example:
Defining a Map-type Listing

(This example is from the Aviation demo; see the chapter “Using Unstructured Data in Cubes.”)
This chapter describes how to add other elements to your cube. It discusses the following topics:

• How to define a calculated measure
• MDX recipes for calculated measures
• How to define calculated members that are not measures
• MDX recipes for non-measure calculated members
• How to define named sets

Note: Users can create additional calculated measures, calculated members, and named sets within the Analyzer.

In MDX terminology, a calculated measure is simply another form of calculated member. This documentation uses the nonstandard phrase *calculated measure* for brevity.

### 10.1 Defining a Calculated Measure

To add a calculated measure:

1. Click **Add Element**.
   - The system displays a dialog box.
2. For **Enter New Item Name**, type the name of the measure.
3. Click **Calculated Member (Measure)**.
4. Click **OK**.
5. Select the calculated measure in the Model Viewer (in the section **Calculated Members**).
6. For **Expression**, specify a numeric MDX expression.
   - The following section has some examples.

Note: The Analyzer displays this new measure with the other measures, in the **Measures** group.
10.2 MDX Recipes for Calculated Measures

For a calculated measure, the MDX expression that you use as its definition should be a numeric expression. The section “Numeric Expressions” in *DeepSee MDX Reference* provides information on all the ways to create numeric expressions in MDX.

For an introduction to DeepSee MDX, see *Using MDX with DeepSee*.

This section discusses recipes for the following scenarios:

- Measures based on other measures
- Measures that show percentages of aggregate values
- Measures that count distinct members
- Semi-additive measures
- Filtered measures, which are null in some cases
- Measures that use KPIs, plugins, or worksheets, which are discussed in later chapters

**Note:** Do not define a calculated measure that refers to a calculated measure that is based on a plugin. Plugins are executed asynchronously, which means that it is not possible to control the relative order of computation.

10.2.1 Measures Based on Other Measures

It is common to base one measure on other measures via an *Expression* like the following:

```
([MEASURES].[my measure 1] + [MEASURES].[my measure 2]) / [MEASURES].[my measure 3]
```

More formally, *Expression* is a numeric-valued MDX expression and can include the following elements:

- References to measures. The syntax is as follows:

  ```
  [MEASURES].[measure name]
  ```

  Or:

  ```
  MEASURES.[measure name]
  ```

  You can omit the square brackets around the measure name, if the measure name contains only alphanumeric characters, does not start with a number, and is not an MDX reserved word.

  The expression is not case-sensitive.

- Numeric literals. For example: 37

- Percentage literals. For example: 10%

  Note that there cannot be any 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. Many MDX functions return numeric values, including *AVG*, *MAX*, *COUNT*, and others. See the *DeepSee MDX Reference* for details. The function names are not case-sensitive.
Tip: The MDX function IIF is often useful in such expressions, for example, to prevent dividing by zero. It evaluates a condition and returns one of two values, depending on the condition.

10.2.2 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 a DeepSee extension. This function executes a MDX subquery, 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 an Expression like the following:

100 * MEASURES.[measure A] / %MDX("SELECT MEASURES.[measure A] ON 0 FROM mycube")

The subquery SELECT MEASURES.[measure A] ON 0 FROM mycube selects the given measure from the cube and aggregates it across all records.

For example:

100 * MEASURES.[%COUNT]/%MDX("SELECT MEASURES.[%COUNT] ON 0 FROM patients")

In the case of the Count measure, you can use a simpler subquery:

100 * MEASURES.[%COUNT]/%MDX("SELECT FROM patients")

The following shows an example that uses the Percent of All Patients calculated measure, which is defined by the preceding Expression:

<table>
<thead>
<tr>
<th>Diagnoses</th>
<th>Patient Count</th>
<th>Percent of All Patients</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>843</td>
<td>84.30</td>
</tr>
<tr>
<td>asthma</td>
<td>55</td>
<td>5.50</td>
</tr>
<tr>
<td>CHD</td>
<td>42</td>
<td>4.20</td>
</tr>
<tr>
<td>diabetes</td>
<td>46</td>
<td>4.60</td>
</tr>
<tr>
<td>osteoporosis</td>
<td>37</td>
<td>3.70</td>
</tr>
</tbody>
</table>

10.2.3 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 levels Doctor Group and Doctor. The calculated measure Unique Doctor Count uses the following Expression, which uses the Doctor level:

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

We can use this measure in a pivot table as follows:
10.2.4 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, a DeepSee 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 expression for **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.

10.2.5 Filtered 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, use an **Expression** like the following:

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

In this case, the `AGGREGATE` function aggregates the given value across all the records that belong to the given member.

For example, the Patients sample has the `Avg Test Score` measure, which 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 \text{Avg Test Score} - \text{CHD}. In this case, you can create a calculated measure that has the following \textbf{Expression}:

\text{AGGREGATE(diag.h1.diagnoses.chd,MEASURES.[avg test score])}

\section*{10.2.6 Measures That Use KPIs, Plugins, or Worksheets}

For any KPI, plugin, or worksheet (all discussed later in this book), you can create a calculated measure that retrieves values from it. Then users can drag and drop this measure within the Analyzer. To create such a calculated measure, use an MDX expression of the following form for \textbf{Expression}:

\text{%KPI(kpiname,propertyname,seriesname,"%CONTEXT")}

Where \textit{kpiname} is the name of the KPI, plugin or worksheet, \textit{propertyname} is the name of the property or column, and \textit{seriesname} is the name of the series. You can omit \textit{seriesname}; if you do, this function accesses the first series in the KPI or plugin.

For MDX-based KPIs and plugins, you can provide a parameter that carries context information. \text{"%CONTEXT"} is a special parameter that provides row, column, and filter context to the KPI or plugin; this information is passed to the base MDX query used by the KPI or plugin. The default for this parameter is \text{"all"}, which uses the row, column, and filter context in combination. For additional options, see the \text{%KPI} function in the \textit{DeepSee MDX Reference}.

For example (for a KPI or plugin with only 1 series):

\text{%KPI("PluginDemo2","Count","%CONTEXT")}

For another example, you can define a calculated measure that uses the sample median plugin (%DeepSee.PlugIn.Median). To do so, use the following \textbf{Expression}:

\text{%KPI("%DeepSee.Median","MEDIAN",1,"%measure","Amount Sold","%CONTEXT")}

\textbf{Note:} Do not define a calculated measure that refers to a calculated measure that is based on a plugin. Plugins are executed asynchronously, which means that it is not possible to control the relative order of computation.

\section*{10.3 Defining a Calculated Member (Non-measure)}

To add a calculated member that is not a measure:

1. Click \textit{Add Element}.
   
   The system displays a dialog box.
2. For \textbf{Enter New Item Name}, type the name of the member.
   
   See \textit{“Names for Model Elements,”} earlier in this book.
3. Click \textit{Calculated Member (Dimension)}.
4. Click \textit{OK}.
5. Select the calculated member in the \textit{Model Viewer} (in the section \textbf{Calculated Members}).
6. For \textbf{Dimension}, type the name of the dimension to which this member belongs.
   
   You can specify any dimension, including an existing dimension that includes non-calculated members or a new dimension.
7. For **Expression**, specify a numeric MDX expression that aggregates other members.
   For details and examples, see “**WITH Clause**” in the *DeepSee MDX Reference*.

## 10.4 MDX Recipes for Non-measure Calculated Members

This section provides recipes for non-measure calculated members for some common scenarios.
For general syntax, see “**WITH Clause**” in the *DeepSee MDX Reference*.
For an introduction to DeepSee MDX, see *Using MDX with DeepSee*.
This section discusses recipes for the following scenarios:
- Time members relative to NOW
- Aggregating multiple members together
- Aggregating ranges of dates
- Aggregating members defined by a term list
- Combining members of different levels, especially for filtering

### 10.4.1 Defining Time Members Relative to NOW

For time levels, it is often useful to have members that are defined relative to 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 **Expression**:

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

For more details and options, see “**NOW Member for Date/Time Levels**” in the *DeepSee MDX Reference*.

### 10.4.2 Aggregating 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 an **Expression** of the following form:

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

For example:

```
%OR({[color].[h1].[favorite color].[red],
    [color].[h1].[favorite color].[blue],
    [color].[h1].[favorite color].[yellow]})
```

In any case, 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.

Or, if the members that you want to combine do not belong to the same level, use the **AGGREGATE** function as follows:

```
AGGREGATE({member_expression, member_expression,...})
```

When you create a member that uses the **AGGREGATE** function, you create a new member that refers to all the records that its component members use. Unlike with **%OR**, a given record is counted more than once if it occurs in multiple members.
10.4.3 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 %TIMERANGE function, which enables you to define a member that consists of all the members in an open-ended range. For example, the following expression defines a range that starts after the 2009 member:

%TIMERANGE(DateOfSale.YearSold.&[2009],,EXCLUSIVE)

The %TIMERANGE function is supported only with time levels and is not supported with relationships.

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]))

10.4.4 Defining an Aggregation 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. (See “Defining Term Lists” in the DeepSee Implementation Guide.)

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 an 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.

10.4.5 Defining a Member for Filtering on Multiple Dimensions

Member-based filters are so useful that it is worthwhile to create members whose sole purpose is for use in filters. 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:

AGGREGATE(member_expression,member_expression,...)

Notice that in this case, we do not use curly braces around the list of members. The syntax shown here forces AGGREGATE to consider the intersection of the members.

For example:

AGGREGATE([BIRTHD].[H1].[YEAR].[NOW],[ALLERSEVD].[H1].[ALLERGY SEVERITIES].[003 LIFE-THREATENING])

The expression ([BIRTHD].[H1].[YEAR].[NOW],[ALLERSEVD].[H1].[ALLERGY SEVERITIES].[003 LIFE-THREATENING]) is a tuple expression, which is the intersection of the member [BIRTHD].[H1].[YEAR].[NOW] and the member [ALLERSEVD].[H1].[ALLERGY SEVERITIES].[003 LIFE-THREATENING] — that is, all patients who were born in the current year and who have a life-threatening allergy.

Or more generally, use an expression of the following form:

AGGREGATE(set_expression)

10.5 Defining a Named Set

To add a named set:

1. Click Add Element.
   The system displays a dialog box.
2. For Enter New Item Name, type the name of the set.
3. Click Named Set.
4. Click OK.
5. Select the named set in the Model Viewer (in the section Named Sets).
6. For Set expression, specify an MDX set expression.
   For details, see the reference section “Set Expressions” in the DeepSee MDX Reference.
11

Defining Subject Areas

This chapter describes how to define subject areas. It includes the following topics:

- Introduction to the parts of the Architect when displaying a subject area
- How to define a new subject area
- How to filter a subject area
- How to specify other options for a subject area
- How to add items to a subject area
- How to add overrides for measures
- How to add overrides for dimensions, hierarchies, and levels
- How to redefine an existing listing or add a new listing
- How to compile a subject area

Note: In the current release, you cannot hide relationships in a subject area. To hide, add, or override named sets and calculated members (including calculated measures), use Studio. See “Reference Information for Subject Area Classes.”

Also see “Defining Compound Cubes,” in the next chapter.

11.1 Introduction to Subject Areas in the Architect

This section describes what the Architect displays when you are viewing a subject area rather than a cube.

1. Click **Open**, click **Subject Areas**, click **AsthmaPatients**, and then click **OK**.

   Now the system displays the page similar to what you see for a cube, except that it does not have the Class Viewer on the left:
11.1.1 Model Viewer

The left area is the Model Viewer, and it shows the overrides defined in this subject area. For example:

This subject area does not define any overrides or any listings. (Many subject areas include only a filter.) Here you can add overrides, select existing overrides for editing, and delete overrides.

You can also define new listings.

11.1.2 Details Area

The right area is the Details Area, and it shows details for the element that is currently selected in Model Viewer (if any), or for the subject area (if nothing is selected).

For example:
11.2 Defining a Subject Area

To define a subject area:

1. In the Architect, click **New**.
2. Click **Subject Area**.
3. Enter the following information at a minimum:
   - **Subject Area Name** — Name of the subject area to use as the default caption and to use in queries.
   - **Class name for the Subject Area** — Complete package and class name for the subject area class.
   - **Base Cube** — Logical name of the cube on which to base this subject area.
     
     You can either type the cube name or click **Browse** and select the cube.
     
     You can also use a comma-separated list of cubes; see “Defining Compound Cubes” in the chapter “Options for Multiple Cubes.”

     The other options are discussed later in this chapter.

     Apart from the class name, you can edit all options after creating the subject area.
4. Click **OK**.
5. Click the subject definition in the Architect to select it. Then edit the **Depends On** option in the **Details Area**. For the value, specify the full package and class name of the cube class on which this subject area is based.

     The subject area class should always be compiled after the cube class. The **Depends On** setting helps control this.
6. Optionally save the subject area. To do so:
   a. Click **Save**.
   b. Click **OK**.

     The system creates the class.

Or manually create the class as described in the appendix “Reference Information for Subject Area Classes.”
11.3 Filtering a Subject Area

In most cases, you use subject areas to control access to data. In these cases, you specify the filter for the subject area. You can do this in either or both of the following ways:

- You can specify a hardcoded filter. To do so, specify a value for the Filter option in the Architect. For this option, specify an MDX filter expression to use as the filter for this subject area.
  
  This section discusses how to create hardcoded filters.
- You can specify a filter whose value is determined at runtime. To do so, customize the %OnGetFilterSpec() method of the subject area class. This method has access to the hardcoded filter, if any, and can modify that filter.
  
  For information, see the section “Customizing the Subject Area %OnGetFilterSpec Callback” later in this book.

Note: The following subsections apply to MDX filters in general, not just to filters of subject areas.

11.3.1 Building a Filter in the Analyzer and Copying It into Your Model

If you are not familiar with MDX, the easiest way to specify a filter expression is as follows:

1. Create a filter by drag and drop as described in Using the DeepSee Analyzer.
2. Click the Query Text button.
   
The Analyzer displays a dialog box with text like this:

   SELECT FROM [PATIENTS] WHERE [AGED].[H1].[AGE GROUP].&[0 TO 29]
3. Ignore all text before WHERE, which might be considerably more complicated depending on what you have done in the Analyzer. Copy all the text that follows WHERE into the system clipboard or other temporary location.
4. Paste the copied text into the Filter option for the subject area.
   
   For example:

   [AGED].[H1].[AGE GROUP].&[0 TO 29]

11.3.2 Writing Filter Expressions

This section discusses common forms of filter expressions.

Also the DeepSee MDX Reference.

11.3.2.1 Simple Filter

A simple filter refers to a single member:

[AGED].[H1].[AGE GROUP].&[0 TO 29]

This filter accesses only patients in the age range 0 to 29.

The preceding expression is an MDX member expression. Note the following details:

- [AGED] is a dimension in this sample.
- [H1] is the hierarchy in this dimension.
• \([\text{AGE\ GROUP}]\) is a level in this hierarchy.
• \&[0\ TO\ 29]\ is the key for a member of that level.

The preceding member expression refers to the \&[0\ TO\ 29]\ member of the \([\text{AGE\ GROUP}]\) level of the \([\text{H1}]\) hierarchy of the \([\text{AGED}]\) dimension.

Also note the following possible timesaving variations:
• You can specify any of these identifiers in upper case, lower case, or mixed case at your convenience. MDX is not case-sensitive.
• You can omit the square brackets for any identifier that does not include space characters.
• You can omit the hierarchy and level names if the resulting expression is not ambiguous.
  You must always include the dimension name.
• You can use the member name instead of the member key. In most cases, this means that you can simply omit the ampersand (\&) from the member identifier.

### 11.3.2.2 Filter with Set of Members

In other cases, you might need multiple members. For example:

\(\{[\text{aged}].[\text{h1}].[\text{age\ group}].[0\ to\ 29],[\text{aged}].[\text{h1}].[\text{age\ group}].[30\ to\ 59]\}\)

The subject area is filtered to include all records of the fact table that belong to any of the elements of the set. This subject area sees patients in the age group 0 to 29 and patients in the age group 30 to 59 — that is, it sees all patients under 60 years of age.

**Important:** Notice that the list is enclosed by braces; this indicates that the list is an MDX *set*. A set is a union of elements.

### 11.3.2.3 Filter with a Tuple

In some cases, you might need to find only records that meet multiple simultaneous criteria. In this case, you use a tuple expression. For example:

\(\{(\text{aged}).[\text{h1}].[\text{age\ group}].[60+],[\text{diagd}].[\text{h1}].[\text{diagnoses}].[\text{diabetes}]\}\)

This subject area sees patients who are 60 or older and who also have diabetes.

**Important:** Notice that the list is enclosed by parentheses; this indicates that the list is an MDX *tuple*. A tuple is an intersection of elements.

### 11.3.2.4 Filter with Multiple Tuples

You can list multiple tuple expressions within a set. For example:

\(\{(\text{aged}).[\text{h1}].[\text{age\ group}].[60+],[\text{diagd}].[\text{h1}].[\text{diagnoses}].[\text{diabetes}],\quad (\text{colord}).[\text{h1}].[\text{color}].[\text{red}],[\text{allergd}].[\text{h1}].[\text{allergy}].[\text{soy}]\}\)

This subject area sees all patients who meet at least one of the following criteria:
• They are at least 60 years old and who also have diabetes.
• They are allergic to soy and their favorite color is red.
11.4 Specifying Other Subject Area Options

In addition to the previously described options, you can specify the following additional options for a subject area:

- **Disabled** — (Optional) If you select this check box, the override is disabled. When you recompile the subject area, this override is ignored and DeepSee uses the definition given in the cube.

- **Display name** — Localizable name of the subject area.

- **Description** — (Optional) Comments to add to the subject area class definition. Each line is saved as a separate comment line at the start of the class definition.

- **Default member** — (Optional) Default member to use when a query skips an axis. Specify an MDX expression that returns a member that is accessible in this subject area. For information on member expressions, see the DeepSee MDX Reference. If this is not specified, the system uses the default member as defined in the cube.

- **Default listing** — (Optional) Logical name of the default listing for this subject area. This listing must be defined in the cube or in the subject area. If this is not specified, the system uses the default listing as defined in the cube.

- **Owner** — (Optional) Specify the owner of the cube. Specify a Caché user name.

- **Count measure caption** — (Optional) Specify an alternative name for the Count measure.

- **Resource** — (Optional) Specify the Caché resource that secures the subject area.

- **Caption**— (Optional) Specify the caption to display in the Analyzer and other utilities when working with this cube.

- **Domain**— (Optional) Specify the name of the domain to contain the localized strings of this subject area. You might find it convenient to use a single domain for all your cubes and subject areas; in other cases, it might be appropriate to have a separate domain for each cube and subject area. See the chapter “Performing Localization” in the DeepSee Implementation Guide.

11.5 Adding Items to a Subject Area

To add an override or a listing to a subject area, use the following general procedure:

1. Click **Add**.

   The system displays a dialog box on which you can choose an item from the cube:
If the subject area already defines an override for an item, this dialog box displays a check mark for that item.

2. Click Measure, Dimension, or Listing.
   
   With Measure, you can override any measure defined in the cube.
   
   With Dimension, you can override any dimension, hierarchy, or level defined in the cube.
   
   With Listing, you can override any listing defined in the cube. You can also add listings.

3. Click the item or items to override and then click OK.
   
   The system adds the items, displays them in the Model Viewer, and shows their details in the Details Area.

4. Edit the details in the Details Area.

### 11.6 Defining an Override for a Measure

To define an override for one or more measures:

1. Optionally click a measure name in the Model Viewer, to indicate where the new override is to be added.
   
   If you do so, the new override is added before the measure you clicked.

2. Click Add.

3. Click Measure.

4. Click the measure or measures to override and then click OK.
   
   The system adds the measures and displays them in the Model Viewer.

5. To define the override for a measure, click the measure name in Model Viewer and edit the following options in the Details Area. These are all optional.
   
   - **Hidden** — Select this to hide this measure in this subject area. Or clear it to use this measure in this subject area.
   
   - **Display name** — Specify a new display name to replace the one defined in the cube.
   
   - **Description** — Specify a new description to replace the one defined in the cube.
Defining Subject Areas

- **Format string** — Specify a new format string to replace the one defined in the cube. See “Specifying a Format String,” earlier in this book.

**Note:** You cannot define overrides for calculated measures this way. A calculated measure is actually a calculated member; to define an override for it, you must use Studio.

### 11.7 Defining an Override for a Dimension, Hierarchy, or Level

To define an override for one or more dimensions, hierarchies, or levels:

1. Click **Add**.
2. Click **Dimension**.
   
   The system displays the dimensions, hierarchies, and levels of the cube as follows:

   ![Add Elements to Subject Area](image)

   - Subject Area Name: AsthmaPatients
   - What would you like to select from?
     - Measure
     - Dimension
     - Listing

3. Click the item or items to override and then click **OK**.
   
   The system adds the items and displays them in the **Model Viewer**.

   You can click any combination of dimensions, hierarchies, and levels. If you click a hierarchy but not its containing dimension, the system always adds the dimension as well (because the dimension is part of the hierarchy definition). Similarly, if you click a level but not its containing hierarchy or dimension, the system always adds the hierarchy and dimension as well.
4. To define the override for an item, click the item name in Model Viewer and edit the following options in the Details Area. These are all optional.

- **Hidden** — Select this to hide this item in this subject area. Or clear it to use this item in this subject area.
- **Display name** — Specify a new display name to replace the one defined in the cube.
- **Description** — Specify a new description to replace the one defined in the cube.
- **Sort (Levels only)** — Specify a different sort order for the members of this level.

**Note:** If you change the sort attribute for a level in a subject area, that change is not in effect until you rebuild the cube.

11.8 Redefining a Listing or Adding a New Listing

To redefine a listing or define a new listing for the subject area:

1. Optionally click a listing name in the Model Viewer, to indicate where the new override is to be added.
   
   If you do so, the new override is added before the listing you clicked.

2. Click **Add**.

3. Click **Listing**.

   The system displays the listings of the cube and of the subject area as follows:

4. Optionally click the listing or listings to redefine.

5. Optionally type the name of a new listing in the box at the bottom.

6. Click **OK**.

   The system adds the listings (with no definitions) and displays them in the Model Viewer.
7. To define a listing, click the listing name in Model Viewer and edit the options in the Details Area. For details, see the chapter “Defining Listings.”

11.9 Compiling a Subject Area

To compile a subject area class in the Architect:

1. Click **Compile**.
   
The system displays a dialog box.

2. Click **Compile**.
   
   DeepSee starts to compile the class and displays progress as it does so.

3. Click **OK**.

Or open the subject area class in Studio and compile it in the same way that you compile other classes.

**Tip:** Before you compile a subject area class, you must compile the cube class on which it depends. It is useful, but not required, to use Studio to edit the subject area class and add the DependsOn keyword to force the classes to be compiled in the correct order. See “Requirements for a Subject Area Class” in the appendix “Reference Information for Subject Area Classes.”
12

Defining Shared Dimensions and Compound Cubes

This chapter describes how to define shared dimensions and compound cubes. It discusses the following topics:

- Overview
- How to define a formally shared dimension
- How to define an informally shared dimension
- How to define compound cubes

You can define compound cubes in the Architect, but for shared dimensions, you must use Studio.

12.1 Overview

This section provides an overview of shared dimensions and compound cubes.

12.1.1 Shared Dimensions

A shared dimension is a dimension that can be used in more than one cube. A shared dimension enables you to do the following:

- Create a dashboard that includes pivot tables from both cubes.
- On the dashboard, include a filter that uses the shared dimension.
  This filter affects pivot tables from both dimensions, if the tables are configured as the target of the filter.
- Create a pivot table that uses both cubes (only if you also define a compound cube, as discussed in the next section).

Typically, dimensions based on location and time (see the note below) can be shared, even for unrelated cubes.

It might be possible to share other dimensions. For example, suppose that one cube represents transactions and another represents the customers who own the transactions. These two cubes might have common dimensions such as customer class, broker, and so on.

You can share dimensions in either of the following ways:
You can share a dimension formally. In this case, the dimension is defined in one cube and is referenced in the other (or others).

In this case, you can also define a compound cube, which combines two or more cubes, which should have formally shared dimensions. This enables you to combine elements from different cubes in a single pivot table.

You can share a dimension informally. In this case, each cube has its own definition of the dimension, and the cubes are independent of each other.

In this case, you cannot use the cubes together in a compound cube. As noted above, however, you can create a dashboard that includes pivot tables from both cubes and applies filters to both of them.

Note: Date dimensions are automatically shared; that is, a date dimension in one cube automatically can affect other cubes that define a date dimension that has the same name.

12.1.2 Compound Cubes

A compound cube is a subject area that combines multiple cubes (typically two). For these cubes, any dimensions that have the same name must be formally shared dimensions. This enables you to create pivot tables that contain elements from multiple cubes.

The following shows a pivot table created from a compound cube:

<table>
<thead>
<tr>
<th>ZIP</th>
<th>Doctor Count</th>
<th>Patients Per Week</th>
<th>Avg Patient Allergy Count</th>
<th>Avg Patient Test Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>32006</td>
<td>5</td>
<td>600</td>
<td>1.05</td>
<td>73.65</td>
</tr>
<tr>
<td>32007</td>
<td>9</td>
<td>1,035</td>
<td>1.11</td>
<td>76.48</td>
</tr>
<tr>
<td>34577</td>
<td>12</td>
<td>1,473</td>
<td>0.95</td>
<td>75.15</td>
</tr>
<tr>
<td>36711</td>
<td>5</td>
<td>747</td>
<td>1.03</td>
<td>76.01</td>
</tr>
<tr>
<td>38928</td>
<td>9</td>
<td>987</td>
<td>1.14</td>
<td>75.44</td>
</tr>
</tbody>
</table>

In this pivot table:

- The Doctor Count measure and the Patients Per Week measure come from the Doctors cube. The Patients Per Week measure is the number of patients seen per week by the given set of doctors.
- The Avg Patient Allergy Count measure and the Avg Patient Test Score measure come from the Patients cube. The CompoundCube subject area defines overrides so that these measures have different names than in that cube.
- The ZIP level is in a shared dimension used by both of these cubes.

In a compound cube, the available dimensions are the dimensions from the first listed cube and all formally shared dimensions. The available measures include all the measures from all the cubes. The following rules apply:

- For any measure that has the same name in all cubes used in the compound cubes, that measure becomes a common measure. For this measure, values are aggregated across all the cubes. For example, suppose that one cube is Employees and another cube is Patients. If both cubes have a Count measure, those counts are aggregated together.
  DeepSee provides an option for renaming the Count measure so that you can prevent this from occurring when it is not appropriate.
- For any measure that exists only in one cube, it is treated as usual.
• For any level that is formally shared, you can use members of that level with any of the cubes to select subsets of their records. In the example shown previously, the 32006 member corresponds to all doctors that have this ZIP code and all patients that have this ZIP code.

This fact means that the measures of all the cubes can potentially have different values for members of such a level. For example, the measure Patients Per Week (which is specific to doctors) and the measure Avg Patient Allergy Count (which is specific to patients) have different values for each ZIP code.

• For any level that is not formally shared, a member of that level selects a subset of the records from the cube that owns it, but selects all records from the other cube.

This fact means that measures from the cube that define this level can potentially have different values for members of such a level, but measures from the other cubes always have the same value. In the following example, the Doctor Type dimension is not shared:

<table>
<thead>
<tr>
<th>Doctor Type</th>
<th>Doctor Count</th>
<th>Avg Patients Per Week</th>
<th>Patient Count</th>
<th>Patient Avg Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allergist</td>
<td>3</td>
<td>137.33</td>
<td>1,000</td>
<td>35.90</td>
</tr>
<tr>
<td>Anesthesiologist</td>
<td>1</td>
<td>148</td>
<td>1,000</td>
<td>35.90</td>
</tr>
<tr>
<td>Dermatologist</td>
<td>1</td>
<td>148</td>
<td>1,000</td>
<td>35.90</td>
</tr>
<tr>
<td>Emergency Physician</td>
<td>1</td>
<td>159</td>
<td>1,000</td>
<td>35.90</td>
</tr>
<tr>
<td>Gastroenterologist</td>
<td>4</td>
<td>153.75</td>
<td>1,000</td>
<td>35.90</td>
</tr>
<tr>
<td>General Physician</td>
<td>7</td>
<td>146</td>
<td>1,000</td>
<td>35.90</td>
</tr>
<tr>
<td>Internist</td>
<td>4</td>
<td>163</td>
<td>1,000</td>
<td>35.90</td>
</tr>
<tr>
<td>OB/GYN</td>
<td>6</td>
<td>141.50</td>
<td>1,000</td>
<td>35.90</td>
</tr>
<tr>
<td>Pediatrician</td>
<td>8</td>
<td>150.29</td>
<td>1,000</td>
<td>35.90</td>
</tr>
<tr>
<td>Radiologist</td>
<td>3</td>
<td>150.67</td>
<td>1,000</td>
<td>35.90</td>
</tr>
<tr>
<td>Surgeon</td>
<td>2</td>
<td>165.50</td>
<td>1,000</td>
<td>35.90</td>
</tr>
</tbody>
</table>

Note that the Doctor Count and Avg Patients Per Week measures (both of which are specific to doctors) have different values for each doctor type.

The other measures are specific to patients. They have the same value for each doctor type; this is the value aggregated across all patients.

### 12.2 Defining a Formally Shared Dimension

**Note:** Date dimensions are automatically shared; that is, a date dimension in one cube automatically can affect other cubes that define a date dimension that has the same name.

To share a dimension formally, define a dimension in one cube definition and point to that from another cube definition:

• One cube owns the dimension. When that cube is built, DeepSee determines the initial members of all levels of that dimension, in the usual way. When the source class receives additional data and the cube is updated, DeepSee adds additional members for any levels, in the usual way.

• Another cube definition has a simple definition for the shared dimension, as follows:

```xml
<dimension name="dimensionname" sharesFrom="othercubename"/>
```

Or this definition can include overrides that use different source properties or source expressions than are used in the other cube. In this case, the dimension name, hierarchy names, and level names must be the same as in the other cube.
This cube must be built after the cube that owns the dimension.

If the source data for this cube returns additional values for a level of the shared dimension, DeepSee adds those to the dimension table for that level.

The fact table for this cube points to the dimension tables of the other cube, for all levels in this dimension.

The following restrictions apply:

- The cube that owns the dimension must be built first.

- Unless you define overrides, the same level definitions must be appropriate for both cubes. That is, the identical source property or source expression must be applicable in both cubes. For example, if the cube that owns the definition uses the source expression `%source.Item.Category`, that source expression must also be appropriate for the other cube.

- For any cubes that share that dimension: the source values (member keys) must be the same for the levels in the shared dimension.

  For example, consider two cubes, each based on a table that includes a city name. For these cubes to share a level that is based on city name, the city names must be identical, including case, in both of the source tables. (Otherwise, you will end up with multiple, similar members such as Jonesville and JONESVILLE.)

  There is no requirement for both source tables to have the same set of values, however. For example, one source table could list a city that is not in the other one. The dimension tables contain the entire superset of values.

Also, for any filters that use these levels, the list of members includes all the members, from all cubes that share the dimension. So, for example, in a given dashboard, a user might see an unfamiliar city name in a filter drop-down, a city name that does not appear in the data used on that dashboard. The user can select it, but no matching data will be found.

### 12.2.1 Examples

To see examples of formally shared dimensions, see the classes in the package `DeepSee.Model.CompoundCube` in the SAMPLES namespace. For example, in `DeepSee.Model.CompoundCube.Patients`, the DocTypeD dimension is defined in the usual way, as follows:

```xml
<dimension name="DocTypeD" displayName="DocTypeD"
    hasAll="false">
    <hierarchy name="H1" displayName="H1">
        <level name="Doctor Type" displayName="Doctor Type"
            sourceProperty="PrimaryCarePhysician.DoctorType"
            factName="DxDocType" />
    </hierarchy>
</dimension>
```

In `DeepSee.Model.CompoundCube.Doctors`, the DocTypeD dimension is defined as a shared dimension:

```xml
<dimension name="DocTypeD" displayName="DocTypeD"
    hasAll="false" sharesFrom="CompoundCube/Patients" >
    <hierarchy name="H1" displayName="H1">
        <level name="Doctor Type" displayName="Doctor Type"
            sourceProperty="DoctorType"
            factName="DxDocType" />
    </hierarchy>
</dimension>
```

### 12.3 Defining Informally Shared Dimensions

To define an informally shared dimension, ensure that the logical dimension name, its hierarchy names, its level names, and its member keys are the same in all relevant cubes. (The underlying details of the source expressions, transformation options, and so on do not matter. All that matters is that the logical names match and the member keys match.)
When you do this, you can define pivot tables in each of these cubes and then place those pivot tables on the same dashboard. If you include a filter widget that uses one of the shared dimensions, it can affect all the pivot tables.

### 12.3.1 Examples

The Patients cube (in the class DeepSee.Model.PatientsCube) includes the HomeD dimension. This dimension includes an H1 hierarchy, which includes the ZIP and City levels.

The CityRainfall cube (in the class DeepSee.Model.RainfallCube) also contains the HomeD dimension, which differs from the one in the Patients cube only as follows:

- The HomeD dimension has the display name CityD (rather than being the same as the internal name).
- The HomeD dimension has an All member.
- The City level uses the City.Name source property (rather than HomeCity.Name).

These definitions mean that you can use these cubes in different pivot tables on the same dashboard, and have them respond in the same way to any filters that use the HomeD dimension. The dashboard Dashboards/Demo Two Subject Areas Together demonstrates this. It has a pivot table that uses the Patients cube and another pivot table that uses the CityRainfall cube. The dashboard includes filter controls that affect both pivot tables.

Similarly, the Cities cube (in the class DeepSee.Model.CityCube) contains a dimension named HomeD, which includes an H1 hierarchy, which includes the ZIP and City levels. The display name for HomeD is CityD, so that the dimension appears to have a different name in this cube. As before, the source properties used by the levels are different in the Cities cube than in the Patients cube.

### 12.4 Defining Compound Cubes

To create compound cubes, you must use Studio. To create a compound cube, do all the following:

- Create a subject area with the **Base cube** option equal to a comma-separated list of cubes. For example, for the subject area CompoundCube/CompoundCube in SAMPLES, **Base cube** is as follows:  

  CompoundCube/Patients,CompoundCube/Doctors,CompoundCube/CityRainfall

  Also edit the **Depends On** option in the Details Area. For the value, specify the full package and class name of all the cube classes.

  Any subject area class should always be compiled after the cube class or classes on which it is based. The **Depends On** setting helps control this.

- In the cubes that the compound cube uses, optionally redefine the Count measure. To do so, specify the `countMeasureName` and (optionally) `countMeasureCaption` attributes in the definitions of the cubes. For example:

  ```xml
  <cube xmlns="http://www.intersystems.com/deepsee" name="Doctors" 
    displayName="Doctors" 
    sourceClass="DeepSee.Study.Doctor" 
    countMeasureName="DoctorCount" 
    countMeasureCaption="Doctor Count">
    ... 
  </cube>
  
  This change does not require rebuilding these cubes.
In the compound cube, optionally change the display names of measures names to be more specific, for use in the compound cube. For example:

```xml
<subjectArea xmlns="http://www.intersystems.com/deepsee/subjectarea"
  name="CompoundCube" displayName="CompoundCube"
  baseCube="Doctors,Patients">
  <measure name="Allergy Count" displayName="Patient Allergy Count"/>
  <measure name="Avg Allergy Count" displayName="Patient Avg Allergy Count"/>
  <measure name="Age" displayName="Patient Age"/>
  <measure name="Avg Age" displayName="Patient Avg Age"/>
  <measure name="Test Score" displayName="Patient Test Score"/>
  <measure name="Avg Test Score" displayName="Patient Avg Test Score"/>
  <measure name="Encounter Count" displayName="Patient Encounter Count"/>
  <measure name="Avg Enc Count" displayName="Patient Avg Enc Count"/>
</subjectArea>
```

Recompile any cube definitions that you change. Recompile the compound cube last.

In a compound cube, the available dimensions are the dimensions from the first listed cube and all formally shared dimensions. The available measures include all the measures from all the cubes.

**Note:** Any dimensions that have the same name in both cubes must be formally shared. Any measures that have the same name in both cubes are aggregated together.

### 12.4.1 Example Compound Cube

To see an example of a compound cube, see the class `DeepSee.Model.CompoundCube.CompoundCube` in the `SAMPLES` namespace. This class is defined as follows:

```csharp
{
  /// This XData definition defines the SubjectArea.
  XData SubjectArea [  XMLNamespace = "http://www.intersystems.com/deepsee/subjectarea"  ]
  {
    <subjectArea name="CompoundCube/CompoundCube"  displayName="CompoundCube/CompoundCube"
      baseCube="CompoundCube/Patients,CompoundCube/Doctors,CompoundCube/CityRainfall"  >
    </subjectArea>
  }
}
```

The cube `CompoundCube/Patients`, which is defined in `DeepSee.Model.CompoundCube.Patients` defines all the dimensions.

The other cubes (`CompoundCube/Doctors` and `CompoundCube/CityRainfall`) define dimensions that are shared from the `CompoundCube/Patients`. Notice that not all the dimensions are defined in all the cubes. The following table shows the dimensions available in each cube:

<table>
<thead>
<tr>
<th>Dimension</th>
<th>CompoundCube/Patients cube</th>
<th>CompoundCube/Doctors cube</th>
<th>CompoundCube/CityRainfall cube</th>
</tr>
</thead>
<tbody>
<tr>
<td>BirthD</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>DocD</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DocTypeD</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>HomeD</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
The HomeD dimension is defined in all three cubes, so this dimension affects the measures of all three cubes. For example, the dashboard Demo Compound Cube includes this pivot table:

<table>
<thead>
<tr>
<th>ZIP</th>
<th>City</th>
<th>Patient Count</th>
<th>Doctor Count</th>
<th>Avg Monthly Rainfall Inches</th>
</tr>
</thead>
<tbody>
<tr>
<td>32006</td>
<td>Juniper</td>
<td>119</td>
<td>7</td>
<td>1.58</td>
</tr>
<tr>
<td></td>
<td>Spruce</td>
<td>116</td>
<td>4</td>
<td>1.61</td>
</tr>
<tr>
<td>32007</td>
<td>Redwood</td>
<td>99</td>
<td>4</td>
<td>1.61</td>
</tr>
<tr>
<td>34577</td>
<td>Cypress</td>
<td>99</td>
<td>5</td>
<td>1.59</td>
</tr>
<tr>
<td></td>
<td>Magnolia</td>
<td>125</td>
<td>2</td>
<td>1.63</td>
</tr>
<tr>
<td></td>
<td>Pine</td>
<td>115</td>
<td>2</td>
<td>1.60</td>
</tr>
<tr>
<td>36711</td>
<td>Centerville</td>
<td>118</td>
<td>4</td>
<td>1.61</td>
</tr>
<tr>
<td>38928</td>
<td>Cedar Falls</td>
<td>101</td>
<td>6</td>
<td>1.59</td>
</tr>
<tr>
<td></td>
<td>Elm Heights</td>
<td>108</td>
<td>6</td>
<td>1.59</td>
</tr>
</tbody>
</table>

The Patient Count measure is defined in CompoundCube/Patients, Doctor Count measure is defined in CompoundCube/Doctors, and Avg Monthly Rainfall Inches measure is defined in CompoundCube/CityRainfall. Notice that the values are different for each measure for each city.

The same dashboard also includes a pivot table that use BirthD for rows:

<table>
<thead>
<tr>
<th>Decade</th>
<th>Patient Count</th>
<th>Doctor Count</th>
<th>Avg Monthly Rainfall Inches</th>
</tr>
</thead>
<tbody>
<tr>
<td>1900s</td>
<td>7</td>
<td>40</td>
<td>1.61</td>
</tr>
<tr>
<td>1910s</td>
<td>19</td>
<td>40</td>
<td>1.61</td>
</tr>
<tr>
<td>1920s</td>
<td>60</td>
<td>40</td>
<td>1.63</td>
</tr>
<tr>
<td>1930s</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Because CompoundCube/Doctors does not define the BirthD dimension, the measure Doctor Count cannot be broken out by birth decade. Notice that the Doctor Count column shows the same number in all cells; this is the total doctor count across birth decades for all patients.

Finally the Demo Compound Cube dashboard also includes a pivot table that use DoctTypeD for rows:

<table>
<thead>
<tr>
<th>Doctor Type</th>
<th>Patient Count</th>
<th>Doctor Count</th>
<th>Avg Monthly Rainfall Inches</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>153</td>
<td></td>
<td>1.60</td>
</tr>
<tr>
<td>Allergist</td>
<td>44</td>
<td>2</td>
<td>1.60</td>
</tr>
<tr>
<td>Anesthesiologist</td>
<td>44</td>
<td>2</td>
<td>1.60</td>
</tr>
<tr>
<td>Cardiologist</td>
<td>11</td>
<td>1</td>
<td>1.60</td>
</tr>
</tbody>
</table>
Because CompoundCube/CityRainfall does not define the DocTypeD dimension, the measure Avg Monthly Rainfall Inches cannot be broken out by doctor type. This measure is aggregated across all patients (by averaging, as defined in the measure).
13
Defining Cube-Cube Relationships

This chapter describes how to define relationships between cubes. It discusses the following topics:

- Overview
- How to define a one-way relationship
- How to define a two-way relationship
- How to build cubes that have relationships

13.1 Overview of Relationships

You can define relationships between cubes as follows:

- One-to-many relationships. Then you can use levels of either cube in both cubes.
  In one cube (the “one” side), the relationship behaves much like a list-based level.
- One-way one-to-many relationships. In one cube, the relationship behaves much like a list-based level. In the other cube, the relationship is not visible.

If you define relationships, you can define a level once rather than multiple times, which minimizes the sizes of fact tables and their indices.

The Patients sample provides four related cubes, with names starting R. The following figure summarizes how these cubes are related:
The one-way arrows represent one-way relationships; the cube that has the arrow pointing away can see the levels of the other cube. The two-way arrows represent two-way relationships.

### 13.1.1 A Look at a One-way Relationship

In this section, we examine a working one-way relationship.

The **RPatients** cube lets you analyze patients and has one-way relationships to the **RCities** and **RDoctors** cubes. If you open the **RPatients** cube in the Analyzer, you see the following cube contents:

The **RCities** and **RDoctors** items are relationships to the **RCities** and **RDoctors** cubes, respectively. If you expand them, you see the levels defined in those cubes:

![Diagram of RPatients cube with measures and dimensions]

When you work with the **RPatients** cube, you can use all these levels in the same way that you use the levels that are defined directly in this cube. The **RDoctors within RCities** is a relationship from the **RCities** cube to the **RDoctors** cube; it is not recommended that you use relationships of relationships, because the result quickly becomes confusing.

### 13.1.2 A Look at a Two-way Relationship

There is a two-way relationship between the **RCities** cube and the **RDoctors** cube. If you open the **RCities** cube, you see the following cube contents:
Similarly, if you open the \texttt{RDoctors} cube, you see the following cube contents:

To define a one-way relationship from one cube to another cube, you define a single relationship in the first cube. To do so, make the following changes in the first cube:

1. Click \texttt{Add Element}.

   The system displays a dialog box.

\section*{13.2 Defining a One-way Relationship}

To define a one-way relationship from one cube to another cube, you define a single relationship in the first cube. To do so, make the following changes in the first cube:
2. For Enter New Item Name, type a relationship name. This determines the logical name of the relationship. It is convenient for this to be the same as the logical name of the other cube.

3. Click Relationship.

4. Click OK.

5. In the Details Area, specify the following values:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display Name</td>
<td>Display name for this relationship. It is useful for this to be the same as the display name of the other cube.</td>
</tr>
<tr>
<td>Property Expression</td>
<td>Specify one of these. The value must be the ID of a record in the base class used by the other cube.</td>
</tr>
<tr>
<td>Cardinality</td>
<td>&quot;one&quot;</td>
</tr>
<tr>
<td>Related cube</td>
<td>Logical name of the other cube.</td>
</tr>
<tr>
<td>Null replacement string</td>
<td>(Optional) Specifies the string (for example, None) to use as the member name if the source data for a relationship is null. There is no default null replacement for relationships.</td>
</tr>
</tbody>
</table>

Do not specify Inverse.

6. Click the cube definition in the Architect to select it. Then edit the Depends On option in the Details Area. This option specifies the class or classes that must be runnable before this class can be compiled.

   By default, this option is blank, and DeepSee automatically sets the DependsOn keyword equal to the name of the source class for the cube.

   To specify this option, specify a comma-separated list of classes and specify the full package and class name for each class in the list. Your list should include the source class for the cube and the cube class on which this cube depends. For example:

   ```
   [ DependsOn = (MyApp.CubeBaseClass, MyApp.OtherCubeClass) ]
   ```

7. Optionally edit the cube class in Studio to define a dependency of this relationship to some other relationship defined in the same cube.

   In some cases, there is a virtual dependency between two relationships. For example, you might have a cube with a Country relationship and a Product relationship. These relationships are logically independent from each other; theoretically any product could be sold in any country. But if specific products are sold only in specific countries, there is a virtual dependency between these relationships. When a user selects a country, it is desirable to show only the products appropriate for that country.

   In such a case, you can add a dependency between the relationships. To do so, specify the dependsOn attribute as described in “Defining Dependencies Between Levels in Different Hierarchies”, earlier in this book. For the value, specify the logical name of another relationship defined in the same cube. (Or, if the relationship depends upon a level, specify the MDX identifier of that level.)

   Note that the dependsOn attribute is completely unrelated to the DependsOn compiler keyword.
### 13.3 Defining a Two-way Relationship

To define a two-way relationship between cubes, you define two complementary `<relationship>` elements, one in each cube. To define these, use the following procedure:

1. Understand the relationship between the base class (class A) used by the first cube and the base class (class B) used by the second cube. Use this to determine the cube class where you will start to define the relationship, as follows:
   - Does one record in class A correspond to multiple records in class B? If so, start with class B.
   - Does one record in class B correspond to multiple records in class A? If so, start with class A.
   - Is there a one-to-one relationship between the classes? If so, start with either class.

For reference, in the following steps, we refer to the first cube as **Dependent Cube** and to the second cube as **Independent Cube**.

2. In the Architect, make the following changes to **Dependent Cube**:
   a. Click **Add Element**. The system displays a dialog box.
   b. For **Enter New Item Name**, type a relationship name. This determines the logical name of the relationship. It is convenient for this to be the same as the logical name of the other cube (*IndependentCubeName*, for example).
   c. Click **Relationship**.
   d. Click **OK**.
   e. In the **Details Area**, specify the following values:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Display Name</strong></td>
<td>Display name for this relationship. It is useful for this to be the same as the display name of the other cube.</td>
<td>Independent Cube Display Name</td>
</tr>
<tr>
<td><strong>Property</strong></td>
<td>Specify one of these. The value must be the ID of a record in the base class used by the other cube.</td>
<td></td>
</tr>
<tr>
<td><strong>Expression</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cardinality</strong></td>
<td>&quot;one&quot;</td>
<td></td>
</tr>
<tr>
<td><strong>Inverse</strong></td>
<td>Value of the inverse relationship in the other cube. It is useful for the name of a relationship to be the same as the logical name of the cube to which it points, so use the logical name of the cube.</td>
<td>DependentCubeName</td>
</tr>
<tr>
<td><strong>Related cube</strong></td>
<td>Logical name of the other cube.</td>
<td>IndependentCubeName</td>
</tr>
</tbody>
</table>
3. Click the cube definition in the Architect to select it. Then edit the **Depends On** option in the Details Area. This option specifies the class or classes that must be runnable before this class can be compiled.

By default, this option is blank, and DeepSee automatically sets the DependsOn keyword equal to the name of the source class for the cube.

To specify this option, specify a comma-separated list of classes and specify the full package and class name for each class in the list. Your list should include the source class for the cube and the cube class on which this cube depends. For example:

```plaintext
[ DependsOn = (MyApp.CubeBaseClass, MyApp.OtherCubeClass)]
```

4. In the Architect, make the following changes to **Independent Cube**:

   a. Click **Add Element**.
      
      The system displays a dialog box.

   b. For **Enter New Item Name**, type a relationship name. This determines the logical name of the relationship. It is convenient for this to be the same as the logical name of the other cube (**DependentCubeName** for example).

   c. Click **Relationship**.

   d. Click **OK**.

   e. In the Details Area, specify the following values:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Display Name</strong></td>
<td>Display name for this relationship. It is useful for this to be the same as the display name of the other cube.</td>
<td>Dependent Cube Display Name</td>
</tr>
<tr>
<td><strong>Cardinality</strong></td>
<td>&quot;many&quot;</td>
<td></td>
</tr>
<tr>
<td><strong>Inverse</strong></td>
<td>Value of the inverse relationship in the other cube. It is useful for the name of a relationship to be the same as the logical name of the cube to which it points, so use the logical name of the cube.</td>
<td>IndependentCubeName</td>
</tr>
<tr>
<td><strong>Related cube</strong></td>
<td>Logical name of the other cube.</td>
<td>DependentCubeName</td>
</tr>
</tbody>
</table>

   f. Optionally edit the cube class in Studio to define a dependency of this relationship to some other relationship defined in the same cube.

In some cases, there is a virtual dependency between two relationships. For example, you might have a cube with a Country relationship and a Product relationship. These relationships are logically independent from each other; theoretically any product could be sold in any country. But if specific products are sold only in specific countries, there is a virtual dependency between these relationships. When a user selects a country, it is desirable to show only the products appropriate for that country.
In such a case, you can add a dependency between the relationships. To do so, specify the `dependsOn` attribute as described in “Defining Dependencies Between Levels in Different Hierarchies”, earlier in this book. For the value, specify the logical name of another relationship defined in the same cube. (Or, if the relationship depends upon a level, specify the MDX identifier of that level.)

Note that the `dependsOn` attribute is completely unrelated to the `DependsOn` compiler keyword.

13.4 Building Cubes That Have Relationships

When you build these cubes, first build the independent cube, which is the one that does not define a source property or source expression for the relationship. More generally, whenever you rebuild the independent cube, you must next rebuild the dependent cube. The suggested best practice is to write a utility method or routine that builds your DeepSee cubes in the appropriate order.
14

Using Unstructured Data in Cubes

This chapter describes how to use unstructured data and the iKnow engine within DeepSee cubes. It discusses the following topics:

• Overview
• How to set up the Aviation demo
• How to load or update an iKnow dictionary
• How to define an iKnow measure
• How to define an iKnow entity dimension
• How to define an iKnow dictionary dimension
• How to add item overrides to an iKnow item level
• How to define an iKnow summary in a listing
• How to add a link from the listing to the complete unstructured text
• How to define a listing for use in the iKnow Content Analysis plugin
• How to generate secondary cubes
• How DeepSee manages iKnow domains

14.1 Overview of iKnow/Cube Integration

The iKnow semantic analysis engine analyzes unstructured data, data that is written as text in a human language such as English or French. This engine is built into Caché in the same way that DeepSee is. For a general introduction, see “Conceptual Overview,” in Using iKnow.

You can use unstructured data within DeepSee cubes, if the source table for a cube includes a property that contains unstructured data (for example, a string field that contains text). Then you can define pivot tables that use iKnow dimensions, and you can use these pivot tables on dashboards as usual.

For example, the source table for a cube might contain both structured and unstructured data. The Aviation demo, discussed later in this chapter, is such an example. For this demo, the source table consists of records of aviation events. For each aviation event, there is a set of short string fields that indicate the incident location, aircraft type, and so on. A longer text field contains the full report of the event.
(You can define also iKnow KPIs, which expose iKnow queries via the DeepSee KPI mechanism. See “iKnow KPIs and DeepSee Dashboards” in Using iKnow.)

### 14.1.1 iKnow Terminology

For iKnow, a key concept is the *entity*. The iKnow engine processes the given text and identifies the entities in it. An entity is a minimal logical unit of text. It is a word or a group of words. Example entities are *clear skies* and *clear sky*. The iKnow language model identifies two kinds of entities:

- A relation is a word or group of words that join two concepts by specifying a relationship between them. A relation is commonly but not always a verb.
- A concept is a word or group of words that is associated by a relation. A concept is commonly but not always a noun or noun phrase.

Also, a *CRC* is a concept-relation-concept sequence, and a *path* is a longer sequence. For more information, see “Logical Text Units Identified by iKnow” in Using iKnow.

Note that *clear skies* and *clear sky* are distinct entities. iKnow does not perform stemming.

Another key (but optional) concept is the *dictionary*. A *dictionary item* is a uniquely identifiable item in your dictionary. A *dictionary term* is a string that could appear somewhere in a text. Typically a single dictionary item corresponds to multiple dictionary terms. For example, you could define a dictionary that associates the item *clouds* with the terms *broken clouds, cumuliform skies, scattered clouds, skies overcast, and sky broken*. Note that these dictionary terms are entities but terms can also be larger units identified by the iKnow engine.

If you define dictionaries, you can perform smart matching, which finds all entities that match any of the given items. The dictionary item *clouds* matches the terms *broken clouds, cumuliform skies, scattered clouds, skies overcast, and sky broken*. For a broader discussion of dictionaries, see “Smart Matching: Creating a Dictionary” and “Smart Matching: Using a Dictionary” in Using iKnow.

### 14.1.2 About iKnow Measures and Dimensions

Unlike other kinds of DeepSee measures, an iKnow measure is not shown in the Analyzer, and you do not directly use it in pivot tables. You define an iKnow measure for each property that you want the iKnow engine to process. Then you can use the measure as the basis of an iKnow dimension.

An iKnow dimension *is* like other DeepSee dimensions; it includes one or more levels, which contain members. Any member consists of a set of records in the source class of the cube. For brevity, the following discussion uses the term *fact* to refer to a record of the source class of the cube.

There are two kinds of iKnow dimensions:

- *Entity dimensions*. An entity dimension contains a single level. Each member of that level corresponds to an entity that the iKnow engine found in the unstructured data.

  The members of this level are sorted in decreasing order by *spread* (number of facts that include this entity). When you expand this level in the left area of the Analyzer, it displays the 100 most common entities. When you use this level as a filter, however, you can search to access any entity.

- *Dictionary dimensions*. A dictionary dimension typically contains two levels as follows:

  - The upper level, the *dictionary level*, contains one member for each dictionary term. That member consists of facts that contain the given dictionary item.

    For example, the *clear* member consists of all facts that contain *any* dictionary item associated with the dictionary term *clear*. This includes the following items: *clear of clouds, clear skies, or clear sky*
– The optional lower level, the *item level*, contains one member for each dictionary item. That member consists of all facts that contain the given dictionary item.

For example, the *clear skies* member consists of each record that contains the *clear skies* dictionary item.

In contrast to an entity dimension, this dimension considers only the entities that match the given dictionary.

Note that because the source text typically includes multiple entities, any given source record is likely to belong to multiple members of a level.

### 14.2 Setting Up the Aviation Events Demo

The SAMPLES namespace provides the Aviation demo, which includes a cube definition, example term lists, and a dashboard.

For this demo, the source table (*Aviation.Event*) consists of records of aviation events. For each aviation event, there is a set of short string fields that indicate the incident location, aircraft type, and so on. A longer text field contains the full report of the event.

For reasons of space, this demo is not initialized when you install Caché. To set up the demo, enter the following command in the Terminal, in the SAMPLES namespace:

```bash
d #class(Aviation.Utils).Setup()
```

To see the sample dashboard:

1. In the Management Portal, access the SAMPLES namespace.
2. Click *Home, DeepSee, User Portal* and then click *View*.

   The system then displays the User Portal, which lists any existing public dashboards and pivot tables in this namespace.

3. Click the *Aviation event reports* dashboard.

The *Aviation event reports* dashboard includes this pivot table:

![Pivot Table](image)

This pivot table is defined as follows:

- The measure is *Count* (count of events).
- The rows display members of the *Report - Injuries* dimension, which is an iKnow dictionary dimension. Each member of this dimension represents an entity that was found by the iKnow engine and matched to an item in an iKnow dictionary.
The columns display members of the Highest Injury level, which is a level in a standard data dimension that is based on a direct classification provided for the reports.

For example, there are 527 reports that were officially classified with None as the highest injury level. Within the unstructured text for these reports, the iKnow engine found 342 reports that have an entity that matches the None item of the Injuries dictionary. Within these reports (officially classified with None as the highest injury level), the iKnow also found two reports that have an entity that matches the minor item of the Injuries dictionary. (When you examine the details of these two reports, you see that these reports were misclassified; there were minor injuries in both of them.)

The following shows another pivot table that uses this sample:

<table>
<thead>
<tr>
<th></th>
<th>Airplane</th>
<th>Balloon</th>
<th>Glider</th>
<th>Gyrocraft</th>
<th>Helicopter</th>
<th>Ultralight</th>
</tr>
</thead>
<tbody>
<tr>
<td>pilot</td>
<td>749</td>
<td>3</td>
<td>21</td>
<td>4</td>
<td>85</td>
<td>1</td>
</tr>
<tr>
<td>airplane</td>
<td>807</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>visual meteorological conditions</td>
<td>561</td>
<td>2</td>
<td>14</td>
<td>4</td>
<td>70</td>
<td>1</td>
</tr>
<tr>
<td>flight</td>
<td>543</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>71</td>
<td>1</td>
</tr>
<tr>
<td>accident</td>
<td>550</td>
<td>3</td>
<td>12</td>
<td>3</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>448</td>
<td>2</td>
<td>9</td>
<td>3</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>engine</td>
<td>379</td>
<td>3</td>
<td>1</td>
<td>42</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>runway</td>
<td>381</td>
<td>11</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>substantial damage</td>
<td>343</td>
<td>8</td>
<td>3</td>
<td>41</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

This pivot table is defined as follows:

- The measure is Count (count of events).
- The rows display members of the Entity dimension, which is an iKnow dimension. Each member of this dimension represents an entity found by the iKnow engine. For example, pilot is an entity found in the unstructured text.
- The columns display members of the Category level of the Aircraft dimension, which is a standard data dimension. This dimension is based on a direct classification provided for the reports. Each member of this dimension represents the source documents associated with a given aircraft category.

This pivot table indicates, for example, that the entity airplane occurs in the reports for 807 events in the Airplane category.

Widgets on this dashboard also include the following options:

- Displays an analysis window that you can use for several specialized kinds of analysis. See the chapter “Using the Pivot Analysis Window” in the DeepSee End User Guide.
- Displays iKnow measure values. This option is enabled when you display a detail listing. See “Displaying iKnow Measure Values” in the DeepSee End User Guide.

### 14.3 Loading and Updating iKnow Dictionaries

This section describes how to load and update iKnow dictionaries for use with DeepSee.

#### 14.3.1 Loading iKnow Dictionaries

To load an iKnow dictionary into Caché:
1. Create a term list that consists of the dictionary items and terms. For information on creating term lists, see “Defining Term Lists” in the DeepSee Implementation Guide.

Each entry in this term list should have the following fields:

- **key** is a unique term that could be found in the text.
- **value** is the corresponding dictionary item.
- **URI** (which is optional) is a unique identifier for the dictionary item within the iKnow domain into which the term list will be loaded. This identifier must be unique for each combination of dictionary name and dictionary item. For example, you could use the following convention:

\[
:dictionary\_name:dictionary\_item
\]

Where **dictionary\_name** is the name of the iKnow dictionary to define or update, and **dictionary\_item** is the value in the **value** field.

If you omit **URI**, the system generates a URI of the form shown here.

Note that **URI** is a custom field in the term list.

For example:

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
<th>URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>broken clouds</td>
<td>clouds</td>
<td>.weather:clouds</td>
</tr>
<tr>
<td>calm winds</td>
<td>mild wind</td>
<td>.weather:wind</td>
</tr>
<tr>
<td>clear of clouds</td>
<td>clear</td>
<td>.weather:visibility</td>
</tr>
<tr>
<td>clear skies</td>
<td>clear</td>
<td>.weather:visibility</td>
</tr>
<tr>
<td>clear sky</td>
<td>clear</td>
<td>.weather:visibility</td>
</tr>
<tr>
<td>cumuliform clouds</td>
<td>clouds</td>
<td>.weather:clouds</td>
</tr>
<tr>
<td>drizzle</td>
<td>rain</td>
<td>.weather:rain</td>
</tr>
<tr>
<td>extreme turbulence</td>
<td>heavy wind</td>
<td>.weather:heavy wind</td>
</tr>
</tbody>
</table>

Use a convenient name for the term list. Note that the term list name does not control the dictionary name.

2. Specify the **Dictionaries** option for each iKnow measure that should use this term list as an iKnow dictionary. See “Defining an iKnow Measure,” later in this chapter.

The **Dictionaries** option specifies the term lists to load as dictionaries for this iKnow measure. DeepSee automatically loads these term lists at cube build time.

**14.3.1.1 Manually Loading a Term List as an iKnow Dictionary**

If you do not use the **Dictionaries** option for an iKnow measure, you must instead invoke a method to load the given term list as an iKnow dictionary. The method is **LoadTermListAsDictionary()**, in the class %iKnow.DeepSee.CubeUtils. It has the following signature:

```
classmethod LoadTermListAsDictionary(pCube As %String, 
pMeasure As %String, 
pTermList As %String, 
pMatch As %Integer = 1, 
pClear As %Integer = 1) as %Status
```

Where:
• \textit{pCube} is the name of the cube that uses this iKnow dictionary.
• \textit{pMeasure} is the name of iKnow measure against which the iKnow engine should match the dictionary terms.
• \textit{pTermList} is the name of term list.
• \textit{pMatch} controls whether the iKnow engine performs matching when you invoke this method. If this argument is 2, the engine matches all existing dictionaries. If it is 1, the engine matches this dictionary. If it is 0, the engine does not perform matching.
• \textit{pClear} controls whether the iKnow engine first deletes existing dictionaries when you invoke this method. If this argument is 2, the engine first deletes all existing dictionaries. If it is 1, the engine first deletes the dictionary defined in the term list. If it is 0, the engine does not delete any dictionaries.

14.3.2 Updating iKnow Dictionaries

If you change a term list that is used as a dictionary, you must update the dictionary. To do so, use the \texttt{UpdateDictionary()} method of \texttt{%iKnow.DeepSee.CubeUtils}:

```csharp
classmethod UpdateDictionary(pTermList As %String, 
pCube As %String = "", 
pMeasure As %String = "", 
pClearFirst As %Boolean = 0) as %Status
```

Where:
• \textit{pTermList} is the name of term list.
• \textit{pCube} is the name of the cube. If you omit this argument, this method is invoked for all cubes in this namespace.
• \textit{pMeasure} is the name of iKnow measure. If you omit this argument, this method is invoked for all iKnow measures in the given cube (or all cubes, depending on \textit{pCube}).
• \textit{pClearFirst} controls whether to drop the existing dictionary before reloading it from the term list. Leave \textit{pClearFirst} as 9 if you only appended to the term list, and use 1 you changed or removed any existing terms.

14.4 Defining an iKnow Measure

To add an iKnow measure:

1. Click \textit{Add Element}.
   The system displays a dialog box.
2. For \textit{Enter New Item Name}, type a measure name.
3. Click \textit{Measure}.
4. Click \textit{OK}.
5. Select the measure in the \textit{Model Viewer}.
6. Specify the following options:
   • \textit{Property} or \textit{Expression} — Specifies a source value that contains unstructured data.
     Or specify a value that contains the full path of a plain text file, where the file contains the text to be processed.
   • \textit{Type} — Select \texttt{iKnow}.
• **iKnow Source** — Specify the type of the source value. Select string, stream, or file. For example, if the selected source property is of type %Stream.GlobalCharacter, select stream. Or if the value is the path to a file, select file.

This option indicates, to the iKnow engine, how to process the values specified in **Property** or **Expression**.

The value domain is for advanced use; see “Alternative Technique: Using an Existing iKnow Domain.”

Note that the **Aggregate** option has no effect on iKnow measures.

7. Save the cube definition in the Architect.

8. If you plan to define one or more iKnow dictionary levels that use this measure, also specify the **Dictionaries** option as follows:
   a. Click the button below **Dictionaries**.
      The system displays a dialog box.
   b. Click the appropriate dictionary in the **Available Dictionaries** list and then click > to move that dictionary to the **Selected Dictionaries** list.
      If **Available Dictionaries** does not list the dictionaries that you need, see “Loading and Updating an iKnow Dictionary,” earlier in this chapter.
   c. Repeat as needed.
   d. Click **OK**.

The **Selected Dictionaries** list is actually a list of term lists. If you follow the steps described here, DeepSee automatically finds the given term lists, loads them as iKnow dictionaries, and performs matching. (If you do not add this attribute, you can instead invoke a method to perform these tasks.)

9. If you are familiar with iKnow domain parameters and you want to use non-default values for them, optionally use the **iKnowParameters** attribute to do so. To do this, edit the cube class in Studio and add the following to the measure definition:

   iKnowParameters="name::value;name::value;name::value"

Use two colons to separate a parameter name and its value. Use a semicolon to separate each name/value pair from the next pair in the list.

The following example specifies the MAT:SkipRelations parameter as 0:

   iKnowParameters="DefaultConfig::Spanish"

For another example:

   iKnowParameters="DefaultConfig::Spanish;MAT:SkipRelations:0"

For details on iKnow domain parameters, see “Setting Up an iKnow Environment” in Using iKnow.

Note that iKnow measures are not stored in the fact table for the cube and are not displayed in the Analyzer. The primary purpose of an iKnow measure is to define an iKnow domain and to serve as the basis of an iKnow dimension. See the next sections.

**14.4.1 Alternative Technique: Using an Existing iKnow Domain**

If you have an existing iKnow domain, you can reuse that. Use the preceding instructions with the following changes:

• Specify **iKnow Source** as domain.
• When you specify the source expression or source property, make sure that it evaluates to the external ID of the iKnow source that corresponds to that DeepSee fact.

• In Studio, add the iKnowDomain attribute to the measure definition. Its value should be the name of an existing iKnow domain.

• Skip steps 8 and 9. That is, do not specify the iKnowDictionaries and iKnowParameters attributes.

Important: In this case, the iKnow domain is managed by iKnow, rather than by DeepSee. At build time, DeepSee does not drop or load any iKnow records. Your custom code must ensure that all data represented and identified by the External ID property/expression at the fact level is properly loaded. At runtime (and only at runtime), DeepSee forwards any calls to iKnow and translates the results from iKnow. DeepSee will not perform any loading activities of its own — not when building the cube, nor when resynchronizing the cube. To load data, specify parameters, or otherwise manage this domain, use the iKnow APIs directly as described in Using iKnow.

14.5 Defining an iKnow Entity Dimension

To add an iKnow entity dimension:

1. Create an iKnow measure for this dimension to use. See the previous topic.
   You can also do this after defining the dimension; if so, edit the dimension later so that it refers to this measure.

2. Click Add Element.
   The system displays a dialog box.

3. For Enter New Item Name, type a dimension name.

4. Click iKnow Dimension and click OK.

5. Click the dimension in the Model Viewer.

6. Make the following changes to the dimension, if needed:
   • iKnow type — Select entity.
   • iKnow measure — Select the iKnow measure for this dimension to use.

7. Click the level in the Model Viewer and optionally modify Name and Display Name.

8. Optionally, to specify the members of this level manually, use Studio and define <member> elements within the level.
   By default, the level consists of all entities, in decreasing order by spread. If you use <member> to specify the members manually, that specifies the members of this level and their order.
   See “Manually Specifying the Members for a Level,” in the chapter “Using Advanced Features of Cubes and Subject Areas.”

Note that it is not necessary to specify anything for Source Values, either for the dimension or for the level. For an iKnow dimension, the associated iKnow measure specifies the source values.
14.6 Defining an iKnow Dictionary Dimension

To add an iKnow dictionary dimension:

1. Load an iKnow dictionary into Caché. See the subsection “Loading iKnow Dictionaries.”
   You can also do this after defining the dimension.

2. Create an iKnow measure for this dimension to use.
   You can also do this after defining the dimension; if so, edit the dimension later so that it refers to this measure.

3. Click Add Element.
   The system displays a dialog box.

4. For Enter New Item Name, type a dimension name.

5. Click iKnow Dimension and click OK.

6. Click the dimension in the Model Viewer.

7. Make the following changes if needed:
   • iKnow type — Select dictionary.
   • iKnow measure — Select the iKnow measure for this dimension to use.

8. Optionally add another level to the same hierarchy in this dimension.
   If the dimension has only one level, that level provides access to dictionary items. If the dimension has two levels, the lower level provides access to entities that match dictionary items.

9. Click each level in the Model Viewer and optionally modify Name and Display Name.

10. Save the cube definition in the Architect.

11. Open the cube class in Studio and find the definition of this dimension. For example, if the dimension has one level, it might look like this (this example shows added line breaks):

    ```xml
    <dimension name="MyDictionaryDimension" disabled="false" hasAll="false" allCaption="MyDictionaryDimension" allDisplayName="MyDict" type="iKnow" iKnowType="dictionary" iKnowMeasure="Report" hidden="false" showHierarchies="default">
        <hierarchy name="H1" disabled="false">
            <level name="Dictionary" disabled="false" list="false" useDisplayValue="true">
            </level>
        </hierarchy>
    </dimension>
    
    Or, if the dimension has two levels:
    ```

    ```xml
    <dimension name="MyDictionaryDimension" disabled="false" hasAll="false" allCaption="MyDictionaryDimension" allDisplayName="MyDict" type="iKnow" iKnowType="dictionary" iKnowMeasure="Report" hidden="false" showHierarchies="default">
        <hierarchy name="H1" disabled="false">
            <level name="Dictionary" disabled="false" list="false" useDisplayValue="true">
            </level>
            <level name="Items" disabled="false" list="false" useDisplayValue="true">
            </level>
        </hierarchy>
    </dimension>
    
    12. In the dictionary level, optionally specify the iKnow dictionary or dictionaries for this level to use. If there are two levels, the dictionary level is the higher of the two levels. If there is one level, that level is the dictionary level.
If you do not specify any iKnow dictionaries, all dictionaries are used.

For each iKnow dictionary to use, add the following between the <level> element and the </level>:

```xml
<member name="dictionary name" />
```

Where dictionary name is the name of an iKnow dictionary.

For example, with a single iKnow dictionary:

```xml
<dimension name="MyDictionaryDimension" disabled="false" hasAll="false" allCaption="MyDictionaryDimension" allDisplayName="MyDict" type="iKnow" iKnowType="dictionary" iKnowMeasure="Report" hidden="false" showHierarchies="default">
  <hierarchy name="H1" disabled="false">
    <level name="Dictionary" disabled="false" list="false" useDisplayValue="true">
      <member name="my dictionary"/>
    </level>
    <level name="Items" disabled="false" list="false" useDisplayValue="true">
      ...
    </level>
  </hierarchy>
</dimension>
```

13. Save the cube definition in Studio.

Note that it is not necessary to specify anything for Source Values, either for the dimension or for the level. For an iKnow dimension, the associated iKnow measure specifies the source values.

### 14.7 Adding Member Overrides to an Item Level

Within a two-level dictionary dimension, by default, the dictionary level determines the members of the lower item level. In the item level, you can add <member> elements that override the definitions determined by the parent.

This is useful, for example, if you want to see only a subset of the dictionary.

If you create these overrides, each <member> element should have the following form:

```xml
<member name="itemURI" displayName="displayName" />
```

Where itemURI is the unique URI of a dictionary item, and displayName is the display name for the dictionary item. See "Loading iKnow Dictionaries."

List the <member> elements in the desired sort order. For example:

```xml
<level name="ReportDictInjuriesDimItem" displayName="Injuries">
  <member name=":injuries:minor" displayName="minor injuries" />
  <member name=":injuries:serious" displayName="serious injuries" />
  <member name=":injuries:fatal" displayName="killed" />
</level>
```

These overrides work as follows:

- If at least one <member> element can be matched to the given dictionary item, this level contains only the members listed by these <member> elements.

- If none of the <member> elements can be matched to dictionary items, these overrides are all ignored.
14.8 Including an iKnow Summary Field in a Listing

It can be useful for your listings to include a summary of the unstructured text. To include such a summary, use the $$$IKSUMMARY token within the listing field definition. This token takes two arguments (in square brackets):

$$IKSUMMARY[iKnowMeasure,summarylength] As Report

Where iKnowMeasure is the name of the iKnow measure to summarize and summary_length is the number of sentences to include in the summary (the default is five). Also, As Report specifies the title of the column. $$IKSUMMARY returns the most relevant sentences of the source, concatenated into a string that is no longer than 32000 characters.

For example:

<listing name="Default" disabled="false" listingType="table"
  fieldList="ID,EventId,Year,AirportName,$$$IKSUMMARY[Report] As Report">
</listing>

Note that $$IKSUMMARY uses the GetSummary() method of %iKnow.Queries.SourceAPI.

14.9 Including a Link from a Listing to the Full Unstructured Text

Your listings can also include a link to a page that displays the full unstructured text. To include such a link, use the $$IKLINK token within the listing field definition. This token takes one argument (in square brackets):

$$IKLINK[iKnowMeasure]

Where iKnowMeasure is the name of the iKnow measure to display.

14.10 Creating a Listing for Use in iKnow Content Analysis Plugin

The Analyzer provides advanced analysis options, which include the iKnow Content Analysis plugin. This option uses a detail listing to display the five most typical and five least typical records. By default, the plugin uses the default listing of the cube.

You might want to create a listing specifically for use here, for reasons of space. If you define a listing named ShortListing, the plugin uses that listing instead.

In either case, the plugin adds a Score column to the right of the columns defined in the listing.

For details on this analysis option, see “iKnow Content Analysis” in the DeepSee End User Guide.
14.11 Generating Secondary Cubes

After you add an iKnow measure to a DeepSee cube (and create iKnow dimensions that use that measure), you can generate secondary cubes that analyze entity occurrences and dictionary matching results.

Always build the main cube before building these cubes.

14.11.1 Entity Occurrence Cube

To generate a cube that represents entity occurrences, use the following command in the Terminal:

```
d ##class(%iKnow.DeepSee.CubeUtils).CreateEOCube(cubename,measurename)
```

Where `cubename` is the name of the cube that contains an iKnow measure, and `measurename` is the name of the iKnow measure.

This method generates a read-only class that provides access to the iKnow entity occurrence data, for benefit of the cube class. The entity occurrence class is named `BaseCubeClass.measurename.EntityOccurrence`, where `BaseCubeClass` is the class name of the base cube class and `measurename` is the name of the iKnow measure.

The method also generates the cube class: `BaseCubeClass.measurename.EOCube`. The new cube definition is as follows:

- The logical name of the cube is `BaseCubemeasurenameEO` where `BaseCube` is the logical name of the base cube and `measurename` is the name of the iKnow measure.
- This cube represents entity occurrences. That is, the fact table for this cube contains one row for each unique entity occurrence.
- The cube defines the `Count` measure, which counts entity occurrences.
- The cube defines the `Entity Value` dimension, which groups entity occurrences by entity value. This is a custom computed dimension; for general information on computed dimensions, see the chapter “Using Advanced Features of Cubes and Subject Areas.”
  
  For this cube, a fact represents an entity occurrence. Note that in this cube, there is a one-to-one relationship between facts and entity values. In contrast, in the main cube, there is a one-to-many relationship between facts and entity values.
- The cube defines the `Roles` dimension, which has the members `concept` and `relation`. These members group the entity occurrences into iKnow concepts and iKnow relations. For information on these terms, see “Logical Text Units Identified by iKnow” in Using iKnow.
- The cube includes a relationship to the base cube. This relationship is called `Main cube`.

The following shows an example pivot table that uses the entity occurrence cube for the Aviation demo:
This pivot table is defined as follows:

- The measure is Count (count of entity occurrences.)
- The rows display members of the Entity Value dimension. Each member of this dimension represents an entity occurrence. For example, the pilot member represents each occurrence of the entity pilot in the sources.
- The columns display members of the Category level of the Aircraft dimension in the main cube. Each member of this dimension represents the source documents associated with a given aircraft category.

This pivot table indicates, for example, that the entity airplane occurred 4980 times in the reports for events in the Airplane category.

### 14.11.2 Matching Results Cube

To generate a cube that represents matching results, use the following command in the Terminal:

```bash
d ##class(%iKnow.DeepSee.CubeUtils).CreateMRCube(cubename,measurename)
```

Where `cubename` is the name of the cube that contains an iKnow measure, and `measurename` is the name of the iKnow measure.

This method generates a read-only class that provides access to the iKnow matching results data, for benefit of the cube class. The matching results class is named `BaseCubeClass.measurename.MatchingResults`, where `BaseCubeClass` is the class name of the base cube class and `measurename` is the name of the iKnow measure.

This method also generates the cube class: `BaseCubeClass.measurename.MRCube`. The new cube definition is as follows:

- The logical name of the cube is `BaseCube.measurename.MRCube` where `BaseCube` is the logical name of the base cube and `measurename` is the name of the iKnow measure.
- This cube represents dictionary matches. That is, the fact table for this cube contains one row for each unique dictionary match.
- The cube defines the Count measure, which counts dictionary matches.
- The cube defines the Score measure, which shows the score of the dictionary matches.
- The cube defines the Dictionary dimension. The Dictionary level groups matches by dictionary, and the Dictionary Item level groups matches by dictionary item.

For this cube, a fact represents a matching result. Note that in this cube, there is a one-to-one relationship between facts and dictionary items. In contrast, in the main cube, there is a one-to-many relationship between facts and dictionary items.
The cube defines the Type dimension, which has the members entity, CRC, path, and sentence. These members group the matches into iKnow entities, CRCs, paths, and sentences. For information on these terms, see “Logical Text Units Identified by iKnow” in Using iKnow.

For this demo, the Type dimension only has the members entity and CRC.

The cube includes a relationship to the base cube. This relationship is called Main cube.

The following shows an example pivot table that uses the matching results cube for the Aviation demo:

<table>
<thead>
<tr>
<th>Injuries</th>
<th>Accident</th>
<th></th>
<th>Incident</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Count</td>
<td>Score</td>
<td>Count</td>
<td>Score</td>
</tr>
<tr>
<td>fatal</td>
<td>106</td>
<td>85.31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>minor</td>
<td>152</td>
<td>151.50</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>none</td>
<td>713</td>
<td>340.52</td>
<td>24</td>
<td>12.85</td>
</tr>
<tr>
<td>serious</td>
<td>93</td>
<td>91.15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This pivot table is defined as follows:

- The measures are Count (count of matching results) and Score (cumulative score of the dictionary matches).
- The rows display members of the Injuries dictionary. Each member of this dimension represents an entity that matches a specific item in this dictionary. For example, the minor member represents each entity in the sources that matches the dictionary item minor.
- The columns display members of the Type dimension in the main cube. Each member of this dimension represents the source documents associated with a given report type, either Accident or Incident.

This pivot table indicates, for example, that the sources of type Incident contain zero matches for the dictionary item fatal. In contrast, the sources of type Accident contain 106 matches for this dictionary item.

### 14.12 iKnow Domain Management in DeepSee

When you use the iKnow features described in this chapter, the system creates one or more iKnow domains. These iKnow domains are managed by DeepSee (unlike iKnow domains that you create directly as described in Using iKnow). To modify them, you should use only the APIs described in this chapter.

DeepSee manages these domains in a way that requires little or no intervention. If you are familiar with iKnow domains, however, you might be interested in the details.

The system creates one iKnow domain for each iKnow measure that you add to a cube. The name of the domain is DeepSee@cubename@measurename where cubename is the logical name of the cube and measurename is the logical name of the iKnow measure.

DeepSee manages these domains as follows:

- When you compile the cube for the first time, the system creates the needed domains.
- When you build the cube, DeepSee invokes the iKnow engine automatically. The iKnow engine processes the text in the iKnow measures and stores the results.
- When you compile the cube again, the system checks to see if the needed domains exist. If so, it reuses them. If not, it creates them.
When it checks whether a given domain can be reused, the system considers the source value or source expression of each iKnow measure (rather than considering the logical name of the iKnow measure). Therefore, when you rename an iKnow measure, the system reuses the existing iKnow domain.

- When you remove an iKnow measure and recompile the cube, the system deletes the corresponding iKnow domain and all associated iKnow engine results.
- When you delete the cube, the system deletes the iKnow domains and removes all associated iKnow engine results.
15
Defining Worksheets

This chapter describes how to define worksheets. It discusses the following topics:

- Overview
- How to access the Worksheet Builder
- How to define worksheets
- How to specify worksheet options
- How to specify additional details for columns and rows
- How to edit existing worksheets
- How to access cells programmatically

Note: To delete a worksheet, you must use either the User Portal or the Folder Manager. For information on the Folder Manager, see “Packaging DeepSee Elements into Classes” in the DeepSee Implementation Guide. For information on the User Portal, see the DeepSee End User Guide.

15.1 Overview of Worksheets

It is often useful to include supplemental data in dashboards, such as, for example, reference values against which to compare measure values. DeepSee provides the ability to create worksheets into which you can type such data. Then you can use these worksheets in two ways:

- As data sources for widgets on a dashboard
- In MDX queries, by using the %KPI function to refer to values in specific cells of the worksheet

15.2 Accessing the Worksheet Builder

To define and modify worksheets, you use the Worksheet Builder. To access this tool in the Management Portal:

1. Click DeepSee.
2. Click Tools.
3. Click **Worksheet Builder**.

The **[DeepSee] > [Worksheet Builder]** page initially looks like this:

![Worksheet Builder](image)

On this page you can do the following:

- Create new worksheets
- Open and modify existing worksheets

### 15.3 Defining a Worksheet

If the Worksheet Builder is currently displaying a worksheet that you do not want to edit, click **New**. Then to define a worksheet, do the following tasks as needed:

- Add rows. To do so, click the Add Row button or the **Add Row** button on the **Details** tab.
  
  The new row is added below your current selection (or to the bottom of the worksheet if nothing is selected).

- Add columns. To do so, click the Add Column button or the **Add Column** button on the **Details** tab.
  
  The new column is added to the left of your current selection (or to the right of the worksheet if nothing is selected).

- Change column and row names. Click a column or a row and then make edits on the **Details** tab.

  For example, to change a row name, click a row and then edit **Row Name** on the **Details** tab:

  ![Row Name](image)

  Then move the cursor to another cell, press **Enter**, or click **OK**.

- Rearrange the worksheet. To do so, use the following icons:
### Specifying Worksheet Options

<table>
<thead>
<tr>
<th>Icon</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>Moves the selected column to the left.</td>
</tr>
<tr>
<td>→</td>
<td>Moves the selected column to the right.</td>
</tr>
<tr>
<td>↑</td>
<td>Moves the selected row up.</td>
</tr>
<tr>
<td>↓</td>
<td>Moves the selected row down.</td>
</tr>
<tr>
<td>✗</td>
<td>Deletes the selected row, column, or value.</td>
</tr>
</tbody>
</table>

- Enter values into cells. To enter a value, click a cell and type a value. Then move the cursor to another cell, press **Enter**, or click **OK**.

  As you type the value, the Worksheet Builder shows the value in a text window at the top:

  ![Text Window](image)

- Specify the numeric formatting for a cell. To do so, click a cell, click the **Details** tab, leave **Cell** selected at the top of that tab, and then specify a format string in **Format**. For information, see “Specifying a Format String” in the chapter “Defining Measures” in Defining DeepSee Models.

  Or, to specify numeric formatting for a column, row, or the whole grid, click the **Details** tab. Then select **Column**, **Row**, or **Grid** at the top of that tab, and then specify a format string in **Format**.

- Specify details for columns and rows. See “Specifying Details for Columns and Rows,” later in this chapter.

Then save the worksheet. To do so, click **Save** or **Save As**; see the next section.

## 15.4 Specifying Worksheet Options

When you click **Save** for the first time, or when you click **Save As**, the Worksheet Builder displays a dialog box in which you can specify the following options for the current worksheet:

- **Folder** — Type a folder name or select an existing folder.
- **Worksheet Name** — Type the name of the worksheet.
- **Public** — Specifies whether this worksheet is displayed in the User Portal main area.

  (Even if the worksheet is not marked as public, you can access it in the Worksheet Builder via **Open**. Also, you can find it in the User Portal by using the Find option. See the chapter “Introduction to the User Portal” in the DeepSee End User Guide.)
Defining Worksheets

- **Locked** — Select this check box to temporarily prevent changes to this worksheet. If you select this option, you must clear the Locked check box before any changes can be made to it.

- **Worksheet Owner** — Optionally specifies the Caché user who owns this worksheet. If a worksheet has an owner, then only the owner can specify the **Access Resource** value for the worksheet; see the next item in this list.

- **Access Resource** — Optionally specifies the Caché resource that is used to control access to this worksheet. See the *DeepSee Implementation Guide*.

- **Description** — Type a description for this worksheet.

- **Keywords** — Type keywords to categorize this worksheet, one keyword (or phrase) per line.

When you are done, click **OK**.

### 15.5 Specifying Details for Columns and Rows

You can specify additional details for columns and rows.

To specify details for a column, click **Column** at the top of the **Details** tab. To specify details for a row, click **Row** at the top of the **Details** tab.

Then you can specify the following details on this tab:

- **Column Name** or **Row Name** — Specify the name for this column or row.

- **Column Expression** or **Row Expression** — Specify the default value for cells in this row or column.

- **Format** — Specify a format string to control numeric formatting for this column or row. For information, see “Specifying a Format String” in the chapter “Defining Measures” in *Defining DeepSee Models*.

  Note that you can also specify numeric formatting for a cell or for the whole grid.

### 15.6 Editing an Existing Worksheet

To edit an existing worksheet:

1. Click **Open**, select the worksheet, and click **OK**.

2. Make any edits as needed. See “Defining Worksheets.”

3. Click **Save** or **Save As**. See “Specifying Worksheet Options.”

### 15.7 Accessing a Worksheet Cell Programmatically

As noted earlier, you can use the MDX `%KPI` function to refer to the value in a specific cell of the worksheet. For convenience, the **KPI** tab of the Worksheet Builder displays the exact syntax needed for the currently selected cell; you can copy and paste this value into your MDX query elsewhere.

For example:
For details on %KPI, see the DeepSee MDX Reference.
This chapter describes how to define quality measures, which are similar to calculated measures. It discusses the following topics:

• Overview
• Introduction to the Quality Measure Manager
• How to create a quality measure
• How to specify the overall expression for a quality measure
• How to edit other parts of the definition
• How to check the expression for a quality measure
• How to delete a quality measure

16.1 Overview of Quality Measures

A quality measure is similar to a calculated measure that can be reused in multiple contexts. A quality measure is defined by a formula that combines MDX expressions. You specify the subject area or subject areas in which it is available, and you can control whether the quality measure is published (and thus available in the Analyzer).

Each quality measure is a Caché class definition, specifically a subclass of %DeepSee.QualityMeasure.QualityMeasure. You can define quality measures either in the Quality Measure Manager or in Studio. This chapter discusses the Quality Measure Manager.

You can use quality measures as follows:

• (When published) They are displayed in the Analyzer, in the Quality Measure section of the left area. You can drag and drop them into pivot tables.

• You can use them in MDX queries. To refer to a quality measure in MDX, use the following syntax:

  [%QualityMeasure].[catalog/set/qm name]

  Where catalog is the catalog to which the quality measure belongs, set is a set in that catalog, and qm name is the short name of the quality measure. (The full name of the quality measure is catalog/set/qm name.)
16.2 Introduction to the Quality Measure Manager

The Quality Measure Manager displays all compile quality measure classes in the given namespace. You can use it to create, modify, and delete them. To access the Quality Measure Manager:

1. In the Management Portal, click DeepSee.
2. Click Tools > Quality Metrics.

This displays the [Caché] > [Quality Measure Manager] page.

Here you can do the following:

- View summary information for the existing quality measures in this namespace. To do so, click Browse.

The middle area then displays something like the following:

![Quality Measure Catalog](image)

- View the definition of a quality measure. To do so, click Open, expand the folders until you find the quality measure, and then click the measure. The middle area then displays the details, which includes a section that shows the definition of the measure. Typically this is as follows:
You can then edit the measure or remove it.

- Create a quality measure. To do so, click **New** and then continue as described in the next section.

## 16.3 Creating a Quality Measure

To create a quality measure:

1. Click **New**.
   
   DeepSee displays a dialog box.

2. Specify the following values, which are all required:
   
   - For **Catalog**, either select an existing catalog or type a new catalog name.
   - For **Set**, either select an existing set or type a new set name.
   - For **Name**, type a name.
   - For **Class Name for the Quality Measure**, type a fully qualified class name (package and class).

3. Click **OK**.
   
   The middle area of the page now displays the initial definition of the quality measure, as follows:
Every quality measure is expressed as a formula as shown here in the Measure field. In this case, the formula is:

\[ \frac{\text{Numerator}}{\text{Denominator}} \]

Where Numerator and Denominator are the names of groups defined in this quality measure.

Each group consists of one or more items, each of which is defined by an MDX expression.

In the initial definition of a quality measure, each of these groups is defined as a constant (100). The initial value of this quality measure is 1.

The bottom area of the page displays any additional information:

4. Click Edit.

Now you can modify the formula and make other changes. See the following sections.

5. When you are done, click either Save to save the definition or click Save As to save it with a new name.

When you save the definition, DeepSee automatically compiles the class and writes the quality measure definition into a system global.

### 16.4 Specifying the Expression for a Quality Measure

When you display a quality measure in edit mode, you can edit the overall expression, which looks like this for a new quality measure:
Here you can do the following:

• Change the overall expression. To do so, click the edit button. Then specify a new expression and click OK. In your formula:
  – Use square brackets around the name of any group that is used in the formula. For example, use \([\text{Numerator}]\) to refer to the Numerator group, if your quality measure defines such a group.
  – Use any of the standard mathematical operators.
  – Include numeric constants as needed.
  – Include parentheses to control precedence as needed.

• Add a group. To do so, click the add button next to the Measure heading. Then edit the new group, which is added to the end of definition.

• Edit a group. To do so, click the edit button. Then specify new values for Name and Description as needed, and click OK.

• Add an element to a group. To do so, click the add button next to the group name. Then edit the new element.

• Edit a element in a group. To do so, click the edit button next to the element name. Then specify new values for Name and MDX Expression as needed, and click OK.
For information on creating MDX expressions, see *Using MDX with DeepSee* and the *DeepSee MDX Reference*.

- Change the order of groups within the measure definition or change the order of elements within a group. To do so, click the up button or down button next to the item that you want to move.

- Delete a group or an element. To do so, click the delete button next to the item you want to delete. DeepSee prompts you to confirm this action.

### 16.4.1 How Groups and Elements Are Combined

The formula for the quality measure (shown below the *Measure* heading) determines how the groups are combined. Within any given group, the elements are combined as an MDX set (that is, they are combined via logical OR). If you need to combine elements via logical AND, create one element that has an MDX tuple expression that combines the desired pieces. For example:

```
([patgrp].[h1].[medicaid].[yes],[age].[h1].[0 to 2])
```

To compute the value for a quality measure in any given context, DeepSee does the following:

1. For each group, creates an MDX set expression that combines the elements in that group.
2. Evaluates each group and determines its value.
3. Combines the group values as given in the formula.

### 16.5 Editing Other Information for a Quality Measure

When you display a quality measure in edit mode, you can also edit *Caption* (at the top of the page), as well as the following area at the bottom of the page:
Here you can do the following:

- Edit an item. To do so, click the edit button next to that item and then specify the details.
- Add an item. To do so, click Add Meta Item.

The editable items here are as follows:

- **Subject Area** — Specifies the cube or subject area in which this quality measure is available.
  To specify multiple cubes or subject areas, edit the quality measure class in Studio.
  Note that the quality measure is automatically available in any subject areas that are based on any cubes in this list.

- **Published** — Specifies whether this quality measure is available in the Analyzer.

- **Link to additional info** — Specifies a link to a URL that contains any additional information.

- **Your custom items** — Specify any additional information to display in the Quality Measure Manager. DeepSee does not use these items.

### 16.6 Checking the Expression for a Quality Measure

You might want to check the overall expression that defines a quality measure, particularly if the definition is complex. To do so, use the `%GetExpression` instance method of the quality measure. For example:

```
SAMPLES> set qm=##class(QM.Preventive.Child.QM7).%New()
SAMPLES> w qm.%GetExpression()
[tests].[h1].[leadscr].[yes]/([patgrp].[h1].[medicaid].[yes],[age].[h1].[0 to 2])
```

This is the expression for the following quality measure:

```
Measure
[Numerator]/[Denominator]

Numerator
Numerator

* Numerator Cohort
  [tests].[h1].[leadscr].[yes]

Denominator
Denominator

* Denominator Cohort
  ([patgrp].[h1].[medicaid].[yes],[age].[h1].[0 to 2])
```

### 16.7 Deleting a Quality Measure

To delete a quality measure:

1. Display its definition.
2. Click **Remove**.

3. Click **OK**.
17

Defining Basic KPIs

This chapter introduces key performance indicators (KPIs) in DeepSee and describes how to define KPIs that use hardcoded queries. It discusses the following topics:

- Introduction
- How to choose between MDX and SQL
- Structure of a KPI result set
- How to define a KPI with a hardcoded query
- Class parameters for the KPI class
- How to specify range and threshold values to affect speedometers
- How to disable the %CONTEXT filter

Also see the chapters “Defining KPIs with Filters and Listings” and “Defining Advanced KPIs.”

For information on defining iKnow KPIs, see “iKnow KPIs and DeepSee Dashboards” in Using iKnow.

17.1 Introduction to KPIs

A KPI is a class based on %DeepSee.KPI. In most cases, a KPI uses a query and displays a result set. (In other cases, a KPI only defines actions; see “Defining Custom Actions” in the DeepSee Implementation Guide).

Like pivot tables, KPIs can be displayed on a dashboard, within a widget.

You can also use the MDX %KPI function to retrieve values for a KPI.

17.1.1 Comparison to Pivot Tables

KPIs are similar to pivot tables in many ways, but provide additional options that are not available for pivot tables. One difference is that a KPI can use an SQL query; this is important because SQL queries and MDX queries are suitable in different scenarios. In some cases, an SQL query is more efficient, and in such cases, you should use an SQL query within a KPI.

The following table shows additional differences and similarities between KPIs and pivot tables:
Defining Basic KPIs

<table>
<thead>
<tr>
<th>Feature</th>
<th>KPIs</th>
<th>Pivot Tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be based on an MDX query</td>
<td>Yes, but see the next subsection</td>
<td>Yes</td>
</tr>
<tr>
<td>Can be based on an SQL query</td>
<td>Yes, but see the next subsection</td>
<td>No</td>
</tr>
<tr>
<td>Query can be determined at runtime</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can be displayed in a scorecard widget, meter widget, or pivot table widget</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can include filter controls</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Member list in filter can be customized</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can define actions (see “Defining Custom Actions” in the DeepSee Implementation Guide)</td>
<td>Yes</td>
<td>No, but can use actions defined for cube class</td>
</tr>
<tr>
<td>Can define range and threshold values</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can be secured via a Caché resource</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can define custom callbacks</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can display listings</td>
<td>Yes, if you define a custom listing</td>
<td>Yes</td>
</tr>
<tr>
<td>Can be exported to Microsoft Excel</td>
<td>Yes, if displayed in a pivot table widget</td>
<td>Yes</td>
</tr>
<tr>
<td>Can be displayed in chart format</td>
<td>Yes, if displayed in a pivot table widget</td>
<td>Yes</td>
</tr>
<tr>
<td>Can access Mini Analyzer</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

17.1.2 Requirements for KPI Queries

In most cases, a KPI uses either an MDX query or an SQL query. There are rules about the form of the query; these rules are imposed by the structure of a KPI result set (discussed in a later section).

- If the query uses MDX, note the following requirements:
  - The query must use members for rows. You can have nested rows
  - The query must use measures for columns.
  - The query cannot include nesting for columns.
- The query must return numeric values.
- If the query returns more than 1000 rows, only the first 1000 rows are used.

You can use queries that do not follow these rules. To do so, you must parse the result set and directly specify properties of the KPI instance. For information, see the chapter “Defining Advanced KPIs.”

Also note that if you display the KPI in a speedometer, only the first row is used.
17.2 Choosing Between MDX and SQL

SQL queries and MDX queries are suitable in different scenarios, and in some cases, an SQL query is more efficient. MDX is generally more suitable when you are aggregating across large numbers of records. In contrast, when you are not aggregating at all, or when you are aggregating only at a low level, SQL performs better. For example, consider the following pivot table:

<table>
<thead>
<tr>
<th>PatientID</th>
<th>Patient Count</th>
<th>Age</th>
<th>Allergy Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBJ_100301</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SUBJ_100302</td>
<td>1</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>SUBJ_100303</td>
<td>1</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>SUBJ_100304</td>
<td>1</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>SUBJ_100305</td>
<td>1</td>
<td>66</td>
<td>1</td>
</tr>
<tr>
<td>SUBJ_100306</td>
<td>1</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>SUBJ_100307</td>
<td>1</td>
<td>66</td>
<td>1</td>
</tr>
</tbody>
</table>

In this pivot table, each row represents one row in the source table. The equivalent SQL query would be faster.

17.3 Structure of a KPI Result Set

The result set of a KPI is organized into series and properties.

A *KPI series* is a row. The following example shows nine series (displayed on the KPI test page, introduced later in this chapter). Each series has a name, which is shown here in the first column.

<table>
<thead>
<tr>
<th>KPI Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series</td>
</tr>
<tr>
<td>Cedar Falls</td>
</tr>
<tr>
<td>Centerville</td>
</tr>
<tr>
<td>Cypress</td>
</tr>
<tr>
<td>Elm Heights</td>
</tr>
<tr>
<td>Juniper</td>
</tr>
<tr>
<td>Magnolia</td>
</tr>
<tr>
<td>Pine</td>
</tr>
<tr>
<td>Redwood</td>
</tr>
<tr>
<td>Spruce</td>
</tr>
</tbody>
</table>

A *KPI property* is a data column. The previous example shows a KPI with two properties.

For KPIs based on MDX queries, a series often corresponds to a member of a level, and a property often corresponds to a measure.
17.4 Defining a KPI with a Hardcoded Query

To create a simple KPI that uses a hardcoded query, do the following in Studio:

1. Click **File > New**, click the **Custom** tab, and then click **New DeepSee KPI**.

2. Specify the following required values:
   - **Package Name** — Package to contain the KPI class.
   - **Class Name** — Short class name of the KPI class.

3. Optionally specify the following additional values:
   - **KPI Caption** — Not used.
   - **KPI Name** — Logical name of the KPI.
   - **Description** — Description of the KPI, to be saved as comment lines for the class.
   - **Source Type** — Specifies the source of the data for this KPI. Select either **mdx** or **sql**. (For information on **manual**, see the chapter “Defining Advanced KPIs.”)
   - **Properties** — Type the names of the properties of this KPI (the column names of the result set). Type each property on a separate line.
   - **Filters** — Type the names of any filters to be used in the KPI query. See the next chapter. Type each filter name on a separate line.
   - **Actions** — Type the names of any actions to be defined in the KPI. See “Defining Custom Actions” in the DeepSee Implementation Guide. Type each action name on a separate line.

   You can edit all these values later as well.

4. Click **Finish**.

   The wizard generates a class definition like this:

   ```java
   Class MyApp.KPI.MyKPI Extends %DeepSee.KPI {
     // This XData definition defines the KPI.
     XData KPI [ XMLNamespace = "http://www.intersystems.com/deepsee/kpi" ]
     {
       <kpi xmlns="http://www.intersystems.com/deepsee/kpi"
            name="MyKPI" sourceType="mdx"
            caption="MyCaption"
       >
         <property name="PatCount" displayName="PatCount"/>
         <property name="AvgAge" displayName="AvgAge"/>
       </kpi>
     }
   }
   ```

   The XData block defines the KPI. In the XData block, `<kpi>` is an XML element. This element starts with the opening `<kpi>` and ends with the closing angle bracket. `xmlns`, `name`, `sourceType`, and `caption` are XML attributes. Each attribute has a value. In this example, the value of the `sourceType` attribute is **mdx**.

   The class also includes stub definitions for several methods; by default, these do nothing. For details, see the following two chapters.

5. Within the `<kpi>` element, add one of the following attribute specifications:

   ```java
   mdx="MDX query"
   ```

   Or:
Where **MDX query** is an MDX SELECT query or **SQL query** is an SQL SELECT query. (Use the `mdx` option if you chose `mdx` in the wizard, and use the `sql` option if you chose `sql`.)

For example:

```xml
<kpi xmlns="http://www.intersystems.com/deepsee/kpi"
     name="MyKPI" sourceType="mdx"
     mdx="SELECT {MEASURES.[%COUNT],MEASURES.[Avg Age]} ON 0, HomeD.H1.City.MEMBERS ON 1 FROM patients"
     caption="MyCaption"/>
```

You can add the attribute specification anywhere between the opening `<kpi>` and the closing angle bracket. The attribute specification can be on its own line, as shown here, or it can be on the same line as other attributes. Within the XData block, Studio provides assistance as you type.

For requirements, see “**Requirements for KPI Queries,**” earlier in this chapter.

For information on MDX, see *Using MDX with DeepSee* and *DeepSee MDX Reference*.

6. Edit each `<property>` element to include the `columnNo` attribute.

This attribute specifies the column of the result set to which that property matches. For example, the preceding MDX query provides the patient count as the first data column and the average age as the second column. So we edit the `<property>` elements as follows:

```xml
<property name="PatCount" displayName="PatCount" columnNo="1"/>
<property name="AvgAge" displayName="AvgAge" columnNo="2"/>
```

For details on specifying the default appearance of each property, see “**<property>,**” in the appendix “**Reference Information for KPI and Plugin Classes,**”

7. Optionally specify class parameters, as described in the next section.

8. Compile the class.

9. Click **View > Web Page.**

You then see something like the following:
The **Series** column indicates the name of each series. This name is available as a label when you display this KPI in a scorecard.

To the right of those columns, this table has one column for each `<property>` of the KPI. This column shows the current value of that property, for each row in the KPI.

### 17.5 Specifying Class Parameters

You can specify some or all of the following class parameters in your KPI class:

**RESOURCE**

Parameter RESOURCE = "MyKPI";

Specifies the resource that secures this KPI; for details, see the [DeepSee Implementation Guide](#).
**DOMAIN**

Parameter DOMAIN = "MyDeepSeeModel";

Specifies the localization domain to which this KPI belongs; for details, see the DeepSee Implementation Guide.

**PUBLIC**

Controls whether the KPI is available for use in scorecards and other dashboard widgets, as well as for use with the MDX %KPI function. If you want to hide the KPI from use in dashboards, add the PUBLIC class parameter to the class, with the value 0.

**LABELCONCAT**

Specifies the character used to concatenate labels for an MDX-based KPI that uses CROSSJOIN or NONEMPTYCROSSJOIN for rows. The default is a slash (/).

For the ASYNCH class parameter, see the chapter “Defining Advanced KPIs.”

### 17.6 Specifying Ranges and Thresholds for Speedometers

Within the definition of a KPI, you can specify its range and threshold values, for use in speedometers. To specify these values, edit the <kpi> element and specify the following attributes:

- **rangeLower** — Default lowest value to display in the meter.
- **rangeUpper** — Default highest value to display in the meter.
- **thresholdLower** — Default lower end of the threshold for this KPI. The threshold area is displayed in contrasting color.
- **thresholdUpper** — Default higher end of the threshold.

For example:

```xml
<kpi name="KPIForRangeDemos" sourceType="mdx"
    mdx='SELECT MEASURES.[%COUNT] ON 0, AgeD.[All Patients] ON 1 FROM PATIENTS'
    rangeLower="0"
    rangeUpper="900"
    thresholdLower="20"
    thresholdUpper="800"
>
    <property name="Patient Count" columnNo="1" />
</kpi>
```

When displayed in a speedometer, this KPI looks as follows (by default):
Notice the value box in the speedometer displays the actual KPI value (1000), even though it is higher than the value of rangeUpper.

You can also set the range and threshold values programmatically, which is useful if hardcoded values are not appropriate. See the chapter “Defining Advanced KPIs.”

**Note:** When you configure a scorecard on a dashboard, you have the options Lower Threshold, Upper Threshold, Lower Range, and Upper Range. Note that the KPI attributes rangeLower, rangeUpper, thresholdLower, and thresholdUpper do not affect these scorecard options.

### 17.7 Disabling the %CONTEXT Filter

As noted earlier, you can use the MDX %KPI function to retrieve values for a KPI. For MDX-based KPIs, the %KPI function has an optional parameter (%CONTEXT) that passes context information to the KPI. By default, this context information is applied to the MDX query as a filter clause. To disable this automatic behavior, override the %GetMDXContextFilter() method as follows:

```diml
Method %GetMDXContextFilter() As %String
{
    Quit ""
}
```
18

Defining KPIs with Filters and Listings

This chapter describes options for defining KPIs that include filters and listings.

- Introduction to KPI filters
- How to make filters interoperable between KPIs and pivot tables
- How to define a KPI filter
- How to override a KPI query at runtime
- How to access the filter values
- How to combine filter values in an MDX query
- MDX KPI examples
- SQL KPI examples
- How to define a listing for a KPI

Also see the chapter “Defining Basic KPIs.” For more advanced KPIs, see the chapter “Defining Advanced KPIs.”

KPIs can also define actions. See “Defining Custom Actions” in the DeepSee Implementation Guide.

18.1 Introduction to Filters

Typical dashboards include one or more filter controls, with which the user interacts with the dashboard. Each filter control is typically displayed as a drop-down list.

For a pivot table, any level in the cube or subject area can be used as filter. For a KPI, however, no filters are defined by default. To add filters to a KPI, you use the following system:

- You define each filter; that is, you specify the filter names and the items in the filter lists.
  
  You can skip this step if you intend to use filters provided by a pivot table.

- You specify the KPI query programmatically so that it uses the filters. To do this, you implement a callback method in the KPI so that it overrides any hardcoded query. Within this method, you have access to the selected filter values.

  In typical cases, you add a WHERE clause that restricts the records used by the query. Then when the user refreshes the dashboard, DeepSee reruns the query and displays the results.

  Because you construct the KPI query yourself, you can use filters in more general ways. That is, you are not required to use the filter selection to affect the WHERE clause of the query. For an extreme example, a filter could display the
options 1, 2, and 3. Then the KPI could execute version 1, 2, or 3 of the query, which might be entirely different versions. For another example, a filter could display a list of measure names, and the KPI could include the selected measures in its query.

18.2 Creating Interoperable Filters

A dashboard can contain both KPIs and pivot tables, displayed in scorecards, pivot table widgets, or other widgets. Within a dashboard, a filter can be configured to affect multiple widgets.

It is possible to define filters that affect both KPIs and pivot tables.

Pivot tables send and receive filter values in a specific syntax, described in the following subsection. To create interoperable filters, you must consider this requirement; the second subsection describes two possible approaches.

18.2.1 Filter Syntax for Pivot Tables

The following table indicates the filter syntax used and expected by pivot tables. This syntax applies to pivot tables that are displayed in any kind of dashboard widget.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Filter Name</strong></td>
<td>All</td>
<td>Complete MDX level expression</td>
</tr>
<tr>
<td><strong>Filter Value</strong></td>
<td>User selects a single member</td>
<td>MDX key for a member of that level, which is an expression of the following form: &amp;[keyval] where keyval is the key value for the member.</td>
</tr>
<tr>
<td></td>
<td>User selects a range of members</td>
<td>Expression of the following form: &amp;[keyval1]:&amp;[keyval2]</td>
</tr>
<tr>
<td></td>
<td>User selects multiple members</td>
<td>Expression of the following form (for example): {&amp;[keyval1],&amp;[keyval2]}</td>
</tr>
<tr>
<td></td>
<td>User selects a single member and clicks Exclude</td>
<td>Expression of the following form: %NOT&amp;[keyval1]</td>
</tr>
<tr>
<td></td>
<td>User selects multiple members and clicks Exclude</td>
<td>Expression of the following form (for example): %NOT{&amp;[keyval1],&amp;[keyval2]}</td>
</tr>
</tbody>
</table>

Other than the optional %NOT string, the filter names and filter values are not case-sensitive.

18.2.2 Ways to Create Interoperable Filters

There are two general ways to create filters that can affect both pivot tables and KPIs:

- You can create a KPI that defines and uses filters in the format required by pivot tables.
• You can define a KPI that converts filter names and values from the pivot table format to the format needed for the KPI query.

These approaches can work with either MDX-based or SQL-based KPIs, but more work is needed to convert values when you use an SQL-based KPI.

For an MDX example, see the dashboard Demo Filter Interoperability in the SAMPLES namespace. This dashboard demonstrates the former approach. This dashboard displays two pivot table widgets; the upper one displays a pivot table, and the lower one displays a KPI. For this dashboard, the Filters worklist on the left includes the following filters:

- The Favorite Color filter is configured as part of the upper widget and is therefore defined by the pivot table that is displayed in that widget.

  Favorite Color is a level in the cube on which this pivot table is based.

- The Doctor Group filter is configured as part of the lower widget and is therefore defined by the KPI that is displayed in that widget.

  This filter is defined programmatically within this KPI. This filter is defined to use values in pivot table format, as given in the previous section.

For both of these filters, the target of this filter is * (which refers to all widgets in this dashboard). As you can see by experimentation, both filters affect both of these widgets.

The KPI includes an additional filter (Yaxis), which controls the y-axis of the MDX query used by the KPI. This filter has no effect on the pivot table.

The sections “MDX Example 2” and “MDX Example 3,” later in this chapter, show how this KPI works.

### 18.3 Defining KPI Filters

To define a filter in a KPI, do both of the following:

- Declare the name of the filter.
- Define the list of elements in the filter.

Then modify the query to use the value or values selected by the user; see the later sections of this chapter.

#### 18.3.1 Declaring the Filters in a KPI

To declare the filters in a KPI, do either or both of the following:

- Specify the `<filter>` element of the `<kpi>` element. For example:

  ```xml
  <kpi name="sample KPI"...>
  <filter name="filter1" displayName="filter1"/>
  <filter name="filter2" displayName="filter2"/>
  <filter name="filter3" displayName="filter3"/>
  ...
  
  By default, users can select multiple elements from a filter list. To require users to select only one element from a filter list, add `multiSelect="false"` to the `<filter>` element. For example:

  ```xml
  <filter name="filter3" displayName="filter3" multiSelect="false"/>
  
  By default, the filter is displayed as a drop-down list. If the filter items are days, you might prefer to display a calendar control. If so, add `searchType="day"` to the `<filter>` element. For example:

  ```xml
  <filter name="filter3" displayName="filter3" multiSelect="false" searchType="day"/>
  ```
A filter can depend upon another filter. To indicate this, use the optional dependsOn attribute of <filter>. For example:

```xml
<filter name="State"/>
<filter name="City" dependsOn="State"/>
```

- Override the `%OnGetFilterList()` callback of your KPI class. This method has the following signature:

```xml
classmethod %OnGetFilterList(Output pFilters As %List,pDataSourceName As %String = "") As %Status
```

Where `pFilters` is an array with the following nodes:

<table>
<thead>
<tr>
<th>Node</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pFilters</td>
<td>Number of filters</td>
</tr>
<tr>
<td>pFilters(n)</td>
<td>Details for the n-th filter. This is a $LISTBUILD list that consists of the following items:</td>
</tr>
<tr>
<td></td>
<td>– A string that equals the logical filter name</td>
</tr>
<tr>
<td></td>
<td>– A string that equals the filter display name</td>
</tr>
<tr>
<td></td>
<td>– A string that equals the filter property; the default is the logical filter name</td>
</tr>
<tr>
<td></td>
<td>– 1 or 0 to indicate whether multiselect is enabled for this filter. Use 1 to enable multiselect or 0 to disable it.</td>
</tr>
</tbody>
</table>

And `pDataSourceName` is for future use.

For example, the following implementation adds a filter named `New Filter`:

```xml
ClassMethod %OnGetFilterList(Output pFilters As %List,pDataSourceName As %String = "") As %Status {
    set newfilter=$LB("New Filter","New Filter Display Name",,0)
    set pFilters($I(pFilters))=newfilter
}
```

### 18.3.2 Specifying the List Items in a KPI Filter

There are several ways you can specify the items in a filter.

**Note:** Note that for an MDX-based KPI, it is useful to ensure that the filter items are member keys; otherwise, more work is needed to construct filter clauses, as described later in this chapter.

To specify the list items in a filter, do any of the following:

- **Specify the valueList attribute of the <filter> element.** If specified, this must be a comma-separated list of values for this filter, in the order in which they are to be displayed.

  For example:

  ```xml
  <filter name="ZipCode" valueList="&[36711],&[34577],&[38928],&[32006],&[32007]" displayList="36711,34577,38928,32006,32007"/>
  ```

  This attribute takes precedence over the `sql` attribute.
If you specify this attribute, you can also specify the `displayList` attribute, as shown above. If specified, this must be a comma-separated list of display values for this filter, in the order in which they are displayed in any controls. If you do not specify this attribute, the filter displays the values given by `valueList`.

- **Override the `%OnGetFilterMembers()` callback of your KPI class.** This method has the following signature:

```plaintext
classmethod %OnGetFilterMembers(pFilter As %String, Output pMembers As %List, pSearchKey As %String = "") as %Status
```

Where `pFilter` is the logical name of a filter, as given in the `<filter>` element, `pMembers` specifies the members, and `pSearchKey` is the search key entered by the user, if applicable (`pSearchKey` is used by the search box control).

`pMembers` has the form:

```plaintext
pMembers($I(pMembers))=$LB(displayValue,actualValue)
```

For example, you could define the `AgeGroup` filter as follows:

```xml
<filter name="AgeGroup" />
```

Then, in the same class define the `%OnGetFilterMembers()` callback as follows:

```plaintext
ClassMethod %OnGetFilterMembers(pFilter As %String, Output pMembers As %List) As %Status
{
    Set status = $$$OK
    Try {
        If (pFilter = "AgeGroup") {
            //get values from level table
            Set sql = "SELECT DISTINCT DxAgeGroup FROM DeepSee_Model_PatientsCube.DxAgeGroup"
            Set st = #class(%SQL.Statement).%New()
            Set status = st.%Prepare(sql)
            If $$$ISERR(status) {Do $system.Status.DisplayError(status) Quit}
            Set rs = st.%Execute()
            While(rs.%Next(.status)) {
                If $$$ISERR(status) {Do $system.Status.DisplayError(status) Quit}
                Set display=rs.DxAgeGroup
                Set actual="&[_display_]"
                Set pMembers($I(pMembers)) = $LB(display,actual)
            }
            If $$$ISERR(status) {Do $system.Status.DisplayError(status) Quit}
        }
    }
    Catch(ex) {
        Set status = ex.AsStatus()
    }
    Quit status
}
```

The following **subsection** describes another way to obtain the list of members.

- **Specify the `sql` attribute of the `<filter>` element.** If specified, this must be an SQL query. The query can return either one or two columns. The first column in the returned dataset must provide the values for the filter. You can include a second column, which would provide the corresponding display values. If you do not, the filter displays the values given by the first column.

For example:

```xml
<filter name="ZipCode1" sql="SELECT DISTINCT PostalCode FROM DeepSee_Study.City"/>
```

If you specify the `sql` attribute, note that `displayList` is ignored.

**Note:** In this case, the filter items are not member keys. Thus, this technique is less useful for MDX-based KPIs.
18.3.2.1 Obtaining Level Members

To obtain the level members, you can use the `%GetMembersForFilter()` method of your KPI class. This method has the following signature:

```plaintext
classmethod %GetMembersForFilter(pCube As %String, pFilterSpec As %String, Output pMembers, pSearchKey As %String = "") as %Status
```

`pCube` is the logical name of a DeepSee cube, and `pFilterSpec` is the complete name of a level in that cube (for example, "[DateOfSale].[Actual].[YearSold]").

`pMembers`, which is returned as output, is a list of members in the form `pMembers(n) = $LB(text, value)`

For any given member, `value` is the member name, and `value` is the member key, for example, &[1998]

For example:

```plaintext
ClassMethod %OnGetFilterMembers(pFilter As %String, Output pMembers As %List) As %Status
{
    Set status = $$$OK
    Set pMembers=""
    Try {
        If (pFilter = "Year") {
            set status=..%GetMembersForFilter("Patients","[BirthD].[H1].[Year]",.pMembers)
        }
        If (pFilter = "ZipCode") {
            set status=..%GetMembersForFilter("Patients","[HomeD].[H1].[ZIP]",.pMembers)
        }
    }
    Catch(ex) {
        Set status = ex.AsStatus()
    }
    Quit status
}
```

Or you can execute a query on the dimension table that holds the level members. See the appendix “Details for the Fact and Dimension Tables.”

18.4 Overriding a Query at Runtime

This section describes generally how to override a KPI at runtime. Use this information when you define filters in a KPI or when you want to override the KPI query for other reasons.

The details are different for MDX and SQL.

For information on accessing the filter values, see the next section.

18.4.1 Overriding an MDX Query

If you use the KPI wizard in Studio and you choose the source type as MDX, the generated KPI class includes a stub definition for the `%OnGetMDX()` method:

```plaintext
/// Return an MDX statement to execute.
Method %OnGetMDX(ByRef pMDX As %String) As %Status
{
    Quit $$$OK
}
```
To implement this method, add your code to method definition. Your code can use the variable \texttt{pMDX}, which contains the MDX query specified in the \texttt{mdx} attribute of \texttt{<kpi>}. If you did not specify that attribute, \texttt{pMDX} is null.

Your code should set \texttt{pMDX} equal to an MDX SELECT query and should return \texttt{$$\$$OK} (or 1).

For information on MDX, see \textit{Using MDX with DeepSee} and \textit{DeepSee MDX Reference}. In particular, note that the \texttt{WHERE} clause uses different syntax in MDX than it does in SQL.

For examples, see “MDX KPI Examples,” later in this chapter.

### 18.4.2 Overriding an SQL Query

If you use the KPI wizard in Studio and you choose the source type as SQL, the generated KPI class includes a stub definition for the \texttt{%OnGetSQL()} method:

```cpp
/// Return an SQL statement to execute.
Method %OnGetSQL(ByRef pMDX As %String) As %Status
{
    Quit $$\$$OK
}
```

To implement this method, add your code to method definition. Your code can use the variable \texttt{pSQL}, which contains the SQL query specified in the \texttt{sql} attribute of \texttt{<kpi>}. If you did not specify that attribute, \texttt{pSQL} is null.

Your code should set \texttt{pSQL} equal to an SQL SELECT query and should return \texttt{$$\$$OK} (or 1).

An SQL-based KPI cannot have more than 1000 rows; the system automatically limits the number of rows returned.

For examples, see “SQL KPI Examples,” later in this chapter.

### 18.5 Accessing Filter Values

This section describes how to access the values of the KPI filters. Use this information when you override a KPI query at runtime, as described in the previous section. For information on combining filter values, see the next section.

For examples, see “MDX KPI Examples” and “SQL KPI Examples,” later in this chapter.

#### 18.5.1 Property to Access Filter Values

When a user selects one or more filter items, the system sets the \texttt{%filterValues} property of your KPI instance. This property is an instance of \texttt{%ZEN.proxyObject} that the system creates as follows:

- It has one property for each filter in the KPI, with the same name (case-sensitive) as the filter.

  For example, if a KPI has filters named \texttt{filter1}, \texttt{filter2}, and \texttt{filter3}, then the \texttt{%filterValues} property of your KPI instance has properties named \texttt{filter1}, \texttt{filter2}, and \texttt{filter3}.

As of release 12.2, Caché supports quoted property names, which means that \texttt{%ZEN.proxyObject} can have properties with names that do not follow the rules for standard class member names. For example, if your KPI has a filter named \texttt{[HomeD].[H1].[City]}, the \texttt{%filterValues} property of your KPI instance has a \texttt{"[HomeD].[H1].[City]"} property. If you follow the guidelines for creating filters that affect both KPIs and pivot tables, your filters do have names of this form; see “Creating Interoperable Filters,” earlier in this chapter.

- Each of these properties of \texttt{%filterValues} contains the value or values selected by the user for the corresponding filter, as follows:
### 18.5.2 Methods to Access Filter Values

To access filter values, you can also use methods of your KPI instance. There are two instance methods that retrieve values in formats that are useful for including in your query:

**%GetMDXForFilter()**

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Value of %filterValues.FilterName</th>
</tr>
</thead>
<tbody>
<tr>
<td>User selects <em>item</em></td>
<td><em>item</em></td>
</tr>
<tr>
<td>User selects a range, starting with <em>item1</em> and ending with <em>item2</em></td>
<td><em>item1</em>: <em>item2</em></td>
</tr>
<tr>
<td>User selects <em>item1</em> and <em>item2</em></td>
<td>{ <em>item1</em>, <em>item2</em> }</td>
</tr>
<tr>
<td>User selects <em>item</em> and clicks <em>Exclude</em></td>
<td>%NOT <em>item</em></td>
</tr>
<tr>
<td>User selects <em>item1</em> and <em>item2</em> and clicks <em>Exclude</em></td>
<td>%NOT { <em>item1</em>, <em>item2</em> }</td>
</tr>
</tbody>
</table>

For example, within %OnGetMDX() or %OnGetSQL(), you can include logic like the following:

```csharp
If (..%filterValues.Gender='''')
    { //your logic here }
```

Or, if you are defining interoperable filters:

```csharp
If (..%filterValues."[GenD].[H1].[Gender]='''")
    { //your logic here }
```

#### 18.5.2 Methods to Access Filter Values

To access filter values, you can also use methods of your KPI instance. There are two instance methods that retrieve values in formats that are useful for including in your query:

**%GetMDXForFilter()**

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>User selects a single item</td>
<td>MDX_level_expression.selected_item</td>
</tr>
<tr>
<td>User selects multiple items</td>
<td>{MDX_level_expression.selected_item1, MDX_level_expression.selected_item2}</td>
</tr>
<tr>
<td>User selects a single item and clicks <em>Exclude</em></td>
<td>MDX_level_expression.%NOT selected_item</td>
</tr>
<tr>
<td>User selects multiple items and clicks <em>Exclude</em></td>
<td>MDX_level_expression.%NOT { selected_item1, selected_item2 }</td>
</tr>
</tbody>
</table>

Note that the returned value is not always a valid MDX expression, despite the name of this method. The purpose of this method is to return strings in a form accepted by pivot tables. Compare these results to those given in “Filter Syntax for Pivot Tables,” earlier in this chapter.
%GetSQLForFilter()

method %GetSQLForFilter(sql_field_expression, filter_name) As %String

Examines the current filter selections and returns a string that you can use in the WHERE clause of an SQL query. sql_field_expression is an SQL field name and can include Caché arrow syntax. filter_name is the name of a filter defined in this KPI.

For example, consider the following method call:

..%GetSQLForFilter("City->Name","City")

The following table shows the values returned by this method call, in different scenarios.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Value Returned by Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>User selects PINE</td>
<td>City-&gt;Name = 'PINE'</td>
</tr>
<tr>
<td>User selects a range, starting with MAGNOLIA and ending with PINE</td>
<td>City-&gt;Name = ('MAGNOLIA':'PINE')</td>
</tr>
<tr>
<td>User selects MAGNOLIA and PINE</td>
<td>City-&gt;Name IN ('MAGNOLIA','PINE')</td>
</tr>
<tr>
<td>User selects PINE and clicks Exclude</td>
<td>City-&gt;Name &lt;&gt; 'PINE'</td>
</tr>
<tr>
<td>User selects MAGNOLIA and PINE and clicks Exclude</td>
<td>City-&gt;Name NOT IN ('MAGNOLIA','PINE')</td>
</tr>
</tbody>
</table>

18.6 Combining Filter Values in an MDX Query

This section discusses the general logic for combining filter values in MDX. This applies to the case when you intend to filter the data (as opposed to using filters in a more exotic way, such as controlling the axes of the query).

Each filter provides zero or more values. When a filter provides no values, ignore that filter. Otherwise, create an MDX set expression as follows:

<table>
<thead>
<tr>
<th>User Selection in Filter</th>
<th>Appropriate MDX Set Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>One item</td>
<td>[dimension].[hierarchy].[level].[member]</td>
</tr>
<tr>
<td>One item and the Exclude option</td>
<td>[dimension].[hierarchy].[level].[member].%NOT</td>
</tr>
<tr>
<td>Multiple items</td>
<td>{item,item,…} where each item has the form [dimension].[hierarchy].[level].[member]</td>
</tr>
<tr>
<td>Multiple items and the Exclude option</td>
<td>EXCEPT([dimension].[hierarchy].[level].MEMBERS,{item,item,…}) where each item has the form [dimension].[hierarchy].[level].[member]</td>
</tr>
<tr>
<td>A range</td>
<td>[dimension].[hierarchy].[level].[member][member]</td>
</tr>
</tbody>
</table>

For an example, see the method BuildString() in the sample class DeepSee.Utils.MDXAutoFiltersKPI.

Then add one or more %FILTER clauses to the query:

- If there is only one filter with a non-null value, the %FILTER clause should be:

  %FILTER set_expression
• If there are two filters with a non-null value, the %FILTER clause should be:
  %FILTER set_expression1 %FILTER set_expression2

• If there are three filters with a non-null value, the %FILTER clause should be:
  %FILTER set_expression1 %FILTER set_expression2 %FILTER set_expression3

And so on.

Or create a MDX tuple expression that combines the sets.

## 18.7 MDX KPI Examples

### 18.7.1 MDX KPI Example 1

This section shows an example of %OnGetMDX() that defines a query that uses filter values. This example is from DeepSee.Model.KPIs.DemoMDX:

```mdx
Method %OnGetMDX(ByRef pMDX As %String) As %Status
{
Set pMDX = "SELECT {[MEASURES].[%COUNT],MEASURES.[%COUNT],MEASURES.[avg test score]} ON 0," _
  "NON EMPTY [homed].[h1].[city].MEMBERS ON 1 FROM [patients] "
Set filtercount=0
Set filterlist=#class(%ListOfDataTypes).%New() ; will be a list of strings
//look at %filterValues to see if a filter has been applied to this KPI instance
If (..%filterValues.ZipCode'=""
{
  Set filtercount=filtercount+1
  Do filterlist.Insert("[homed].[h1].[zip]."_..%filterValues.ZipCode_")
}
If (..%filterValues.Gender'=""
{
  Set filtercount=filtercount+1
  Do filterlist.Insert("[gend].[h1].[gender]."_..%filterValues.Gender_")
}
If (..%filterValues.AgeGroup'=""
{
  Set filtercount=filtercount+1
  Do filterlist.Insert("[aged].[h1].[age group]."_..%filterValues.AgeGroup_")
}
//if there are no filters, quit because we have the query we want
If (filtercount=0) { Quit $$OK }
//if there are filters, loop through them and build an MDX WHERE clause
//which will be a tuple expression containing the members specified by the filters
For i=1:1:filtercount {
  If (i=1)
  {
    Set where=" WHERE {_filterlist.GetAt(i)
  }
  Elseif (i>1) {
    Set where=where_","_filterlist.GetAt(i)
  }
}
Set where=where_")" ; close the tuple expression in the WHERE clause
Set pMDX = pMDX _ where
Quit $$OK
}
```

This example starts with the following MDX query:

```
SELECT {[MEASURES].[%COUNT],MEASURES.[%COUNT],MEASURES.[avg test score]} ON 0,
NON EMPTY [homed].[h1].[city].MEMBERS ON 1 FROM [patients]
```

It then appends a WHERE clause of the following form:
WHERE (member_expression1, member_expression2, member_expression3)

Where each member_expression is a member selected in one of the filters. This WHERE clause uses an MDX tuple expression (parentheses that enclose a list of member identifiers). This expression matches only those records that belong to all the selected members.

18.7.2 MDX KPI Example 2

The DeepSee.Model.KPIs.DemoMDXAutoFilters KPI in SAMPLES is much simpler than the previous one, but does much more because of the superclass that it uses:

```
Class DeepSee.Model.KPIs.DemoMDXAutoFilters Extends DeepSee.Utils.MDXAutoFiltersKPI {
    Parameter CUBE = "PATIENTS";
    Parameter DOMAIN = "PATIENTSAMPLE";
    XData KPI [ XMLNamespace = "http://www.intersystems.com/deepsee/kpi" ] {
        <kpi name="DemoMDXAutoFilters" displayName="DemoMDXAutoFilters" sourceType="mdx"
        mdx="SELECT {[Measures].[%COUNT],[Measures].[Avg Age],[Measures].[Avg Allergy Count]} ON 0,
        NON EMPTY [DiagD].[H1].[Diagnoses].Members ON 1 FROM [Patients]">
            <property name="Patient Count" displayName="Patient Count" columnNo="1" />
            <property name="Avg Age" displayName="Avg Age" columnNo="2" />
            <property name="Avg Allergy Count" displayName="Avg Allergy Count" columnNo="3" />
        </kpi>
    }
}
```

Notice that this class does not directly define any filters, does not directly define the filter members, and defines only a hardcoded MDX query. When you display the test page for this KPI, however, you can use all the levels of the Patients cube as filters, and the KPI appends a suitable %WHERE clause to the query. For example:
This KPI extends the sample DeepSee.Utils.MDXAutoFiltersKPI, which performs the work of defining and using the filters. Within this class:

- `%OnGetFilterList()` retrieves all the levels defined in the cube, as given by the CUBE class parameter.
- `%OnGetFilterMembers()` is implemented. For each level, it retrieves the level members, in the format required by pivot tables; see “Filter Syntax for Pivot Tables,” earlier in this chapter.
- The instance method FilterBuilder() iterates through the cube-based filters, retrieves the current value of each, and then combines them into a string that is suitable for use as an MDX %FILTER clause.
- `%OnGetMDX()` is implemented. It appends the %FILTER clause to the hardcoded query.

### 18.7.3 MDX KPI Example 3

The DeepSee.Model.KPIs.DemoInteroperabilityKPI in SAMPLES is a more complex version of the preceding example. Within this class, `%OnGetFilterList()` retrieves all the levels defined in the cube, as given by the CUBE class parameter. It then adds an additional filter called Yaxis:
ClassMethod %OnGetFilterList(ByRef pFilters As %List, pDataSourceName As %String = "") As %Status
{
    // call method in superclass so we can get filters of the associated cube
    set tSC=##super(.pFilters,pDataSourceName)
    quit:$$$ISERR(tSC) tSC

    // update pFilters array to include the custom filter
    set pFilters($i(pFilters)) = $lb("Yaxis","Yaxis",0)
    quit $$$OK
}

%OnGetFilterMembers() is implemented. For the filter Yaxis, this method provides a set of members. For other filters, it retrieves the level members, in the format required by pivot tables; see “Filter Syntax for Pivot Tables,” earlier in this chapter. This method is as follows:

ClassMethod %OnGetFilterMembers(pFilter As %String, Output pMembers As %List, pSearchKey As %String = ",
pDataSourceName As %String = "") As %Status
{
    set pMembers=""
    if (pFilter="Yaxis") {
        set pMembers($I(pMembers))=$LB("Home City","[homed].[h1].city")
        set pMembers($I(pMembers))=$LB("Favorite Color","[colord].[h1].favorite color")
        set pMembers($I(pMembers))=$LB("Profession","[profld].[h1].profession")
        set pMembers($I(pMembers))=$LB("Diagnoses","[diagld].[h1].diagnoses")
    } else {
        // call method in superclass so we can get filter members for the associated cube
        do ..%GetMembersForFilter(..#CUBE,pFilter,pMembers)
    }
    quit $$$OK
}

Finally, %OnGetMDX() constructs the MDX query. The Yaxis filter determines which level is used for rows. Then the method appends the %FILTER clause to the query; the %FILTER clause uses any cube-based filters as in the previous example.

Method %OnGetMDX(ByRef pMDX As %String) As %Status
{
    set yaxis="", NON EMPTY [profld].[h1].profession.MEMBERS ON 1"
    // check custom filter value
    if (..%filterValues."Yaxis"="") {
        set yaxis="", NON EMPTY "...%filterValues.Yaxis_.MEMBERS ON 1"
    }
    set pMDX="SELECT {MEASURES.[%COUNT],MEASURES.[avg age]} on 0"_yaxis_" FROM "...#CUBE"
    // append a %FILTER clause to handle any other filter values
    Set pMDX = pMDX _ ..FilterBuilder()
    Quit $$$OK
}

18.8 SQL KPI Examples

This section shows examples of %OnGetSQL() that defines queries that use filter values.

18.8.1 KPI Example 1

The following example is from DeepSee.Model.KPIs.DemoSQL. In this case, the filter adds GROUP BY and ORDER BY clauses to the SQL query.

Method %OnGetSQL(ByRef pSQL As %String) As %Status
{
    // this is the start of the SQL query for this KPI
    Set pSQL = "SELECT Count(*),AVG(Age) FROM DeepSee_Study.Patient"

    Set where = ""
    // look at %filterValues to see if a filter has been applied to this KPI instance
    If $IsObject(..%filterValues.ZipCode')=""
    {

// Call utility method that returns filter data in convenient format
Set sqlstring=..%GetSQLForFilter("HomeCity->PostalCode","ZipCode")
Set where = "WHERE "_sqlstring

Set groupby="GROUP BY HomeCity "
Set orderby="ORDER BY HomeCity "
// assemble the SQL statement
Set pSQL=pSQL_where_groupby_orderby
Quit $$$OK

18.8.2 KPI Example 2

The following example is from HoleFoods.KPISQL:

Method %OnGetSQL(ByRef pSQL As %String) As %Status
{
 If $IsObject(..%filterValues) { 
   Set tWHERE = ""
   If (..%filterValues.City'="") { 
     Set tWHERE = tWHERE _ "$S(tWHERE="","1:" AND ") _ " Outlet->City = " " _ ..%filterValues.City _ ":" 
   } 
   If (..%filterValues.Product'="") { 
     Set tWHERE = tWHERE _ "$S(tWHERE="","1:" AND ") _ " Product = " " _ ..%filterValues.Product _ ":" 
   } 
   If (tWHERE'="") { 
     // insert WHERE clause within query
     Set tSQL1 = $P(pSQL,"GROUP BY",1)
     Set tSQL2 = $P(pSQL,"GROUP BY",2)
     Set pSQL = tSQL1 _ " WHERE " _ tWHERE
     If (tSQL2 '="") { 
       Set pSQL = pSQL _ " GROUP BY" _ tSQL2
     } 
   }
   Quit $$$OK
 }

In this case, the KPI defines the initial query within the sql attribute, as described in the previous chapter. The %OnGetSQL() method modifies that query.

18.9 Defining a Listing for a KPI

You can define a KPI so that it includes a listing option. If the KPI also includes filters, the listing definition must consider the filter selections.

To define the listing, you implement another callback method in the KPI.

To define a listing for a KPI, override the %OnGetListingSQL() method, which has the following signature:

ClassMethod %OnGetListingSQL(ByRef pFilters As %String,
ByRef pSelection As %String) As %String

This method returns the text of a listing query. The arguments are as follows:

- **pFilters** is a multidimensional array that contains the current filter values. This array has the following nodes:

<table>
<thead>
<tr>
<th>Node</th>
<th>Node Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pFilters(filter_name) where filter_name is the name of a filter defined in this KPI</td>
<td>Current value of this filter</td>
</tr>
</tbody>
</table>
For details, see “Defining KPI Filters.”

- **pSelection** is a multidimensional array that contains the information about the current selection. This array has the following nodes:

<table>
<thead>
<tr>
<th>Node Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pSelection(&quot;selectedRange&quot;)</strong></td>
<td>Currently selected cells in the pivot as a string in the form &quot;startRow,startCol,endRow,endCol&quot; (1-based).</td>
</tr>
<tr>
<td><strong>pSelection(&quot;rowValues&quot;)</strong></td>
<td>Comma-separated list of the values for the selected rows. In these values, any comma is presented as a backslash ().</td>
</tr>
<tr>
<td><strong>pSelection(&quot;sortColumn&quot;)</strong></td>
<td>Specifies the number of the column to use for sorting the listing. Use 0 for no sorting.</td>
</tr>
<tr>
<td><strong>pSelection(&quot;sortDir&quot;)</strong></td>
<td>Specifies the sort direction, &quot;ASC&quot; or &quot;DESC&quot;</td>
</tr>
</tbody>
</table>

The method should return an SQL select query. In this query, you can also use Caché arrow syntax and SQL functions, as with other listings.

Or you can override the **%OnGetListingResultSet()** method. In this case, you must prepare and execute the result set.

Note that listings are available only within pivot table widgets.

### 18.9.1 Example

The following example is from HoleFoods.KPISQL:

```csharp
ClassMethod %OnGetListingSQL(ByRef pFilters As %String, ByRef pSelection As %String) As %String
{
    Set tSQL = "SELECT TOP 1000 %ID,DateOfSale,Product FROM HoleFoods.SalesTransaction"

    // apply sorting, if asked for
    If (+$G(pSelection("sortColumn"))>0) {
        Set tSQL = tSQL _ " ORDER BY " _ pSelection("sortColumn") _ " " _ $G(pSelection("sortDir"))
    }

    Quit tSQL
}
```

---

Defining DeepSee Models

181
This chapter describes how to implement KPIs that use advanced features. It discusses the following:

- How to define manual KPIs
- How to define KPIs that cache their results
- How to define KPIs that execute asynchronously

Also see the chapters “Defining Basic KPIs” and “Defining KPIs with Filters and Listings.”

19.1 Defining Manual KPIs

Any KPI is an instance of a subclass of %DeepSee.KPI. In a manual KPI, callback methods set properties of that instance. This section discusses the following topics:

- Available properties of the KPI instance
- How to override properties of the KPI
- How to define a manual query

19.1.1 Available Properties

In the callback methods of your KPI instance, the following properties are available:

- %seriesCount — Specifies the number of series (rows) in this KPI.
- %seriesNames(n) — Specifies the name of the series n, where n is an integer.
- %data(n,propname) — Specifies the value of the given property (propname), for the series n.
- %rangeLower — Specifies the lower range value, which sets the default lower range indicator when this KPI is displayed in a meter.
- %rangeUpper — Specifies the upper range value, which sets the default upper range indicator when this KPI is displayed in a meter.
- %thresholdLower — Specifies the lower threshold value, which sets the default lower threshold indicator when this KPI is displayed in a meter.
- %thresholdUpper — Specifies the upper threshold value, which sets the default upper threshold indicator when this KPI is displayed in a meter.
19.1.2 Overriding KPI Properties

The %OnLoadKPI() callback enables you to override properties of the KPI object instance before it is displayed. You can use this to specify the range and threshold values at run time. This callback has the following signature:

Method %OnLoadKPI() As %Status

You can also set these properties within other methods of the KPI class.

19.1.2.1 Example

The following example is from HoleFoods.KPISalesVsTarget:

Method %OnLoadKPI() As %Status
{
    Set tSC = $$$OK
    // Compute additional values
    Set tFilters = ..%filterValues
    // compute recent history using query
    If (((tFilters.Year='')&&(tFilters.Year='*'))) {
        // Take &[] off of Year value! Take &[] off of Year value!
        Set tStartMonth = "Jan-"_$E(tFilters.Year,3,6)
        Set tEndMonth = "Dec-"_$E(tFilters.Year,3,6)
    } Else {
        Set tStartMonth = "NOW-12"
        Set tEndMonth = "NOW"
    }
    Set tROWS = ..RowsClause
    Set tMDX = "SELECT ":%LIST(DateOfSale.[MonthSold]."_tStartMonth_":_tEndMonth_")" _"ON COLUMNS FROM HOLEFOODS WHERE Measures.[Amount Sold] " _..FilterClause
    Set tRS = ##class(%DeepSee.ResultSet).%New()
    Set tSC = tRS.%PrepareMDX(tMDX)
    If $$$ISERR(tSC) Quit tSC
    Set tSC = tRS.%Execute()
    If $$$ISERR(tSC) Quit tSC
    For n = 1:1:..%seriesCount {
        Set tValue = tRS.%GetOrdinalValue(1,n)
        Set ..%data(n,"History") = tValue
    }
    Quit tSC
}

This method populates the History property of this KPI. For each product, this property is a comma-separated list of the past sales, month by month.

19.1.3 Defining a Manual Query

To base a KPI on a manual (custom) query, do the following:

• Specify sourceType="manual" within the <kpi> element.
• Override the %OnExecute() callback method of the KPI class. This method has the following signature:

method %OnExecute() as %Status

In this method, define a query using any logic you need. Then set the %seriesCount, %seriesNames, and %data properties.
19.1.3.1 Example

The following shows a simple example with hardcoded values:

```pascal
Method %OnExecute() As %Status
{
  Set ..%seriesCount=3
  Set ..%seriesNames(1)="alpha"
  Set ..%seriesNames(2)="beta"
  Set ..%seriesNames(3)="gamma"
  Set ..%data(1,"property1")=123
  Set ..%data(1,"property2")=100000
  Set ..%data(1,"property3")=1.234
  Set ..%data(2,"property1")=456
  Set ..%data(2,"property2")=200000
  Set ..%data(2,"property3")=2.456
  Set ..%data(3,"property1")=789
  Set ..%data(3,"property2")=300000
  Set ..%data(3,"property3")=3.789
  Quit $$$OK
}
```

19.2 Defining Cacheable KPIs

By default, a KPI that uses an MDX query is cached (along with all other MDX queries). This cache may or may not be recent enough for your purposes; that is, you can also cache the KPI specifically as described in this section.

By default, non-MDX KPIs are not cached.

To modify a KPI so that its results are cached, make the following changes to the KPI class:

- Specify the `CACHEABLE` class parameter as 1.
- Implement the `%OnComputeKPICacheKey()` method.

```pascal
Method %OnComputeKPICacheKey(Output pCacheKey As %String, pQueryText As %String = "") As %Status

Where `pQueryText` is the text of the KPI query and `pCacheKey` is a unique key to associated with the cached results. Typically this is a hashed version of the query text.
```

- Implement the `%OnComputeKPITimestamp()` method.

```pascal
Method %OnComputeKPITimestamp(ByRef pTimestamp As %String, pSourceType As %String, pQueryText As %String = "") As %Status

Where `pSourceType` is a string that indicates the query type ("mdx", "sql", or "manual"). `pQueryText` is the text of the KPI query, and `pTimestamp` is the timestamp of the KPI.

For a given KPI, if `%OnComputeKPITimestamp()` returns the same timestamp stored in the KPI cache for the given key, DeepSee uses the cached value. Otherwise, DeepSee reruns the KPI.

By default, `%OnComputeKPITimestamp()` returns a timestamp that is precise to the minute. This means that the cache is kept for a minute (at most) by default.

To clear the cache for a given KPI, call its `%ClearKPICache()` method.

19.3 Defining Asynchronous KPIs

Except for plugins (in the next chapter), KPIs are executed synchronously.
To modify a KPI so that it executes asynchronously, make the following changes to the KPI class:

- Specify the **ASYNCH** class parameter as 1.
- Also modify the KPI so that its results are cached. See the previous section. This is required so that DeepSee has a place to store the results.
- Within `%OnCompute()`, optionally call `%SetPercentComplete()` to indicate the state of processing. For details, see “Indicating State of Completion” in the next chapter.
This chapter describes how to define plugins, which are a specialized form of KPI. It discusses the following:

- Introduction
- Requirements for a simple plugin
- How to implement the %OnCompute() method
- How to indicate the state of completion
- How to create a plugin that can be used with multiple cubes
- How to define a calculated member that uses a plugin

**Note:** Before reading this chapter, be sure to read the previous chapters on KPIs.

### 20.1 Introduction

A plugin is a class that defines one or more computations to use in the Analyzer and in queries. A plugin has the following features:

- In any given context, the plugin instance has access to the lowest-level data.
- It executes asynchronously. When the plugin is used in a pivot table, DeepSee can display the plugin current status (as the string \(n\%\) complete) in any pending cells. The pivot table automatically refreshes when the results are available.
- Values returned by the plugin are cached.

Plugins are especially appropriate for complex or time-consuming computations. For example, you might have a computation that uses several different parts of the source record, as well as external information; a plugin would be suitable in this case.

### 20.1.1 How Plugins Can Be Used

Depending on the plugin class, you can use it in some or all of the following ways:

- With the MDX %KPI function. This is possible in all cases.
  
  This means that in all cases, you can define a calculated member that uses the plugin; see “Measures That Use KPIs or Plugins,” earlier in this book.
• Directly in the Analyzer and in widgets. This is possible if the `PLUGINTYPE` class parameter is "Pivot" and the `PUBLIC` class parameter is 1 (the default).

To create a plugin that cannot be directly used in the Analyzer or in widgets, specify `PLUGINTYPE` as "Aggregate". Or specify `PUBLIC` as 0.

## 20.2 Requirements for a Simple Plugin

To define a simple plugin, create a class as follows:

- **Use `%DeepSee.KPIPlugIn` as a superclass.**
- **Define an XData block named KPI that specifies at least one property.** For example:
  ```xml
  XData KPI [ XMLNamespace = "http://www.intersystems.com/deepsee/kpi" ]
  {
  <kpi name="PluginDemo" displayName="PluginDemo" caption="PluginDemo" >
  <property name="PatientCount" displayName="PatientCount" />
  <property name="HighScoreCount" displayName="HighScoreCount" />
  </kpi>
  }
  
  You can also include filters, as with other KPIs.
  
  For details, see the appendix “Reference Information for KPI and Plugin Classes.”

- **Specify the `BASECUBE` class parameter.** For a simple plugin, specify the logical name of a single cube or subject area. (But also see “Creating a Plugin for Multiple Cubes,” later in this chapter.)

- **Specify the base MDX query to use.** Either specify the `mdx` attribute of `<kpi>` or implement the `%OnGetMDX()` method in the following generic way:

  ```csharp
  Method %OnGetMDX(ByRef pMDX As %String) As %Status
  {
  Set pMDX = "SELECT FROM ".#BASECUBE
  Quit $$$OK
  }
  
  DeepSee automatically applies context information (row, column, and filter) to this base query.

- **Optionally specify the `LISTINGSOURCE` class parameter.** Use one of the following values:
  - "SourceTable" — The plugin queries the source table of the given cube. This value is the default.
  - "FactTable" — The plugin queries the fact table of the given cube. See the appendix “Details for the Fact and Dimension Tables.”

  Choose the source that is more convenient.

- **Specify the fields to retrieve in the queried table.** Do one of the following:
  - Specify the `LISTINGFIELDS` class parameter. Specify a comma-separated list of fields to use.

    For example:
    ```csharp
    Parameter LISTINGFIELDS = "Field1, Field2, Field3";
    
    You can specify an alias for any field. For example:
    ```csharp
    Parameter LISTINGFIELDS = "Field1, Field2 as FieldAlias, Field3";
    ```
You can also use Caché arrow syntax and SQL functions, as with other listings. If you use Caché arrow syntax, be sure to specify an alias for the field.

- Implement the %OnGetListingFields() method. For example, the following method causes the plugin to retrieve a single field:

```caché
Method %OnGetListingFields() As %String
{
    //could use an API to get the field name, but in this case factName is set
    //so the field name is known
    Set tListingFields = "MxTestScore"
    Quit tListingFields
}
```

For information, see “Defining a Listing for a KPI,” earlier in this book.

- Implement the %OnCompute method. The following section provides details on this task.
- Optionally specify the PLUGINTYPE and PUBLIC class parameters. See “How Plugins Can Be Used,” earlier in this chapter.

## 20.3 Implementing %OnCompute()

For each pivot table cell where the plugin is used, the plugin executes the listing query and then executes the %OnCompute method. This method has the following signature:

```caché
Method %OnCompute(pSQLRS As %SQL.StatementResult, pFactCount As %Integer) As %Status
```

Where:

- `pSQLRS` is an instance of %SQL.StatementResult that contains the results from executing the listing query.

  For information on using this class, see “Using Dynamic SQL” in Using Caché SQL.

- `pFactCount` is total number of facts in the given context.

In your implementation of this method, do the following:

1. Iterate through the statement result. To do so, use the %Next() method of this instance.
2. As needed, retrieve values for each row. The statement result instance (`pSQLRS`) provides one property for each field in the listing query; the name of the property is the same as the field name.

   For example, in the previous section, %OnGetListingSQL() retrieves a single field, MxTestScore. In this case, `pSQLRS` has a property named MxTestScore.
3. Perform the desired computations.
4. Set the properties of the plugin instance, as described in the previous chapter. At a minimum, set the following properties:
   - `%seriesCount` — Specifies the number of series (rows) in this plugin.
     InterSystems recommends that plugins have only one series. (For plugins with PLUGINTYPE equal to "Pivot", when a user drags and drops a plugin property, the Analyzer uses only the first series.)
   - `%seriesNames(n)` — Specifies the name of the series `n`, where `n` is an integer.
   - `%data(n,propname)` — Specifies the value of the given property (`propname`), for the series `n`.
     The property name must exactly match the name of a <property> in the XData block.
For example:

```java
// place answer in KPI output
Set ..%seriesCount = 1
Set ..%seriesNames(1) = "PluginDemo"
// set Count property of KPI -- just use received pFactCount
Set ..%data(1,"PatientCount") = pFactCount

// iterate through result set to get HighScoreCount
set n = 0
Set highcount = 0
While (pSQLRS.%Next(.tSC)) {
  If $$$ISERR(tSC) Quit
  set n = n + 1
  Set testscore = pSQLRS.MxTestScore
  if (testscore>95) {
    Set highcount = highcount + 1
  }
}
Set ..%data(1,"HighScoreCount") = highcount
```

This is an extract from DeepSee.Model.KPIs.PluginDemo in SAMPLES, which is available in the Analyzer for use with the Patients cube.

### 20.4 Indicating State of Completion

Plugins are executed asynchronously. When a query containing plugins is executed, the query can be complete before the plugins have completed execution. In this case, there are cells whose results are pending. Within these cells, you can display the plugin current status (as the string n% complete). To do so, within %OnCompute(), periodically invoke the %SetPercentComplete() instance method; the argument is an integer between 0 and 100. For example, you could do the following while iterating through the statement result:

```java
// update pct complete
If (n#100 = 0) {
  Do ..%SetPercentComplete(100*(n/pFactCount))
}
```

The appropriate approach depends on the logic in %OnCompute(). In some cases, the majority of the computation time might occur outside of this iteration.

The pivot table automatically refreshes when the results are available.

### 20.5 Creating a Plugin for Multiple Cubes

The previous sections describe how to create a plugin that can be used with a single cube or subject area. You can also create a plugin that can be used in multiple cubes. In practice, this is difficult to do because it is usually necessary to programmatically determine the fields to query.

To create a plugin that you can use with multiple cubes, use the following additional instructions:

- Specify the BASECUBE class parameter as one of the following:
  - A comma-separated list of logical cube or subject area names
  - "*" — refers to all cubes and subject areas in this namespace

  This option determines which listed cubes and subject areas can use the plugin.

- Include the following filter definition within the XData block:
<filter name="%cube" displayName="Subject Area" />

The name must be %cube but you can use any value for the display name.

When you use this plugin within the Analyzer (if applicable), DeepSee passes the name of the current cube or subject area to this filter. Similarly, when you use this plugin within an MDX query, the FROM clause of the query determines the value of this filter.

- Implement the %OnGetMX() method so that it uses the value of the %cube filter. For example:

  Method %OnGetMDX(ByVal pMDX As %String) As %Status
  {
    Set tBaseCube = ""
    // Use %cube filter to find the base cube
    If $IsObject(..%filterValues) {
      If (..%filterValues.%cube'="") {
        Set tBaseCube = ..%filterValues.%cube
      }
    }
    If (tBaseCube'="") {
      Set pMDX = "SELECT FROM ".tBaseCube
    }
    Quit $$$OK
  }

- Ensure that the listing query can work with all the desired cubes and subject areas. Either:
  - For hardcoded listings, use only fields that are suitable in all cases.
  - Programatically determine the fields to use.

### 20.5.1 Determining the Listing Fields Programmatically

If the query for the plugin specifies LISTINGSOURCE as "FactTable", there are additional tools that enable you to programmatically determine the fields to use in %OnGetListingSQL(). You can do the following:

- Include the following filter definition within the XData block:

  <filter name="%measure" displayName="Measure" />

  The name must be %measure but you can use any value for the display name. This filter provides a list of all measures defined in the applicable cube or subject area.

- Within %OnGetListingSQL():
  1. Examine the value of the %measure filter.
  2. Use the %GetDimensionInfo() method of the %DeepSee.Utils class to retrieve, by reference, information about the selected measure.
     Use this information as input for the next step.
  3. Use the %GetDimensionFact() method of the %DeepSee.Utils class to retrieve the name of the field that stores the selected measure.

For an example, see %DeepSee.Plugin.Median. Also see the class reference for the %DeepSee.Utils class.
20.6 Defining a Calculated Member That Uses a Plugin

For any plugin (and any other KPI), you can create a calculated member that retrieves values from it. Then users can drag and drop this member within the Analyzer. To create such a calculated member:

- Define a calculated measure as generally described in “Defining a Calculated Measure,” earlier in this book.
- For **Expression**, specify an MDX expression of the following form:

  \%KPI(pluginname,propertyname,seriesname,\%CONTEXT)

  Where `pluginname` is the name of the plugin, `propertyname` is the name of the property, and `seriesname` is the name of the series. You can omit `seriesname`; if you do, this function accesses the first series in the plugin.

  \%CONTEXT" is a special parameter that provides row, column, and filter context to the plugin; this information is passed to the base MDX query used by the plugin.

  For example (for a plugin with only 1 series):

  \%KPI("PluginDemo2","Count","\%CONTEXT")

  For plugins with `PLUGINTYPE` equal to "Pivot", when a user drags and drops a plugin property, the Analyzer automatically uses syntax like this in the underlying MDX query that it generates.

  For additional options, see the \%KPI function in the *DeepSee MDX Reference*.
This chapter describes how to use the more advanced and the less common features of cubes and subject areas. It discusses the following topics:

- How to specify maxFacts in the cube definition
- How to restrict the records used by the cube
- How to define intermediate expressions for use in building the cube
- How to specify the members manually for a level
- How to define computed dimensions
- How to define the cell cache for better initial performance
- How to add custom indices to the fact table
- How to customize other cube callback methods
- How to customize the subject area %OnGetFilterSpec() callback method
- How to use cube inheritance to define reusable elements

For these tasks, you must use Studio.

21.1 Specifying maxFacts in the Cube Definition

While you are developing a cube, you typically recompile and rebuild it frequently. If you are using a large data set, you might want to limit the number of facts in the fact table, in order to force the cube to be rebuilt more quickly. To do this, you can specify the pMaxFacts argument for %BuildCube(); see “Building the Cube in the Terminal,” earlier in this book.

Or you can specify the maxFacts attribute as follows:

1. In Studio, open the cube class.
2. Find the <cube> element:
3. Add the maxFacts attribute to this element:

```xml
<cube name="HoleFoods"
caption="HoleFoods Sales"
defaultListing="Listing"
nullReplacement="Missing Value"
actionClass="HoleFoods.KPIAction"
sourceClass="HoleFoods.Transaction"
maxFacts="10000">

The value that you specify determines the maximum size of the fact table.

4. Save and recompile the class.

5. Rebuild the cube.

**Important:** Be sure to remove the maxFacts attribute before you deploy the cube.

### 21.2 Restricting the Records Used in the Cube

By default, DeepSee uses all the records in the source class. You can modify the cube definition so that some of the records are ignored. You can do in either or both of the following ways:

- By specifying the buildRestriction attribute of the cube. Note that this has no effect if the cube is based on a data connector.
- By defining the %OnProcessFact() callback of the cube class.

#### 21.2.1 buildRestriction Attribute

To ignore some of the records of the base table and not include them in the cube, you can specify the buildRestriction attribute of the cube definition. This attribute is used to construct an optional WHERE clause to use when building or updating the cube. Specify an SQL comparison expression that uses fields in the source table. For example:

```xml
<cube name="MyCube"
defaultListing="Listing"
nullReplacement="Missing Value"
sourceClass="User.MyClass"
buildRestriction="Domain='12345'"
>

For information on editing the cube definition in Studio, see the previous section.
21.2.2 %OnProcessFact() Callback

To ignore some of the records of the base table and not include them in the cube, you can define the %OnProcessFact() method in your cube definition class:

```plaintext
classmethod %OnProcessFact(pID As %String, ByRef pFacts As %String, 
Output pSkip As %Boolean, pInsert As %Boolean) as %Status
```

DeepSee calls this method immediately after accessing each row in the base table, when building or updating the fact table. It passes the following values to this method:

- **pID** is the ID of the row in the source data being processed.
- **pFacts** is a multidimensional array that contains the values that will be used for the row. This array has the following structure:

<table>
<thead>
<tr>
<th>Node</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pFacts (factName)</td>
<td>where factName is the name of the level or measure in the fact table. See the appendix “Details for the Fact and Dimension Tables.”</td>
</tr>
<tr>
<td>Value of that level or measure. For example, Magnolia or 47.</td>
<td></td>
</tr>
</tbody>
</table>

- **pInsert** is 1 if this is a new row.

If you want to skip this record, your method should set pSkip to true; otherwise it should set pSkip to false.

For example, the following callback causes the cube to ignore all patients whose favorite color is blue:

```plaintext
ClassMethod %OnProcessFact(pID As %String, ByRef pFacts As %String, 
Output pSkip As %Boolean, pInsert As %Boolean) As %Status
{
  if pFacts("DxColor")="Blue"
  {
    set pSkip=1
  } else {
    set pSkip=0
  }
  quit $$$OK
}
```

This example assumes that the cube defines the factName attribute as DxColor for the favorite color level.

For another example, the following callback would cause the cube to ignore all records whose ID was less than 1000000:

```plaintext
ClassMethod %OnProcessFact(pID As %String, ByRef pFacts As %String, 
Output pSkip As %Boolean, pInsert As %Boolean) As %Status
{
  if pID<1000000
  {
    set pSkip=1
  } else {
    set pSkip=0
  }
  quit $$$OK
}
```

21.3 Defining and Using Intermediate Expressions

In some cases, you might have multiple measures or dimensions that use similar logic, perhaps running the same subroutine or using SQL to refer to another table. To improve performance of cube building, you can define expressions, which contain
intermediate values (one value for each fact), and then you can use these expressions within the definitions of other cube elements. To do so:

To do so:

1. In Studio, open the cube class.
2. Find the <cube> starting element:

```xml
<cube name="Patients" displayName="Patients"
      owner="#SYSTEM"
      sourceClass="DeepSee.Study.Patient"
      nullReplacement="None"
      defaultListing="Patient details">
```

3. After the greater than sign (>), add one or more <expression> elements. For example:

```xml
<expression name="patDetails" sourceExpression='%cube.GetPatientDetails(%source.PatientID)' />
```

At a minimum, the <expression> element must have the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of this expression.</td>
</tr>
<tr>
<td>sourceExpression</td>
<td>Optionally specify a Caché ObjectScript expression that returns a single value for any given source record. If you specify this, do not specify sourceProperty. For an &lt;expression&gt; element, it is more likely that you will use sourceExpression (because your cube elements can directly use properties, without the need for the intermediate step provided by &lt;expression&gt;). An &lt;expression&gt; element can refer to another &lt;expression&gt; element.</td>
</tr>
<tr>
<td>sourceProperty</td>
<td>Optionally specify the property name relative to the base class used by the cube; you can use dot syntax to refer to properties of properties. If you specify this, do not specify sourceExpression.</td>
</tr>
</tbody>
</table>

For additional options for sourceExpression and sourceProperty, see “Specifying the Source Values for a Dimension or Level” and “Details for Source Expressions,” earlier in this book.

4. Within the definition of a measure, dimension, level, or another <expression>, use the following syntax to refer to an expression:

```xml`
%expression.expressionName
```

5. Save and recompile the class.
6. Rebuild the cube.

### 21.3.1 Example

First, let us consider a scenario where we might want to use an <expression> element. The Patients cube has several levels that are defined by source expressions that access data via SQL queries to the PatientDetails table. For example, the Favorite Color level is defined with the following source expression:

```xml`
%cube.GetFavoriteColor(%source.PatientID)
```

The GetFavoriteColor() method contains embedded SQL as follows:
ClassMethod GetFavoriteColor(patientID As %String) As %String
{
    New SQLCODE
    &sql(SELECT FavoriteColor INTO :ReturnValue
    FROM DeepSee_Study.PatientDetails
    WHERE PatientID=:patientID)
    If (SQLCODE'=0) {
        Set ReturnValue=""
    }
    Quit ReturnValue
}

The Profession and Industry levels are defined similarly. As a consequence, when the Patients cube, the system executes three queries on the PatientDetails table for each row of the source class.

You can redefine the cube so that the system executes only one query on the PatientDetails table for each row of the source class. To do so:

1. In Studio, open the Patients cube class, DeepSee.Model.PatientsCube/
2. Within the <cube> element in this class, add the following element:

   <expression name="patDetails" sourceExpression='%cube.GetPatientDetails(%source.PatientID)' />

   This expression runs a method in the cube class. The method is defined as follows:

   ClassMethod GetPatientDetails(patientID As %String) As %String
   {
       New SQLCODE
       &sql(SELECT Profession->Industry,Profession->Profession,FavoriteColor
       INTO :Industry,:Profession,:FavoriteColor
       FROM DeepSee_Study.PatientDetails
       WHERE PatientID=:patientID)
       If (SQLCODE'=0) {
           Set Industry="",Profession="",FavoriteColor=""
       }
       SetReturnValue=$LISTBUILD(Industry,Profession,FavoriteColor)
       Quit ReturnValue
   }

   This method retrieves several fields for a given patient, builds a list that contains the information, and returns the list.

3. Redefine the levels that use the PatientDetails table as follows:
   • Redefine the Industry level to use the following sourceExpression:

         sourceExpression='$LI(%expression.patDetails,1)'

   • Redefine the Profession level to use the following sourceExpression:

         sourceExpression='$LI(%expression.patDetails,2)'

   • Redefine the Favorite Color level to use the following sourceExpression:

         sourceExpression='$LI(%expression.patDetails,3)'

4. Save and recompile the class.

5. Rebuild the cube.

21.4 Manually Specifying the Members for a Level

After you define a level in the Architect, you can manually specify the members and their order. To do so, modify the cube class in Studio as follows:
1. In Studio, open the cube class.

2. Find the section that defines the level:

```xml
<level name="MyLevel" displayName="MyLevel" ... >
</level>
```

3. Just before the `</level>` line, add a set of `<member>` elements like the following:

```xml
<level name="MyLevel" displayName="MyLevel" ... >
  <member name="first" displayName="first" spec="1" />
  <member name="second" displayName="second" spec="2" />
  <member name="third" displayName="third" spec="3" />
  <member name="fourth" displayName="fourth" spec="4" />
  <member name="fifth" displayName="fifth" spec="5" />
</level>
```

Each `<member>` element identifies a member as follows:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of this member. This name must exactly match a value of the source property or source expression, including case.</td>
</tr>
<tr>
<td>displayName</td>
<td>Optional display name of this member. These are localized in the same way as other display names in the cube.</td>
</tr>
<tr>
<td>spec</td>
<td>Optional key for this member. If you omit this, <code>name</code> becomes the key.</td>
</tr>
</tbody>
</table>

Notes:

- In these attributes, you cannot use XML reserved characters. See the following subsection.
- Include as many `<member>` elements as needed.
- The order of these elements determines the default sort order of the corresponding members.
- Include all members of the level (there is no wildcard option).

4. Save and recompile the class.

5. Rebuild the cube.

**Note:** This option is useful only when the set of members is unlikely to change. If the system receives new records that contain a value not given by one of the `<member>` elements, those records are not represented in this level.

### 21.4.1 XML Reserved Characters

When you edit a cube class in Studio, you cannot use XML reserved characters for the values of `name`, `displayName`, or other attribute values. Instead substitute as follows:

<table>
<thead>
<tr>
<th>Reserved Character</th>
<th>Use This Instead</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td><code>&amp;lt;</code></td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td><code>&amp;amp;</code></td>
</tr>
</tbody>
</table>

When you create cube elements in the Architect, and you type these characters into input fields, the system automatically makes these substitutions. If you examine a cube class in Studio, you might notice that the Architect also makes other substitutions. The preceding table lists only the substitutions that are necessary.
21.5 Defining Computed Dimensions

A computed dimension contains members that are determined at runtime by SQL queries. You specify the query for each member.

Computed dimensions do not have any association with calculated members. A computed dimension is specific to DeepSee. A calculated member is a standard concept in MDX.

**Note:** The null replacement option does not have any effect on computed dimensions.

21.5.1 Example Computed Dimension

The Patients cube includes the `ComputedD` dimension, which is disabled by default. If you open the cube class in Studio, change `disabled="true"` to `disabled="false"`, and recompile, you can use this dimension in the Analyzer.

For this dimension, the Analyzer displays the following members:

```
▼ ComputedD
    ▼ Computed
        □ member 1
        □ member 2
```

You can use this dimension, the `Computed` level, and the members in the same way that you use other elements of the cube.

Each of these members is defined by an SQL query that retrieves the IDs of specific patients.

21.5.2 Defining a Computed Dimension

You can use the following procedure to define a computed dimension fairly easily:

1. Use the Architect to add a dimension, hierarchy, and level with the names you want.
   - It does not matter what type of dimension you choose, because you will change that later.
   - Do not specify a source property or source expression.
   - This step creates a shell that you can easily edit in Studio. Also, the Architect initializes the display names to be the same as the names, which is convenient.

2. In Studio, open the cube class.

3. Modify the `<dimension>` element that corresponds to the computed dimension. Make the following changes:
   - Edit the `type` attribute to be as follows:
     `type="computed"`
   - Add either of the following to the `<dimension>` element:
     ```
     dimensionClass="SQL"
     dimensionClass="%DeepSee.ComputedDimension.SQL"
     ```

   The `dimensionClass` attribute refers to a helper class. If you do not specify a full package and class name, DeepSee assumes that this class is in the package `%DeepSee.ComputedDimension`. The class `%DeepSee.ComputedDimension.SQL` is the helper class for SQL-based computed dimensions. Other types of computed dimensions are beyond the scope of this documentation.
For example:
<dimension name="New Computed Dimension" displayName="New Computed Dimension"
    disabled="false"
    hasAll="false"
    type="computed"
    dimensionClass="SQL" />

The lines are not broken in this way by default; line breaks are added here to make the example easier to read.

4. Find the section that defines the level:

<level name="New Computed Level" displayName="New Computed Level"
    disabled="false"
    list="false"
    useDisplayValue="true" />

5. Just before the </level> line, add a line like the following:

<member name="" displayName="" spec="" />

Edit this as follows:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of this member.</td>
</tr>
<tr>
<td>displayName</td>
<td>Display name of this member.</td>
</tr>
<tr>
<td>spec</td>
<td>SQL SELECT query that returns the IDs of one or more records of the fact table used by this cube. To refer to the fact table, you can use either the token $$$TABLE or the full table name. For details on the fact table, see the appendix “Details for the Fact and Dimension Tables.” Also see “Variations for spec,” later in this chapter.</td>
</tr>
</tbody>
</table>

For example:

member name="example member 1" displayName="member 1"
    spec="select ID from DeepSee_Model_PatientsCube.Fact WHERE MxAge<50 AND DxHomeCity->DxHomeCity='Elm Heights'" />

In any of these attributes, you cannot use XML reserved characters. For substitutions, see “XML Reserved Characters.”

6. Add other <member> elements as needed.

The order of these elements determines the default sort order of the corresponding members.

7. Save and recompile the class.

As soon as you do so, the new dimension and its members are available for use.

21.5.3 A Closer Look at the Example

In the Patients cube, the ComputedD dimension contains the following members:

- member 1 is defined by the following SQL query:

```
select ID from DeepSee_Model_PatientsCube.Fact WHERE MxAge<50 AND DxHomeCity->DxHomeCity='Elm Heights'
```

This query uses the fact table for the cube; for details, see the appendix “Details for the Fact and Dimension Tables.”

- member 2 is defined by the following SQL query:

```
select ID from $$$TABLE WHERE MxAge=40 AND DxHomeCity->DxHomeCity='Juniper'
```
This query uses $$\text{TABLE}$$, which is replaced by the actual name of the fact table. Unlike the previous query, this query is not valid in the Management Portal, which does not have information to convert the $$\text{TABLE}$$ token.

The following pivot table shows these members:

<table>
<thead>
<tr>
<th>Computed</th>
<th>Avg Age</th>
<th>Patient Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>member 1</td>
<td>24.50</td>
<td>801</td>
</tr>
<tr>
<td>member 2</td>
<td>40</td>
<td>18</td>
</tr>
</tbody>
</table>

For example, in this case member 1 consists of 801 patients whose average age is 24.50 years.

21.5.4 Variations for spec

The following variations are possible for spec:

- The query can instead refer to the base class of the cube, if that class has indices that are appropriate for the query. The IDs for that class are identical to the IDs used in the fact table.

  This is helpful in cases when you want a dimension to reflect frequently changing groups and you want to avoid rebuilding or synchronizing the cube.

- As shorthand, you can use a WHERE clause instead of a SELECT statement. In this case, DeepSee generates a SELECT statement that uses your WHERE clause; this statement selects IDs from the fact table.

  For an example, see the members in the By Industry level in the example computed dimension later in this section.

- As shorthand, you can refer to a field in the fact table via the token $$\text{FACT}$$. To do so, in the level, specify the factName attribute as the name of that field. For example:

  `<level name="By Allergy Count" factName="Mx1968652733I"` >

  Then in the SELECT statement (or the WHERE clause), use $$\text{FACT}$$ to refer to this field. Be sure to include spaces around this token so that DeepSee parses it appropriately.

  For an example, see the members in the By Allergy Count level in the example computed dimension later in this section.

The following computed dimension works with the Patients cube. To use it, copy and paste it into the cube class and then recompile that class.

```xml
<dimension name="Other Groups" displayName="Groups (Computed Dimension)
 hasAll="false" type="computed" dimensionClass="SQL">
  <hierarchy name="H1">
    <level name="By Industry" >
      <member name="Retail Trade" 
        spec="WHERE DxIndustry->DxIndustry='Retail Trade'"> </member>
      <member name="Finance and Insurance" 
        spec="WHERE DxIndustry->DxIndustry='Finance and Insurance'"> </member>
      </level>
  </hierarchy>
  <hierarchy name="H2">
    <level name="By Allergy Count" factName="Mx1968652733I" >
      <member name="Highly allergic" 
        spec="WHERE $$\text{FACT} > 2""> </member>
      <member name="Not allergic" 
        spec="select ID from $$\text{TABLE} WHERE $$\text{FACT} = 0"> </member>
    </level>
  </hierarchy>
</dimension>
```
21.6 Defining the Cell Cache

Your cube class can contain an additional XData block (CellCache) that specifies cube cells that can be precomputed and cached, which speeds up the initial performance of DeepSee. This section shows an example and gives the details for the contents.

For information on precomputing the cube cells specified in this XData block, see the *DeepSee Implementation Guide*.

### 21.6.1 Example CellCache XData Block

```xml
/// This xml document defines aggregates to be precomputed.
XData CellCache [ XMLNamespace = " http://www.intersystems.com/deepsee/cellCache" ]
{
<cellCache xmlns= "http://www.intersystems.com/deepsee/cellCache" >
  <group name= "BS">
    <item>
      <element >[Measures].[Big Sale]</element >
    </item>
  </group>
  <group name= "G1">
    <item>
      <element >[UnitsPerTransaction].[H1].[UnitsSold]</element >
      <element >[Measures].[Amount Sold]</element >
    </item>
    <item>
      <fact >DxUnitsSold</fact >
      <element >[Measures].[Amount Sold]</element >
    </item>
  </group>
</cellCache >
}
```

### 21.6.2 <cellCache> Details

The `<cellCache>` element is as follows:

- It must be in the namespace "http://www.intersystems.com/deepsee/cellCache"
- It contains zero or more `<group>` elements.

Each `<group>` element is as follows:

- It has a `name` attribute, which you use later when specifying which groups of cells to precompute.
- It contains one or more `<item>` elements.

Each `<item>` element represents a combination of cube indices and corresponds to the information returned by `%SHOWPLAN`. An `<item>` element consists of one or more `<element>` elements.

An `<element>` can include one or more of either of the following structures, in any combination:

- `<fact>fact_table_field_name</fact>`

Or:

- `<element>mdx_member_expression</element >`

Where:

- `fact_table_field_name` is the field name in the fact table for a level or measure, as given by the `factName` attribute for that level or measure.
mdx_member_expression is an MDX expression that evaluates to a member. This can be either a member of a level or it can be a measure name (each measure is a member of the special MEASURES dimension).

Note: Each group defines a set of intersections. The number of intersections in a group affects the processing speed when you precompute the cube cells.

21.7 Adding Custom Indices to the Fact Table

DeepSee automatically defines all the indices that it needs. However, you can use the fact table directly for your own purposes, and if you do, you might need additional indices. To add them, edit the cube class in Studio and add <index> elements as needed.

See “<index>” in the appendix “Reference Information for Cube Classes.”

21.8 Customizing Other Cube Callbacks

The class %DeepSee.CubeDefinition provides callback methods that you can override to customize the cube behavior further. This section describes the most commonly overridden methods (apart from %OnProcessFact(), which is discussed earlier in this chapter). Also see the class reference for %DeepSee.CubeDefinition.

21.8.1 %OnGetDefaultListing() Callback

The %OnGetDefaultListing() callback enables you to programmatically specify the name of the listing to use, in the case where DeepSee uses the default listing. This callback has no effect when a specific listing is requested.

```c
ClassMethod %OnGetDefaultListing() As %String
{
  Quit "Listing By Product"
}
```

21.8.2 %OnExecuteListing() Callback

In some cases, additional setup work is required before a listing query can run.

To do this, implement the %OnExecuteListing() method in your cube definition class:

```c
classmethod %OnExecuteListing(pSQL As %String) as %Status

DeepSee calls this method immediately before it executes a listing query. When DeepSee calls this method, it passes the value pSQL, which is the listing query that will be executed.
```

21.8.3 %OnGetFilterSpec() Callback

Instead of (or in addition to) specifying filters in subject areas, you can implement the %OnGetFilterSpec() callback. This enables you to filter the cube at runtime. The signature of this method is as follows:

```c
classmethod %OnGetFilterSpec(pFilterSpec As %String) as %String

You can also implement this callback within a subject area, which is more common. For details, see the next section.
21.9 Customizing the Subject Area %OnGetFilterSpec() Callback

Instead of (or in addition to) specifying a hardcoded filter for a subject area, you can implement the %OnGetFilterSpec() callback. This enables you to specify the contents of the filter at runtime. The signature of this method is as follows:

```classmethod %OnGetFilterSpec(pFilterSpec As %String) as %String```

Here `pFilterSpec` is the value of the filterSpec attribute in `<subjectArea>`. The method must return a valid MDX set expression (see “Writing Filter Expressions,” earlier in this book). For example:

```ClassMethod %OnGetFilterSpec(pFilterSpec As %String) As %String
{
    Quit "AgeD.H1.[20 to 29]"
}
```

The following shows another simple example. In this case, the method checks the `$ROLES` special variable and removes the subject area filtering if the user belongs to the `%All` role:

```ClassMethod %OnGetFilterSpec(pFilterSpec As %String) As %String
{
    if $ROLES["%All" { //remove any filtering
        set pFilterSpec=""
    }
    Quit pFilterSpec
}
```

For another example, the following callback modifies the original filter value by performing a cross join of the original value and an additional filter:

```ClassMethod %OnGetFilterSpec(pFilterSpec As %String) As %String
{
    //test to see if $ROLES special variable includes TestRole
    if $ROLES["%DB_SAMPLE" { //a member expression like the following is a simple set expression
        set colorrestrict="colord.h1.[favorite color].red"
        //create a tuple that intersects the old filter with the new filter
        //this syntax assumes original is just a member
        set newfilter="CROSSJOIN(_pFilterSpec_","_colorrestrict_")"
        set pFilterSpec=newfilter
    }
    Quit pFilterSpec
}
```

21.10 Using Cube Inheritance to Define Reusable Elements

In some cases, it is necessary to define multiple similar cubes. DeepSee provides a simple inheritance mechanism to enable you to do so easily. This mechanism works as follows:

1. Define one cube class that contains the core items that should be in all the similar cubes.
   This cube is the parent cube.
2. Optionally mark this cube as abstract so that it cannot be used directly.
To do so, specify `abstract="true"` in the `<cube>` element of this class. Then the compiler does not validate it, the Analyzer does not display it, and you cannot execute queries against it.

3. Define the child cubes in their own cube classes. For each of these cubes, specify the `inheritsFrom` attribute. For the value of this attribute, specify the logical name of the parent cube.

   This step means that each of these subcubes contains all the definitions from the parent cube.

4. Optionally specify additional definitions (dimensions, measures, listings, and so on) in these cubes.

   Also optionally redefine anything specified in the parent cube. To do so, specify a definition in the child cube and use the same name as in the parent cube.

   Each subcube has its own fact table and indices, and (at run time) these are independent of the parent cube. The inheritance mechanism is used only at build time, and affects only the definitions in the cubes.

Note the following points:

- The cube inheritance mechanism has no relationship with class inheritance.
- The parent cube must be compiled before any of its child cubes.
- A child cube cannot remove items that are in the parent cube.
- A child cube cannot redefine only part of an element. For example, it cannot modify part of a dimension. Any definition in the child cube must be a complete definition.
- You can specify only one cube for `inheritsFrom`.
- For `inheritsFrom`, you can specify a cube that inherits from another cube.

**Tip:** To refer to methods in the parent cube class, either use the usual full syntax (`##class()`) or do the following:

1. Modify the child cube class that it extends the parent cube class.

2. Use the variable `%cube` as described in “Defining the Source Values for a Dimension or Level.”
A Reference Information for Cube Classes

The Architect creates and modifies cube classes, which you can also create and edit directly in Studio. This appendix provides reference information on these classes. It discusses the following topics:

- Basic requirements for a cube class
- Common attributes in a cube
- `<cube>`
- `<measure>`
- `<dimension>`
- `<hierarchy>`
- `<level>`
- `<member>`
- `<property>`
- `<listing>`
- `<calculatedMember>`
- `<namedSet>`
- `<relationship>`
- `<index>`
- `<expression>`

Also see “Using Advanced Features of Cubes and Subject Areas” and “When to Recompile and Rebuild,” earlier in this book.

A.1 Requirements for a Cube Class

To define a cube, create a class that meets the following requirements:

- It must extend `%DeepSee.CubeDefinition`.
- It must contain an XData block named `Cube`
- For this XData block, XMLNamespace must be specified as follows:
XMLNamespace = "http://www.intersystems.com/deepsee"

- The root element within the XData block must be <cube> and this element must follow the requirements described in the rest of this appendix.

- It is useful, but not required, for the class to specify the DependsOn compiler keyword so that this class is compiled only after the source class of the cube is compiled and can be used.

- The class can define the DOMAIN parameter, which specifies the domain to which any localized strings belong. For example:

  Parameter DOMAIN = "PATIENTSAMPLE";

  For details, see the chapter “Performing Localization” in the DeepSee Implementation Guide.

For example:

```java
  XData Cube [ XMLNamespace = "http://www.intersystems.com/deepsee" ] {
    <cube
      name="Patients"
      owner="_SYSTEM"
      caption="Patients"
      sourceClass="DeepSee.Study.Patient"
      other_cube_options...
    >
    <measure measure_definition/>
    ...
    <dimension dimension_definition/>
    ...
  </cube>
}
```

Studio provides assistance as you type. For example:

```xml
  <measure name="MyMeasure"
    formatString A
    hidden B
    name C
    scale D
    sourceExpression E
    sourceProperty F
  >
```

### A.2 Common Attributes in a Cube

Most of the elements in the cube have the following attributes, which are listed here for brevity:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Logical name of the element for use in MDX queries against this cube. See See “Names for Model Elements,” earlier in this book.</td>
</tr>
</tbody>
</table>
### Purpose Attribute

(Optional) Localized name of this element for use in user interfaces. If you do not specify this attribute, the user interface instead displays the value specified by the `name` attribute. For details, see the chapter “Performing Localization” in the *DeepSee Implementation Guide*.

### description

(Optional) Description of this element.

### disabled

(Optional) Controls whether the compiler uses this element. If this attribute is "true" then the compiler ignores the element; this is equivalent to commenting out the element. By default, this attribute is "false".

---

### A.3 `<cube>`

The `<cube>` element is the root element in the XData block. This element contains the following contents:

<table>
<thead>
<tr>
<th>Attribute or Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes in a Cube” earlier in this appendix.</td>
</tr>
</tbody>
</table>
| sourceClass | Complete package and class name of the base class used by this cube. This must be one of the following:
  - A persistent class.
  - A data connector (a class that extends `%DeepSee.DataConnector`). See the *DeepSee Implementation Guide*. |
<p>| owner | (Optional) Name of owner of cube. |
| resource | (Optional) Name of the resource used to control access to this cube, when it is accessed via the DeepSee Architect. See the <em>DeepSee Implementation Guide</em>. |
| caption | (Optional) Caption to display in the Analyzer and other utilities when working with this cube. |
| countMeasureCaption | (Optional) Caption to use for the default measure, which counts records. The default caption is <code>Count</code>. Internally, the name of the measure is <code>%Count. | | countMeasureName | (Optional) Name to use for the default measure, which counts records. The default name of this measure is </code>%Count. |
| defaultListing | (Optional) Specify the logical name of the <code>&lt;listing&gt;</code> to use as the default in this cube. See “&lt;listing&gt;,” later in this appendix. |
| nullReplacement | (Optional) Specifies the string to use as the member name if the source data for a level is null. For example, specify <code>nullReplacement=&quot;None&quot;</code>. This option is overridden by the <code>nullReplacement</code> attribute, if specified, for a given <code>&lt;level&gt;</code>. This option has no effect on levels that use range expressions. |</p>
<table>
<thead>
<tr>
<th>Attribute or Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>bucketSize</td>
<td>(Optional) Specifies the size of the caching buckets used for this cube. A <em>group</em> is 64,000 contiguous rows, and a <em>bucket</em> is an integer multiple of groups. The default bucket size is 8. That is, by default, the first 512,000 rows are bucket number 1, the next 512,000 rows are bucket number 2, and so on. In some cases, you may want to increase this value if you expect to have a large number of facts and do not expect many updates to older values within the cube. If you change this value, be sure to delete all cached results for this cube. To do so, call the <code>%KillCache()</code> method of the cube definition class.</td>
</tr>
<tr>
<td>actionClass</td>
<td>(Optional) Specifies an associated KPI class that defines actions that are available for pivot tables based on this cube. Specify the full package and class name of a subclass of <code>%DeepSee.KPI</code>.</td>
</tr>
<tr>
<td>maxFacts</td>
<td>(Optional) Specifies the maximum number of rows in the fact table. This determines the number of rows of the base table that the system uses when building the cube. Use this attribute to assist in debugging when you do not want to process the entire base table. By default, all rows of the base table are processed.</td>
</tr>
<tr>
<td>bitmapChunkInMemory</td>
<td>(Optional) Controls how DeepSee builds the indices: using a process-private global (slower but does not lead to &lt;STORE&gt; errors) or entirely in memory (faster). The default is &quot;false&quot;, in which case DeepSee uses a process-private global for temporary storage while it builds in the indices. If your cube does not contain a large number of indices (dimensions and measures) and does not contain dimensions with very large numbers of members, you can use &quot;true&quot; for faster performance. It is worthwhile to try this setting; if you encounter &lt;STORE&gt; errors when you build the cube, then change the value back to the default.</td>
</tr>
<tr>
<td>initialBuildOrder</td>
<td>(Optional) Used to construct an optional ORDER BY clause for use when building the entire cube; does not affect incremental updates. Specify a comma-separated list of fields in the source table. You can use the SQL keywords ASC and DESC. For example: &quot;Age DESC,Gender&quot;</td>
</tr>
<tr>
<td>buildRestriction</td>
<td>(Optional) Used to construct an optional WHERE clause to use when building or updating the cube. Specify an SQL comparison expression that uses fields in the source table. For example: &quot;Gender='F'&quot; This attribute has no effect if the cube is based on a data connector.</td>
</tr>
<tr>
<td>inheritsFrom</td>
<td>Do not use.</td>
</tr>
<tr>
<td>defaultMeasure, defaultMember, precompute</td>
<td>Do not use.</td>
</tr>
<tr>
<td>&lt;measure&gt;</td>
<td>(Optional) You can include zero or more &lt;measure&gt; elements, each of which defines a measure.</td>
</tr>
<tr>
<td>&lt;dimension&gt;</td>
<td>(Optional) You can include zero or more &lt;dimension&gt; elements, each of which defines a dimension.</td>
</tr>
<tr>
<td>Attribute or Element</td>
<td>Purpose</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------</td>
</tr>
<tr>
<td><code>&lt;listing&gt;</code></td>
<td>(Optional) You can include zero or more <code>&lt;listing&gt;</code> elements, each of which defines a listing, to be available in the Analyzer.</td>
</tr>
<tr>
<td><code>&lt;namedSet&gt;</code></td>
<td>(Optional) You can include zero or more <code>&lt;namedSet&gt;</code> elements, each of which defines a named set, which is an alias for a member or for a set of members.</td>
</tr>
<tr>
<td><code>&lt;calculatedMember&gt;</code></td>
<td>(Optional) You can include zero or more <code>&lt;calculatedMember&gt;</code> elements, each of which defines a member that is calculated in terms of other members. This is typically used to define measures that are calculated based on other measures.</td>
</tr>
<tr>
<td><code>&lt;relationship&gt;</code></td>
<td>(Optional) You can include zero or more <code>&lt;relationship&gt;</code> elements. A <code>&lt;relationship&gt;</code> element connects another cube to your cube so that you can use levels of that other cube when you query your own cube.</td>
</tr>
<tr>
<td><code>&lt;index&gt;</code></td>
<td>(Optional) You can include zero or more <code>&lt;index&gt;</code> elements, each of which defines an optional, custom index on the fact table. This can be useful if you plan to run SQL queries on the fact table. For information on the fact table, see <a href="#">Accessing DeepSee Data Programmatically</a>.</td>
</tr>
<tr>
<td><code>&lt;expression&gt;</code></td>
<td>(Optional) You can include zero or more <code>&lt;expression&gt;</code> elements, each of which defines an expression (typically an intermediate value), for use in the definition of one or more measures or dimensions. Values of expressions are not stored but are available during cube build time.</td>
</tr>
</tbody>
</table>

For example:

```xml
<cube
  name="Patients"
  owner="_SYSTEM"
  caption="Patients"
  sourceClass="DeepSee.Study.Patient"
  >
  ...
</cube>
```

### A.4 `<measure>`

The `<measure>` element defines a measure; your cube can have any number of measures. This element has the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes in a Cube” earlier in this appendix.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Purpose</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>sourceProperty</td>
<td>Specify one of these attributes in almost the same way that you would specify Property or Expression in the Architect; see “Specifying the Source Values for a Dimension or Level” and “Details for Source Expressions,” earlier in this book. Notes:</td>
</tr>
<tr>
<td></td>
<td>• For sourceProperty, use the same value you would enter for Property, enclosed within single quotes.</td>
</tr>
<tr>
<td></td>
<td>For example: sourceProperty='Age'</td>
</tr>
<tr>
<td></td>
<td>• For sourceExpression, use the same value you would enter for Expression, enclosed within single quotes.</td>
</tr>
<tr>
<td></td>
<td>For example: sourceExpression='%source.property/100'</td>
</tr>
<tr>
<td></td>
<td>You can enclose the value in double quotes instead if the value itself does not contain any double quotes. For example: sourceExpression='%source.property/100'</td>
</tr>
<tr>
<td>sourceExpression</td>
<td>(Optional) Specifies how to aggregate values for this measure, whenever combining multiple records. If you specify this, use one of the following values:</td>
</tr>
<tr>
<td></td>
<td>• &quot;SUM&quot;</td>
</tr>
<tr>
<td></td>
<td>• &quot;COUNT&quot; — Counts the records for which the source data has a non-null (and nonzero) value.</td>
</tr>
<tr>
<td></td>
<td>• &quot;MAX&quot; — Uses the largest value in the set.</td>
</tr>
<tr>
<td></td>
<td>• &quot;MIN&quot; — Uses the smallest value in the set.</td>
</tr>
<tr>
<td></td>
<td>• &quot;AVG&quot; — Calculates the average value for the set.</td>
</tr>
<tr>
<td></td>
<td>For a boolean or a string measure, this attribute must be &quot;COUNT&quot;.</td>
</tr>
<tr>
<td>type</td>
<td>(Optional) Specifies the data type for the measure. Most measures are numeric, which is the default. See “Specifying the Measure Type.”</td>
</tr>
<tr>
<td>iKnowSource,</td>
<td>(Optional) See “Using Unstructured Data in Cubes.”</td>
</tr>
<tr>
<td>iKnowDomain,</td>
<td>iKnowDictionaries,</td>
</tr>
<tr>
<td>iKnowParameters</td>
<td>(Optional) Specifies how to aggregate values for this measure, whenever combining multiple records. If you specify this, use one of the following values:</td>
</tr>
<tr>
<td>aggregate</td>
<td>(Optional) If hidden=&quot;true&quot; then the measure is defined and can be used in queries, but is not listed as an available measure. This lets you define measures that serve as intermediate calculations.</td>
</tr>
<tr>
<td>hidden</td>
<td>(Optional) Specifies the number of decimal places to keep. By default, scale is 0, and DeepSee rounds each measure value to a whole number before writing it to the fact table.</td>
</tr>
<tr>
<td>scale</td>
<td>(Optional) See “Linking to Another Table.”</td>
</tr>
</tbody>
</table>
### Purpose

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
</table>
| searchable        | (Optional) If searchable="true" then DeepSee displays this measure as an option in advanced filters, where you can create filter expressions that refer to the measure value (typically to compare the measure value to a constant).  
For a searchable measure, DeepSee adds an additional index, if appropriate, to support these expressions. See “Measure Search Expressions” in the DeepSee MDX Reference.  
Note that it may be possible to use a measure as a searchable measure in a manual MDX query even if it is not marked as searchable="true". In all cases, the measure is also available for use in the same way as any other measure. |
| factName          | (Optional) Name, in the generated fact table, for the column that corresponds to this measure. If this attribute is null, the system generates a name. |
| factSelectivity   | (Optional, not used by DeepSee) Value to override the generated SELECTIVITY parameter for the property in the fact table class. DeepSee queries do not use this parameter. This option is intended for cases where SQL is used directly against the generated fact table and the default SELECTIVITY needs to be overridden.  
Specify a positive value less than or equal to 1. For details, see SetFieldSelectivity() in %SYSTEM.SQL. |
| formatString      | (Optional) Controls how the values are displayed. See “formatString Details.” |
| units             | (Optional) Indicates the units in which the measure value is expressed. Currently this attribute is provided only for general information. |

For example:

```xml
<measure name="Test Score" sourceProperty="TestScore" aggregate="SUM"/>
<measure name="Avg Test Score" sourceProperty="TestScore" aggregate="AVG"/>
<measure name="Allergy Count"
    sourceExpression="##class(Cubes.StudyPatients).GetAllergyCount(%source.%ID)"/>
```

Notice that the Allergy Count measure uses a utility method that is defined within the cube class.

#### A.4.1 %COUNT Measure

DeepSee includes a predefined measure named %COUNT, which returns the number of facts in a query.

#### A.4.2 Measures Dimension

DeepSee automatically creates the Measures dimension, and places all measures into it.
A.4.3 formatString Details

The `formatString` attribute is a pieced string as follows:

Format^Color

- For the `Format` piece, specify a string that consists of one to four subpieces as follows:

  positive_format;negative_format;zero_format;missing_format;

  Where `positive_format` controls how a positive value is displayed, `negative_format` controls how a negative value is displayed, `zero_format` controls how zero is displayed, and `missing_format` controls how a missing value is displayed.

  Each subpiece is a string that includes one of the following base units: #, #, #, #, or #, #. For details, see “Specifying a Format String,” earlier in this book.

- For the `Color` piece, specify a string that consists of one to four subpieces as follows:

  positive_color;negative_color;zero_color;missing_color;

  Where `positive_color` controls the color for a positive value, `negative_color` controls the color for a negative value, `zero_color` controls the color for zero, and `missing_color` controls the color for a missing value.

  For each of these, specify a CSS color name or a hex color code.

For date measures, use the following variation:

%date%^Color

Where `Color` is as given above. The `%date%` piece formats the date according to the default date format for the current process.

A.5 <dimension>

The `<dimension>` element defines a dimension, which contains one or more hierarchies, each of which contains one or more levels. Your cube can have any number of dimensions.

The `<dimension>` element has the following contents:

<table>
<thead>
<tr>
<th>Attribute or Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes in a Cube” earlier in this appendix.</td>
</tr>
<tr>
<td>Attribute or Element</td>
<td>Purpose</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>sourceProperty</td>
<td>Specify one of these attributes in almost the same way that you would specify <strong>Property</strong> or <strong>Expression</strong> in the Architect; see “Defining the Source Values for a Dimension or Level.” Notes:</td>
</tr>
<tr>
<td></td>
<td>• For <code>sourceProperty</code>, use the same value you would enter for <strong>Property</strong>, enclosed within single quotes.</td>
</tr>
<tr>
<td></td>
<td>For example: <code>sourceProperty='Age'</code></td>
</tr>
<tr>
<td></td>
<td>• For <code>sourceExpression</code>, use the same value you would enter for <strong>Expression</strong>, enclosed within single quotes.</td>
</tr>
<tr>
<td></td>
<td>For example: <code>sourceExpression='%source.property_&quot;ABC&quot;'</code></td>
</tr>
<tr>
<td></td>
<td>You can enclose the value in double quotes instead if the value itself does not contain any double quotes. For example: <code>sourceExpression='%source.property'</code></td>
</tr>
<tr>
<td>type</td>
<td>Specify one of the following:</td>
</tr>
<tr>
<td></td>
<td>• &quot;data&quot; (the default) — Creates an ordinary dimension.</td>
</tr>
<tr>
<td></td>
<td>• &quot;time&quot; — Creates a time dimension. See “Defining a Time Level.”</td>
</tr>
<tr>
<td></td>
<td>• &quot;age&quot; — Creates an age dimension. See “Defining an Age Level.”</td>
</tr>
<tr>
<td></td>
<td>• &quot;computed&quot; — Creates a computed dimension. See “Defining Computed Dimensions.”</td>
</tr>
<tr>
<td></td>
<td>• &quot;iKnow&quot; — Creates a dimension uses text processed by iKnow. See “Using Unstructured Data in Cubes.”</td>
</tr>
<tr>
<td>iKnowMeasure and iKnowType</td>
<td>(Optional) See “Using Unstructured Data in Cubes.”</td>
</tr>
<tr>
<td>hasAll</td>
<td>(Optional) Indicates whether this dimension has an All Level. The default is &quot;true&quot;.</td>
</tr>
<tr>
<td>allCaption</td>
<td>(Optional) Name used for the All Level and All Member for this dimension. The default name is All dimension, where <code>dimension</code> is the dimension name.</td>
</tr>
<tr>
<td>allDisplayName</td>
<td>(Optional) Specifies the localized name used for the All member. If you do not specify this attribute, the user interface instead displays the value specified by the <code>allCaption</code> attribute.</td>
</tr>
<tr>
<td>hidden</td>
<td>(Optional) If hidden=&quot;true&quot; then the dimension is defined and can be used in queries, but is not listed as an available dimension. The default is &quot;false&quot;.</td>
</tr>
<tr>
<td>showHierarchies</td>
<td>(Optional) Controls whether the Analyzer displays the hierarchy names within this dimension. Specify one of the following:</td>
</tr>
<tr>
<td></td>
<td>• &quot;default&quot; — Display the hierarchy names only if there is more than one hierarchy.</td>
</tr>
<tr>
<td></td>
<td>• &quot;true&quot; — Always display the hierarchy names.</td>
</tr>
<tr>
<td></td>
<td>• &quot;false&quot; — Never display the hierarchy names.</td>
</tr>
<tr>
<td></td>
<td>This attribute has no effect on the queries themselves.</td>
</tr>
</tbody>
</table>
### A.6 `<hierarchy>`

The `<hierarchy>` element specifies a hierarchy within the given dimension. This element has the following contents:

<table>
<thead>
<tr>
<th>Attribute or Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes in a Cube,” earlier in this appendix.</td>
</tr>
</tbody>
</table>

### A.7 `<level>`

The `<level>` element specifies a level within the given hierarchy. It has the following contents:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes in a Cube” earlier in this appendix.</td>
</tr>
</tbody>
</table>
### Attribute | Purpose
--- | ---
sourceProperty, sourceExpression | Specify one of these attributes. The syntax is almost the same as you would use for Property or Expression in the Architect; see “Defining the Source Values for a Dimension or Level.” Notes:
- For sourceProperty, use the same value you would enter for Property, enclosed within single quotes.
  
  For example: `sourceProperty='Age'`
- For sourceExpression, use the same value you would enter for Expression, enclosed within single quotes.
  
  For example: `sourceExpression='%source.property"ABC"'`

You can enclose the value in double quotes instead if the value itself does not contain any double quotes. For example:

`sourceExpression="%source.property"

For the requirements for levels within a time or age dimension, see “Defining a Time Level” and “Defining an Age Level.”

For levels in an iKnow dimension, sourceProperty and sourceExpression are ignored, because these levels use a different mechanism to specify the source values. See “Using Unstructured Data in Cubes.”

timeFunction | Only for levels within a time or age dimension. See “Defining a Time Level” and “Defining an Age Level.”
timeOffset | (Optional) Only for levels within a time dimension. See “Specifying a Date Offset.”
timeFormat | (Optional) Only for levels within a time dimension. See “Specifying a Member Name Format.”
list | (Optional) If this attribute is "true" then the source value is expected to be a list, and each item in the list becomes a member of this level. The default is "false".

By default, the list is expected to be in **$LIST** format. To use a string consisting of a character-delimited list, specify listDelimiter.

A list-based level cannot have properties, parent levels, or child levels.

listDelimiter | (Optional) Specifies the delimiter used to separate items in the list that is used as the source data for the level. Use this if the list is a character-separated list.
nullReplacement | (Optional) Specifies the string to use as the member name if the source data for this level is null. For example, specify `nullReplacement="No City"`.
rangeExpression | (Optional) For numeric data, this specifies how to assign numeric values to bins (each bin is a member of this level). For other data, this specifies replacement values. This attribute also controls the default order of the members of this level. See the subsections “Defining a Basic Range Expression” and “Defining a Compact Multi-range Expression.”
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>useDisplayValue</td>
<td>(Optional) For properties that have values for the <code>DISPLAYLIST</code> and <code>VALUENAME</code> parameters, this attribute specifies which value to include in the index. If this attribute is &quot;false&quot; (the default), the system uses the value given by <code>VALUENAME</code>; if this attribute is &quot;true&quot;, the system uses the value given by <code>DISPLAYLIST</code>. This option is ignored if you specify <code>linkClass</code> and <code>linkProperty</code>.</td>
</tr>
<tr>
<td>sort</td>
<td>(Optional) Specifies how to sort members of this level by default. For a level in a time dimension, specify &quot;asc&quot; or &quot;desc&quot;. For a level in a data dimension, specify &quot;asc&quot;, &quot;asc numeric&quot;, &quot;desc&quot;, or &quot;desc numeric&quot;.</td>
</tr>
<tr>
<td>linkClass, linkProperty</td>
<td>(Optional) See “Linking to Another Table.”</td>
</tr>
<tr>
<td>factName</td>
<td>(Optional) Name, in the generated fact table, for the column that corresponds to this level. If this attribute is null, DeepSee generates a name.</td>
</tr>
<tr>
<td>dependsOn</td>
<td>(Optional) Specifies the level on which this level has a dependency. Specify the full MDX identifier of the level. Or specify a comma-separated list of MDX level identifiers. See “Defining Dependencies Between Levels in Different Hierarchies.” This attribute is completely unrelated to the DependsOn compiler keyword.</td>
</tr>
<tr>
<td>factSelectivity</td>
<td>(Optional, not used by DeepSee) Value to override the generated <code>SELECTIVITY</code> parameter for the property in the fact table class. DeepSee queries do not use this parameter. This option is intended for cases where SQL is used directly against the generated fact table and the default <code>SELECTIVITY</code> needs to be overridden. Specify a positive value less than or equal to 1. For details, see <code>SetFieldSelectivity()</code> in <code>%SYSTEM.SQL</code>.</td>
</tr>
</tbody>
</table>

For example:

```
<level name="ZIP" sourceProperty="HomeCity.PostalCode" />
```

For another example, the following level definition also defines a property for that level:

```
<level name="City" sourceProperty="HomeCity.Name">
  <property name="Population" sourceProperty="HomeCity.Population" />
</level>
```

## A.7.1 Linking to Another Table

Within a `<level>`, `<measure>`, or `<property>` element, the `linkClass` and `linkProperty` attributes enable you to use a property in another class, a class that you cannot access via Caché dot syntax. To use the link feature, you must be able to look up a record in the other class by the ID of the record.

The feature works as follows:

- Specify `sourceProperty` or `sourceExpression` as a value that gives the ID of the desired record in the other class.
• Specify linkClass as the complete package and class name for the other class.

• Specify linkProperty as the property in the other class on which you want to base the level that you are defining.

For example, you could add the following to the Hole Foods sample:

```xml
<level name="Product" sourceProperty="Product" linkClass="HoleFoods.Product" linkProperty="Name" />
```

If the external class linkClass and a property linkProperty are defined, DeepSee runs a query against the external class to fetch the value of the given property from that class, for the record whose ID equals the given sourceProperty or sourceExpression.

### A.7.2 Specifying a Date Offset

In some cases, you may need a time level to match a corporate financial calendar that does not start on 1 January. For example, in many companies, the financial year starts 1 Oct. Consider the following pivot table:

<table>
<thead>
<tr>
<th>Year</th>
<th>Amount Sold</th>
</tr>
</thead>
<tbody>
<tr>
<td>FY 2005</td>
<td>7,131.04</td>
</tr>
<tr>
<td>FY 2006</td>
<td>12,892.19</td>
</tr>
<tr>
<td>FY 2007</td>
<td>14,653.86</td>
</tr>
<tr>
<td>FY 2008</td>
<td>16,540.96</td>
</tr>
<tr>
<td>FY 2009</td>
<td>18,987.04</td>
</tr>
<tr>
<td>FY 2010</td>
<td>16,666.45</td>
</tr>
</tbody>
</table>

In this case, the member FY 2005 consists of records of sales between 1 October 2004 and 30 September 2005, inclusive.

To create a level like this, specify the timeOffset and timeFormat attributes of the level (which must be within a time-type dimension). This section discusses the timeOffset attribute; the next section discusses timeFormat.

The timeOffset attribute specifies an amount of time to add to the source values used in this level; this amount of time can be negative or positive. The system uses this at cube build time.

For timeOffset, specify an amount of time by specifying a string of the following form:

```
#y#m#d
```

Where # is a number, #y represents an amount of time in years, #m represents an amount of time in months, and #d represents an amount of time in days. If you omit an element, DeepSee uses zero in its place. For example, the string 3m15d represents three months and 15 days.

The most common value for timeOffset is -3m, which you use if the fiscal year starts in 1 October of the previous year. If timeOffset="-3m", DeepSee subtracts three months to each time value used in this level. For example, for this level, the date 1 Jan 2010 is converted to 1 Oct 2009.

Other levels are unaffected, even within the same dimension. This means that you can also define more granular levels that display the actual dates. For an example, see “Handling a Calendar That Has a Date Offset.”

### A.7.3 Specifying a Member Name Format

For a level within a time-type dimension, you can use the timeFormat attribute to specify the format of the display names of the members of the level. DeepSee applies this formatting at query execution time. There is no effect on how values are stored or indexed.
For the `timeFormat` attribute, you specify a string that consists of the following case-sensitive date pieces and other pieces:

<table>
<thead>
<tr>
<th>Piece</th>
<th>Replaced By</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>Year number</td>
</tr>
<tr>
<td>q</td>
<td>Quarter number</td>
</tr>
<tr>
<td>mmmmm</td>
<td>Full month name</td>
</tr>
<tr>
<td>mmm</td>
<td>Short month name</td>
</tr>
<tr>
<td>mm</td>
<td>Month number with leading 0 if needed</td>
</tr>
<tr>
<td>m</td>
<td>Month number without a leading 0</td>
</tr>
<tr>
<td>ddddd</td>
<td>Full day name</td>
</tr>
<tr>
<td>ddd</td>
<td>Short day name</td>
</tr>
<tr>
<td>dd</td>
<td>Day number of the month with leading 0 if needed</td>
</tr>
<tr>
<td>d</td>
<td>Day number of the month without a leading 0</td>
</tr>
<tr>
<td>\x</td>
<td>x</td>
</tr>
<tr>
<td>period (.), slash (/), hyphen (-), space</td>
<td>Unchanged</td>
</tr>
<tr>
<td>other characters</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

All names are based on the current server locale.

**Important:** For time levels, member display names must be unique. Also, a member can have exactly one display name. These rules mean that not all the preceding date pieces are appropriate for all time levels.

The following table lists the date pieces that are suitable for different time levels.

<table>
<thead>
<tr>
<th>timeFunction Used By Level</th>
<th>Suitable Date Pieces</th>
<th>Default timeFormat</th>
<th>Other Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year</td>
<td>y</td>
<td>y (2004)</td>
<td>\F\Y y (FY 2004) \F\Y (FY2004)</td>
</tr>
<tr>
<td>QuarterYear</td>
<td>y, q</td>
<td>\Qq y (Q3 2004)</td>
<td>\Qq \F\Y y (Q1 FY 2004) \Qq \F\Y y (Q1 FY2004)</td>
</tr>
<tr>
<td>QuarterNumber</td>
<td>q</td>
<td>\Qq (Q3)</td>
<td></td>
</tr>
<tr>
<td>MonthYear</td>
<td>y, mmmmm, mmm, mm, m</td>
<td>mmmmm y (February 2004)</td>
<td>mmm y (Feb 2004) y-mm (2004–02) mm/yy (02/2004) m/yy (2/2004) \F\Y-yy (FY2004-02)</td>
</tr>
<tr>
<td>MonthNumber</td>
<td>mmmmm, mmm, mm, m</td>
<td>mmmmm (February)</td>
<td>mmm (Feb) mm (02) m (2)</td>
</tr>
<tr>
<td>timeFunction Used By Level</td>
<td>Suitable Date Pieces</td>
<td>Default timeFormat</td>
<td>Other Examples</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------------</td>
<td>-------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>DayMonthYear</td>
<td>y, mmmm, mmm, mm, m, dddd, ddd, dd, d</td>
<td>mmmm dd y (Feb 1 2004)</td>
<td>y-mm-dd (2010-02-03)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mm/dd/y (02/03/2010)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>m/d/y (2/3/2010)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>dddd, mmmm dd y (Wednesday, February 03 2010)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ddd, mmm dd y (Wed, Feb 03 2010)</td>
</tr>
</tbody>
</table>

### A.7.4 Defining a Basic Range Expression

For `rangeExpression`, the basic syntax is as follows:

```
value_or_range:new_value;value_or_range:new_value; ... ;
```

Here `new_value` is a single value, which DeepSee treats as a string and uses as the name of a level member. Also, `value_or_range` is any of the following:

- A single value such as 5 or Louisiana
- A range in the form `(value1,value2)`
- A range in the form `[value1,value2]`
- A range in the form `(value1,value2]`
- A range in the form `[value1,value2)`

In these expressions, `value1` and `value2` are numeric values or null (that is, omitted from the expression). The left and right parentheses indicate that the range is not inclusive at the given endpoint. The left and right square brackets indicate that the range is inclusive at the given endpoint. For example, `(45,49]` represents all values greater than 45 and less than or equal to 49.

### A.7.5 Defining a Compact Multi-range Expression

In some cases, it is necessary to define many ranges for a given level, which can be a tedious process. Also the resulting range expression (when viewed in Studio) is difficult to read. In such cases, you may want to use the following alternative syntax for the value of `rangeExpression`:

```
rangeExpression="[start:increment:end]:replacement;"
```

Or use the appropriate parenthesis in place of either square bracket (details below).

This syntax generates a series of ranges. Here, `start` is the numeric start value of the first range, `end` is the numeric end value of the last range, and `increment` is the numeric value by which the ranges are defined. The first range extends from `start to start plus increment`, and so on.

Also, `replacement` is an expression that is used as the replacement value. Within `replacement`, you can use the following elements:

- `%1` — is replaced with the start value of the range
%2 — is replaced with the end value of the range

$eval(expression) — causes the system to evaluate the contained expression. For example: $eval (%2-1) returns the end value of the range minus one

The square brackets or parentheses affect how the start and end values are treated, in the generated start and end ranges:

- A square bracket causes the value to be included in the range.
- A parenthesis causes the value to be excluded from the range.

At the intermediate range boundaries, the boundary value is always assigned to the upper range, rather than the lower range. That is, for intermediate ranges, the opening bracket is always [ and the closing bracket is always ).

Consider the following range expression:

rangeExpression="[0:30:90]:%1 to $eval(%2-1);"

This generates the same members as the following longer form:

rangeExpression="[0,30):0 to 29;[30,60):30 to 59;[60,90):60 to 90;"

If the value end - start is not an integer multiple of increment, the last range extends beyond end. For example, consider the following range expression:

rangeExpression="[0:30:100]:%1 to $eval(%2-1);"

This generates the same members as the following longer form:

rangeExpression="[0,30):0 to 29;[30,60):30 to 59;[60,90):60 to 90;[90,119):90 to 119;"

A.8 <member>

The <member> element is primarily for advanced use. See “Manually Specifying the Members of Level” and “Defining Computed Dimensions,” earlier in this book.

The <member> element has the following contents:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes in a Cube” earlier in this appendix.</td>
</tr>
<tr>
<td>spec</td>
<td>If the level is contained in a dimension that has type=&quot;computed&quot;, specify the SQL query that returns the IDs of the fact table rows (or the source table rows) that this member uses. In other cases, optionally specify the value to use as the member key. If you omit this, name becomes the member key.</td>
</tr>
</tbody>
</table>
A.9 <property>

A level may contain zero or more custom level properties. These are properties whose value is derived from the source data and is associated with a specific member of a level. For example, the city level could include properties such as population or ZIP code. For each city, there would be one value of each of these properties.

The <property> element has the following contents:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes in a Cube,” earlier in this appendix.</td>
</tr>
<tr>
<td>sourceProperty</td>
<td>Specify one of these attributes in almost the same way that you would specify Property or Expression in the Architect; see “Specifying the Source Values for a Dimension or Level” and “Details for Source Expressions.” Notes:</td>
</tr>
<tr>
<td>sourceExpression</td>
<td>• For sourceProperty, use the same value you would enter for Property, enclosed within single quotes.</td>
</tr>
<tr>
<td></td>
<td>• For example: sourceProperty='MyProp'</td>
</tr>
<tr>
<td></td>
<td>• For sourceExpression, use the same value you would enter for Expression, enclosed within single quotes.</td>
</tr>
<tr>
<td></td>
<td>• For example: sourceExpression='%source.MyProp_'&quot;ABC&quot;'</td>
</tr>
<tr>
<td></td>
<td>You can enclose the value in double quotes instead if the value itself does not contain any double quotes. For example: sourceExpression='%source.MyProp'</td>
</tr>
<tr>
<td>sort</td>
<td>(Optional) Specifies how to use this property to sort members of the level that contains this property. You can sort a level by multiple properties; if you do, the sorting is applied in the order in which you define the &lt;property&gt; elements; the first property controls the primary sort, the second property controls the secondary sort, and so on. Specify &quot;asc&quot;, &quot;asc numeric&quot;, &quot;desc&quot;, or &quot;desc numeric&quot;</td>
</tr>
<tr>
<td></td>
<td>By default, the property does not affect the sort order of the members.</td>
</tr>
<tr>
<td>isName</td>
<td>(Optional) If &quot;true&quot;, this attribute specifies that DeepSee should use the values of this property to specify the names of the members of the dimension that contains this property. Specify either &quot;true&quot; or &quot;false&quot; (the default).</td>
</tr>
<tr>
<td>isReference</td>
<td>(Optional) If &quot;true&quot;, this attribute specifies that DeepSee should not store the property value, but should instead define the property as an SQL computed field that refers to the original source tables.</td>
</tr>
<tr>
<td>useDisplayValue</td>
<td>(Optional) For class properties that have values for the DISPLAYLIST and VALUELIST parameters, this attribute specifies which value to use for the property. If this attribute is &quot;true&quot; (the default), the system uses the value given by DISPLAYLIST; if this attribute is &quot;false&quot;, the system uses the value given by VALUELIST.</td>
</tr>
</tbody>
</table>
PurposeAttribute  
(Optional) See “Linking to Another Table.”

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>linkClass,</td>
<td>(Optional) Name used (in the generated dimension table) for the column that corresponds to this property. If this attribute is null, DeepSee generates a name. You do not need to worry about this unless you plan to issue SQL queries directly against the generated dimension table. Despite the name of this attribute, properties are not contained in the fact table. They are contained in the table that corresponds to the level to which the property belongs.</td>
</tr>
<tr>
<td>linkProperty</td>
<td></td>
</tr>
<tr>
<td>factName</td>
<td>(Optional) Name used (in the generated dimension table) for the column that corresponds to this property. If this attribute is null, DeepSee generates a name. You do not need to worry about this unless you plan to issue SQL queries directly against the generated dimension table. Despite the name of this attribute, properties are not contained in the fact table. They are contained in the table that corresponds to the level to which the property belongs.</td>
</tr>
<tr>
<td>formatString</td>
<td>(Optional) Controls how the values are displayed. See “formatString Details,” earlier in this appendix. In the format string, you can also use the special character 0, which serves as a placeholder for leading zeros. For example, 00000 would show a five-digit number padded with leading zeros.</td>
</tr>
<tr>
<td>hidden</td>
<td>(Optional) Not currently used.</td>
</tr>
</tbody>
</table>

For example:

```xml
<property name="Population" sourceProperty="City.Population"/>
```

### A.9.1 Intrinsic Properties

For each level, DeepSee also automatically defines a set of intrinsic properties. These are listed in the *DeepSee MDX Reference*.

### A.10 <listing>

A cube can contain zero or more named listings. These are available in the Analyzer.

The default listing for a cube is either the listing specified in the `defaultListing` attribute for the `<cube>` element (if specified) or the first `<listing>` element contained in the `<cube>`.

The `<listing>` element has the following contents:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name,</td>
<td>See “Common Attributes in a Cube” earlier in this appendix.</td>
</tr>
<tr>
<td>displayName,</td>
<td></td>
</tr>
<tr>
<td>description,</td>
<td></td>
</tr>
<tr>
<td>disabled</td>
<td></td>
</tr>
<tr>
<td>fieldList</td>
<td>Comma-separated list of fields to display. For options, see “Specifying a Simple Listing,” earlier in this book. For options in a data connector listing, see “Specifying a Data Connector Listing.”</td>
</tr>
<tr>
<td>Attribute</td>
<td>Purpose</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| formatList | List (delimited by carets) of CSS formatting instructions for the listing. Each piece of the list is applied to the corresponding column in the listing. This list must have the same number of elements as there are columns, and each list element must have one of the following forms:  
  - `{CSS formatting instruction}`, for example, `{text-align:center;}` — Formats the corresponding column.  
  You can use `{text-align:left;}` or `{text-align:center;}`. Other instructions are ignored.  
  - `display:none;` — Hides the corresponding column.  
  - `Null` — Displays the corresponding column in the default style. |
| orderBy   | Applies only to simple listings. This attribute is a comma-separated list of fields by which to sort the listing. The overall sort is controlled by the first field in the list, the secondary sort is controlled by the second field, and so on.  
After a field name, you can include the ASC or DESC keyword to sort in ascending or descending order, respectively. For additional options, see “Specifying a Simple Listing,” earlier in this book.  
This attribute is ignored if the source is a data connector. |
| sourceClass | Specifies the data connector, if any, on which this listing is based. If you specify this attribute, specify the name of a class that extends `%DeepSee.DataConnector`. See the DeepSee Implementation Guide. |
| sql       | Specifies the custom SQL query, if any, for this listing. See “Specifying a Custom Listing,” earlier in this book.                                                                                       |
| listingType | (Optional) Specifies the format of the listing. The default is "table". If you instead specify "map" the system displays a map-type listing, which displays a map with points marked by latitude and longitude. In this case, the listing query must contain the fields Latitude and Longitude (case-sensitive). See “Defining a Map-type Listing,” earlier in this book. |
| resource  | (Optional) Name of the resource used to control access to this listing. See the DeepSee Implementation Guide.                                                                                          |

There are three kinds of listings, considering their sources:
• Basic listings, which directly use the source table used by the cube
• Data connector listings (which use data connectors)
• Custom listings (which use custom queries)

In all cases, the system creates and uses an SQL query.

The following table indicates which attributes of `<listing>` you specify for these kinds of listings:

<table>
<thead>
<tr>
<th>Kind of Listing</th>
<th>fieldList</th>
<th>orderBy</th>
<th>sourceClass</th>
<th>sql</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>Required</td>
<td>Optional</td>
<td>Do not specify; this attribute would take precedence.</td>
<td>Do not specify; this attribute would take precedence.</td>
</tr>
<tr>
<td>Data connector</td>
<td>Optional</td>
<td>Ignored</td>
<td>Required</td>
<td>Ignored</td>
</tr>
<tr>
<td>Custom</td>
<td>Ignored</td>
<td>Ignored</td>
<td>Do not specify; this attribute would take precedence.</td>
<td>Required</td>
</tr>
</tbody>
</table>

### A.11 `<calculatedMember>`

A cube can contain zero or more calculated members. A calculated member is a member that is defined in terms of other members. You can add two kinds of calculated member:

• You can define a new measure that is based on other measures. For example, you can define a measure via a formula like the following:

```
Measure 3 = (Measure 1 + Measure 2) / Measure 2
```

This is not the exact syntax.

• You can define a new member that is based on other non-measure members. For example, you could create a `Primary Colors` member that combines the red, yellow, and blue members of the Favorite Color dimension.

The new `Primary Colors` member refers to all the records of the fact table that correspond to the red, yellow, or blue members.

In MDX, a measure is considered to be a member, which is why both kinds of calculated elements are considered to be calculated members.

**Note:** When you use this element to define a measure, the `cube` command in the MDX shell does not currently list this measure. You can, however, use the set in the MDX shell or in the query API.

The `<calculatedMember>` element has the following contents:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes in a Cube” earlier in this appendix. For the name attribute, DeepSee does not check whether the name is already in use. If you use a name that was previously used by another member of this dimension, you are overriding that member.</td>
</tr>
<tr>
<td>dimension</td>
<td>Dimension to which this member belongs.</td>
</tr>
</tbody>
</table>
PurposeAttribute

MDX expression that defines the values for this member in terms of references to other members. For simple but common scenarios, see “Defining Calculated Elements” in Using the DeepSee Analyzer. For details and examples, see the DeepSee MDX Reference.

FormatString

(Optional) Controls how the values are displayed. See “Specifying a Format String.”

Units

(Optional) Indicates the units in which the measure value is expressed. Currently this attribute is provided only for general information.

For example:

```
<calculatedMember name="Avg Age" dimension="MEASURES"
valueExpression="[MEASURES].[Age]/[MEASURES].[%COUNT]"/>
```

In any context where you use this calculated member, DeepSee first evaluates the Age and Patient Count measures in that context and then performs the division.

**Note:** You can define a calculated member in two other ways:

- Within an MDX query by using the WITH clause.
- Within the CREATE MEMBER statement.
  
  This is useful only within the MDX Shell.

  See the DeepSee MDX Reference.

### A.12 <namedSet>

A cube can contain zero or more named sets. A named set is an alias for a member or for a set of members. You can use named sets for rows or columns in your queries, and DeepSee substitutes the member or the set when you run the query.

**Note:** When you define a named set, the cube command in the MDX shell does not list this set. You can, however, use the set in the MDX shell or in the query API.

The <namedSet> element has the following contents:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name, DisplayName, Description, Disabled</td>
<td>See “Common Attributes in a Cube” earlier in this appendix. For the name attribute, DeepSee does not check whether the name is already in use. If you use a name that was previously used by another named set, you are overriding that named set.</td>
</tr>
<tr>
<td>SetExpression</td>
<td>MDX expression that returns a member or a set of members. For details and examples, see the DeepSee MDX Reference.</td>
</tr>
</tbody>
</table>

For example:

```
<namedSet name="SampleSet" setExpression="[homed].[h1].[city].MEMBERS" />
```
You can define a named set in two other ways:

- Within an MDX query by using the WITH clause.
- Within the CREATE SET statement.

See the DeepSee MDX Reference.

### A.13 <relationship>

To define a one-way relationship from one cube to another cube, you define a `<relationship>` element in the first cube. To define a two-way relationship, you define two complementary `<relationship>` elements, one in each cube.

The `<relationship>` element has the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the relationship. You use this logical name in MDX queries to use levels of the other cube. Typically this is the name of the other cube.</td>
</tr>
<tr>
<td>displayName, description, disabled</td>
<td>See “Common Attributes in a Cube” earlier in this appendix.</td>
</tr>
<tr>
<td>relatedCube</td>
<td>Logical name of the other cube.</td>
</tr>
<tr>
<td>inverse</td>
<td>Value of the name attribute in the <code>&lt;relationship&gt;</code> element in the other cube.</td>
</tr>
<tr>
<td>cardinality</td>
<td>Cardinality of the relationship.</td>
</tr>
<tr>
<td>sourceProperty</td>
<td>See “Defining a One-way Relationship” and “Defining a Two-way Relationship”</td>
</tr>
<tr>
<td>sourceExpression</td>
<td>(Optional) Specifies the string to use as the member name if the source data for this relationship is null. For example, specify nullReplacement=&quot;No City&quot;. Specify this attribute within a <code>&lt;relationship&gt;</code> that specifies sourceProperty or sourceExpression.</td>
</tr>
<tr>
<td>nullReplacement</td>
<td>(Optional) Name used (in the fact table) for the column that corresponds to this relationship. If this attribute is null, the system generates a name.</td>
</tr>
<tr>
<td>factName</td>
<td>Do not use. These are ignored.</td>
</tr>
<tr>
<td>linkClass, linkProperty</td>
<td>(Optional) Specifies the relationship on which this relationship has a dependency. Specify the logical name of the relationship. See “Defining Dependencies Between Levels in Different Hierarchies.” (Or, if the relationship depends upon a level, specify the MDX identifier of that level.) This attribute is completely unrelated to the DependsOn compiler keyword.</td>
</tr>
</tbody>
</table>
Important: When you compile the cube classes, first compile the independent cube, which is the one that does not define a source property or source expression for the relationship. To control the compilation order, specify the DependsOn keyword in the class definition of the dependent cube.

Similarly, you must build the independent cube first. The DependsOn keyword has no effect on the order in which cubes are built.

The following table summarizes which keywords to specify in each of scenarios:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>In a One-way Relationship...</th>
<th>In a Two-way Relationship...</th>
</tr>
</thead>
<tbody>
<tr>
<td>inverse</td>
<td>Omit this</td>
<td>Specify this in both cubes</td>
</tr>
</tbody>
</table>
| cardinality                    | Use "one"                    | • In the dependent cube, use "one"
|                                |                              | • In the independent cube, use "many" |
| sourceProperty or sourceExpression | Specify as usual            | Applicable only in the dependent cube |
| factName                       | Specify as usual             | Applicable only in the dependent cube; ignored in the other cube |

For examples, see “Defining Cube-Cube Relationships,” earlier in this book.

A.14 <index>

The <index> element specifies an optional, custom index that you add to the fact table. DeepSee does not use this index (because it automatically adds the indices that it needs).

You can add a custom index if you plan to access the fact table via SQL. The <index> element has the following contents:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name,</td>
<td>See “Common Attributes in a Cube” earlier in this appendix.</td>
</tr>
<tr>
<td>displayName,</td>
<td></td>
</tr>
<tr>
<td>description,</td>
<td></td>
</tr>
<tr>
<td>disabled</td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>Specifies the type of index. Use &quot;bitmap&quot;, &quot;bitslice&quot;, &quot;index&quot;, or &quot;key&quot;.</td>
</tr>
<tr>
<td>properties</td>
<td>Specifies the fields of the fact table on which this index is based. Specify a comma-separated list of properties of the fact table class.</td>
</tr>
</tbody>
</table>

For example:

<index name="IndexName" type="bitmap" properties="MxAge,DxGender"/>

Tip: Remember that you can specify the factName attribute for most levels and measures, to control the names of the properties in the generated fact table class.
A.15 <expression>

The <expression> element defines an optional expression that has a value for each row in the source table. Your cube can have any number of expressions.

The <expression> element has the following contents:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, description, disabled</td>
<td>See “Common Attributes in a Cube” earlier in this appendix.</td>
</tr>
<tr>
<td>displayName, factName, linkClass, linkProperty</td>
<td>Do not specify these attributes.</td>
</tr>
</tbody>
</table>
Reference Information for Subject Area Classes

A subject area is analogous to an SQL view of a table. A subject area is a subcube that enables users to focus on smaller sets of data without the need for multiple cubes. A subject area also enables you to customize captions and defaults of the cube.

This appendix contains reference information for subject area classes. It discusses the following topics:

- Basic requirements for a subject area class
- Common attributes in a subject area
  - `<subjectArea>`
  - `<measure>`
  - `<dimension>`
  - `<hierarchy>`
  - `<level>`
  - `<listing>`
  - `<calculatedMember>`
  - `<namedSet>`

Also see “Using Advanced Features of Cubes and Subject Areas,” earlier in this book.

You must recompile a subject area class after making any change.

B.1 Requirements for a Subject Area Class

To define a subject area, create a class that meets the following requirements:

- It must extend `%DeepSee.SubjectArea`.
- It must contain an XData block named `SubjectArea`.
- For this XData block, `XMLNamespace` must be specified as follows:

  ```xml
  XMLNamespace = "http://www.intersystems.com/subjectarea"
  ```
• The root element within the XData block must be `<subjectArea>` and this element must follow the requirements described in the rest of this appendix.

• It is useful, but not required, for the class to specify the DependsOn keyword so that this class is compiled only after the cube class is compiled and can be used.

• The class can define the `DOMAIN` parameter, which specifies the domain to which any localized strings belong. For example:

  Parameter DOMAIN = "PATIENTSAMPLE";

  For details, see the chapter “Performing Localization” in the *DeepSee Implementation Guide*.

For example:

```java
Class DeepSee.Model.SubjectAreas.AsthmaPatients Extends %DeepSee.SubjectArea
  [DependsOn=Cubes.StudyPatients]
{

  /// This XData definition defines the SubjectArea.
  XData SubjectArea [ XMLNamespace = "http://www.intersystems.com/deepsee/subjectarea" ]
  {
    <subjectArea name="AsthmaPatients"
      displayName="Asthma Patients"
      baseCube="Patients" filterSpec="diag.d1.diagnoses.asthma" >
    </subjectArea>
  }
}
```

Studio provides assistance as you type.

### B.2 Common Attributes in a Subject Area

Most of the elements in the subject area have the following attributes, which are listed here for brevity:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Logical name of the element as specified in the base cube on which this subject area is based.</td>
</tr>
<tr>
<td>displayName</td>
<td>(Optional) Localized name of this element for use in user interfaces. If you do not specify this attribute, the user interface instead displays the logical name. For details, see the chapter “Performing Localization” in the <em>DeepSee Implementation Guide</em>.</td>
</tr>
<tr>
<td>description</td>
<td>(Optional) Description of this element. If you do not specify this attribute, the user interface instead displays the description specified in the cube definition.</td>
</tr>
<tr>
<td>disabled</td>
<td>(Optional) Controls whether the compiler uses the override defined by this element. If this attribute is &quot;true&quot; then the compiler ignores this override; this is equivalent to commenting out the override. By default, this attribute is &quot;false&quot;. If the override is disabled, then DeepSee uses the definition as given in the cube.</td>
</tr>
</tbody>
</table>
B.3 <subjectArea>

The <subjectArea> element is the root element in the XData block in a subject area class. This element contains the following items:

<table>
<thead>
<tr>
<th>Attribute or Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes” earlier in this appendix.</td>
</tr>
<tr>
<td></td>
<td>Note that the name attribute must be unique within the Caché namespace.</td>
</tr>
<tr>
<td>baseCube</td>
<td>Logical name of the cube on which this subject area is based. (This can also be a comma-separated list of logical cube names; see “Defining Compound Cubes,” earlier in this book.)</td>
</tr>
<tr>
<td>owner</td>
<td>(Optional) Name of owner of subject area.</td>
</tr>
<tr>
<td>resource</td>
<td>(Optional) Name of the resource used to control access to this subject area, when it is accessed via the DeepSee Architect. See the DeepSee Implementation Guide.</td>
</tr>
<tr>
<td>filterSpec</td>
<td>(Optional) MDX set expression to use as a filter for this subject area. See the subsection “Filtering a Subject Area.” The default is an empty string so that there is no filtering.</td>
</tr>
<tr>
<td>caption</td>
<td>(Optional) Caption for this subject area. If you do not specify this, the system uses the caption of the base cube instead.</td>
</tr>
<tr>
<td>countMeasureCaption</td>
<td>(Optional) Caption to use for the default measure, which counts records. The default caption is Count. Internally, the name of the measure is %Count.</td>
</tr>
<tr>
<td>defaultListing</td>
<td>(Optional) Specify the logical name of the &lt;listing&gt; to use as the default in this subject area; see “&lt;listing&gt;.” If you do not specify this, the system uses the default listing as specified in the base cube.</td>
</tr>
<tr>
<td>defaultMember, defaultMeasure</td>
<td>Do not use.</td>
</tr>
</tbody>
</table>

B.3.1 Filtering a Subject Area

The filterSpec attribute enables you to specify a filter that applies to the subject area. This attribute must equal a valid MDX set expression. For example:

\{AgeD.H1.[10 to 19],AgeD.H1.[20 to 29]\}


Instead of (or in addition to) specifying filterSpec, you can implement the %OnGetFilterSpec callback; see the section “Customizing the Subject Area %OnGetFilterSpec Callback” earlier in this book.
### B.4 `<measure>`

Within a subject area, a `<measure>` element hides or customizes a measure. This element has the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes” earlier in this appendix.</td>
</tr>
<tr>
<td>hidden</td>
<td>(Optional) If hidden=&quot;true&quot; then the measure is defined and can be used in queries, but is not listed as an available measure in this subject area. The default is &quot;false&quot;.</td>
</tr>
<tr>
<td>formatString</td>
<td>(Optional) If specified, this attribute overrides the formatString attribute specified for this measure in the base cube. See “Specifying a Format String.”</td>
</tr>
</tbody>
</table>

**Note:** You cannot define overrides for calculated measures this way. A calculated measure is actually a calculated member; to define an override for it, you must use `<calculatedMember>`.

### B.5 `<dimension>`

Within a subject area, a `<dimension>` element hides or customizes a dimension. This element has the following contents:

<table>
<thead>
<tr>
<th>Attribute or Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes” earlier in this appendix.</td>
</tr>
<tr>
<td>hidden</td>
<td>(Optional) If hidden=&quot;true&quot; then the dimension is defined and can be used in queries, but is not listed as an available dimension. The default is &quot;false&quot;.</td>
</tr>
<tr>
<td>allCaption</td>
<td>(Optional) If specified, this attribute overrides the allCaption attribute specified for this dimension in the base cube.</td>
</tr>
<tr>
<td>allDisplayName</td>
<td>(Optional) If specified, this attribute overrides the allDisplayName attribute specified for this dimension in the base cube.</td>
</tr>
<tr>
<td><code>&lt;hierarchy&gt;</code></td>
<td>(Optional) You can include zero or more <code>&lt;hierarchy&gt;</code> elements, each of which can hide a hierarchy (or part of a hierarchy).</td>
</tr>
</tbody>
</table>

### B.6 `<hierarchy>`

Within a subject area, a `<hierarchy>` element hides or customizes a hierarchy. This element has the following contents:
### Attribute or Element | Purpose
---|---
name, displayName, description, disabled | See “Common Attributes” earlier in this appendix.

### <level>

(Optional) You can include zero or more `<level>` elements, each of which can hide a level.

### hidden

(Optional) If this attribute is "true" then the hierarchy is not available in this subject area. The default is "false".

### B.7 `<level>`

Within a subject area, a `<level>` element hides or customizes a level. This element has the following contents:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes” earlier in this appendix.</td>
</tr>
<tr>
<td>hidden</td>
<td>(Optional) If this attribute is &quot;true&quot; then the level is not visible in this subject area. The default is &quot;false&quot;.</td>
</tr>
<tr>
<td>sort</td>
<td>(Optional) For a level in a time dimension, this specifies how to sort members of this level by default. Specify either &quot;asc&quot; or &quot;desc&quot;. This attribute has no effect for levels in data or age dimensions.</td>
</tr>
</tbody>
</table>

### B.8 `<listing>`

Within a subject area, a `<listing>` element hides, customizes, or adds a listing. This element has the following contents:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes” earlier in this appendix.</td>
</tr>
<tr>
<td>hidden</td>
<td>(Optional) If this attribute is &quot;true&quot; then the listing is not visible in this subject area. The default is &quot;false&quot;.</td>
</tr>
<tr>
<td>other attributes</td>
<td>(Optional) See &quot;&lt;listing&gt;&quot; in the previous appendix. Specify these only if you are redefining a listing or adding a new listing.</td>
</tr>
</tbody>
</table>
B.9 `<calculatedMember>`

Within a subject area, a `<calculatedMember>` element hides or customizes a calculated member.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes in a Subject Area” earlier in this appendix.</td>
</tr>
<tr>
<td>dimension</td>
<td>Dimension to which this member belongs.</td>
</tr>
<tr>
<td>hidden</td>
<td>(Optional) If this attribute is &quot;true&quot; then the calculated member is not visible in this subject area. The default is &quot;false&quot;.</td>
</tr>
<tr>
<td>other attributes</td>
<td>(Optional) See “&lt;calculatedMember&gt;” in the previous appendix. Specify these only if you are redefining or adding a calculated member.</td>
</tr>
</tbody>
</table>

B.10 `<namedSet>`

Within a subject area, a `<namedSet>` element hides or customizes a named set. This element has the following contents:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes in a Subject Area” earlier in this appendix.</td>
</tr>
<tr>
<td>hidden</td>
<td>(Optional) If this attribute is &quot;true&quot; then the named set is not visible in this subject area. The default is &quot;false&quot;.</td>
</tr>
<tr>
<td>other attributes</td>
<td>(Optional) See “&lt;calculatedMember&gt;” in the previous appendix. Specify these only if you are redefining or adding a named set.</td>
</tr>
</tbody>
</table>
Reference Information for KPI and Plugin Classes

This appendix contains Reference Information for KPI and plugin classes. It discusses the following topics:

• Basic requirements for the class
• Common attributes in a KPI or plugin
• <kpi>
• <property>
• <filter>
• <action>

You must recompile a KPI class after making any change.

C.1 Basic Requirements

To define a KPI, create a class that meets the following requirements:

• It must extend %DeepSee.KPI.
• It must contain an XData block named KPI
• For this XData block, XMLNamespace must be specified as follows:
  XMLNamespace = "http://www.intersystems.com/kpi"
• The root element within the XData block must be &lt;KPI&gt; and this element must follow the requirements described in this appendix.
• The class can define several class parameters. See “Class Parameters for the KPI Class,” earlier in this book.

The requirements for a plugin are the same, with the following exceptions:

• The class must extend %DeepSee.KPIplugIn rather than %DeepSee.KPI.
• The class can define the PLUGINTYPE class parameter; see “Defining Plugins.”
C.2 Common Attributes in a KPI or Plugin

Most of the elements in a KPI or plugin have the following attributes, which are listed here for brevity:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Logical name of the element.</td>
</tr>
<tr>
<td>displayName</td>
<td>(Optional) Localized name of this element for use in user interfaces. If you do not specify this attribute, DeepSee instead uses the value specified by the <code>name</code> attribute. For details, see the chapter “Performing Localization” in the DeepSee Implementation Guide.</td>
</tr>
<tr>
<td>description</td>
<td>(Optional) Description of this element.</td>
</tr>
<tr>
<td>disabled</td>
<td>(Optional) Controls whether the compiler uses this element. If this attribute is &quot;true&quot; then the compiler ignores it. By default, this attribute is &quot;false&quot;</td>
</tr>
</tbody>
</table>

C.3 `<kpi>`

The `<kpi>` element is the root element in the XData block in a KPI or plugin class. This element contains the following items:

<table>
<thead>
<tr>
<th>Attribute or Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes in a KPI,” earlier in this appendix.</td>
</tr>
<tr>
<td>caption</td>
<td>Not used.</td>
</tr>
<tr>
<td>sourceType, mdx, sql</td>
<td>See “Specifying the Query for the KPI.”</td>
</tr>
<tr>
<td>rangeLower, rangeUpper, thresholdLower, thresholdUpper</td>
<td>See “Specifying Ranges and Thresholds for Speedometers.”</td>
</tr>
<tr>
<td>actionClass</td>
<td>(Optional) Specifies an associated KPI class that defines actions that are available to this KPI, in addition to the actions defined within this KPI. Specify the full package and class name of another KPI.</td>
</tr>
<tr>
<td><code>&lt;property&gt;</code></td>
<td>(Optional) You can include zero or more <code>&lt;property&gt;</code> elements, each of which corresponds to a column of the query that the KPI uses. You cannot display the query results unless you specify <code>&lt;property&gt;</code> elements.</td>
</tr>
<tr>
<td><code>&lt;action&gt;</code></td>
<td>(Optional) You can include zero or more <code>&lt;action&gt;</code> elements, each of which is available for use when you create a scorecard based on this KPI.</td>
</tr>
<tr>
<td><code>&lt;filter&gt;</code></td>
<td>(Optional) You can include zero or more <code>&lt;filter&gt;</code> elements, each of which is available for use when you create a scorecard based on this KPI.</td>
</tr>
</tbody>
</table>
C.4 <property>

Within <kpi>, a <property> element contains the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes in a KPI,” earlier in this appendix. Note that the name or displayName of a KPI property is used as the default caption in any meter widget that displays this property; it is convenient to specify a name that is suitable for that use.</td>
</tr>
<tr>
<td>columnNo</td>
<td>Number of the column in the query that contains the data for this property. The first data column is 1.</td>
</tr>
<tr>
<td>format</td>
<td>(Optional) Default numeric format for this property, when this KPI is displayed in a pivot table widget. For example: format=&quot;##.###&quot;</td>
</tr>
<tr>
<td>style</td>
<td>(Optional) CSS style to apply to the cells that display this property, when this KPI is displayed in a pivot table widget. For example: style=&quot;color:red&quot;</td>
</tr>
<tr>
<td>headerStyle</td>
<td>(Optional) CSS style to apply to the corresponding header cells, when this KPI is displayed in a pivot table widget. For example: style=&quot;color:red;font-style:italic&quot;</td>
</tr>
<tr>
<td>defaultValue</td>
<td>Not used.</td>
</tr>
</tbody>
</table>

Note that the KPI test page ignores the format, style, and headerStyle attributes. These attributes affect only pivot table widgets.

C.5 <filter>

Within <kpi>, a <filter> element contains the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes in a KPI,” earlier in this appendix.</td>
</tr>
<tr>
<td>filterProperty</td>
<td>Logical name of the &lt;property&gt; element that this filter controls. This option is for use when retrieving the filter values programmatically.</td>
</tr>
<tr>
<td>defaultValue</td>
<td>(Optional) Default value for this filter. Note that you can also specify a default value for the filter when you include the KPI in a scorecard; that default value takes precedence over the defaultValue attribute of the &lt;filter&gt; element.</td>
</tr>
<tr>
<td>sql, valueList, displayList</td>
<td>See “Defining KPI Filters.”</td>
</tr>
<tr>
<td>searchType</td>
<td>(Optional) Specifies the kind of control to use when displaying this filter in a widget. The default control is a drop-down list. To display a calendar instead, specify this attribute as &quot;day&quot;</td>
</tr>
</tbody>
</table>
### C.6 `<action>`

Within `<kpi>`, an `<action>` element contains the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, displayName, description, disabled</td>
<td>See “Common Attributes in a KPI” earlier in this appendix. The <code>name</code> attribute cannot be any of the following (not case-sensitive): applyFilter, setFilter, refresh, showListing, viewDashboard, navigate, newWindow, rowCount, rowSort, colCount, colSort.</td>
</tr>
</tbody>
</table>

Note that the `<action>` element does not define the actions themselves, which you define within the `%OnDashboardAction()` callback method of the class; see the chapter “Defining the Available Actions” in the *DeepSee Implementation Guide*.

The `<action>` element is necessary to notify the Dashboard Designer which actions are defined in a KPI or plugin, so that you can select them for use.
Details for the Fact and Dimension Tables

Internally, DeepSee uses a star schema, which consists of the fact table and dimension tables, to which the fact table refers. You can directly query these tables, which DeepSee populates when you build a cube, and which DeepSee updates when changes occur. This chapter discusses the following:

- Fact table
- Dimension tables

Important: Do not redefine these tables or attempt to write data to them except via the authorized interfaces; see the chapter “Keeping the Cubes Current” in the DeepSee Implementation Guide.

When you compile a cube definition class, DeepSee also generates a table name Listing, in the same package as the fact table. Do not use the Listing table, which is for internal use only.

D.1 Fact Table

When you compile a cube definition class, DeepSee generates the corresponding fact table with the name Package_CubeClass.Fact where Package_CubeClass corresponds to the package and class of the cube definition, following the usual rules for translating package and class name to table names.

DeepSee updates the fact table whenever you build the cube or update it incrementally.

It is useful to examine this table, as well as the class definition that uses it.

The fact table contains the following fields:

- ID — ID of this row, assigned when the row is created.
- %partition — For future use. Ignore this field.
- %sourceID — The ID of the record in the base class on which this record is based. This field is a pointer to the source table.
- One field for each measure, to contain this measure value for this record. (There is one exception; see “Reuse of a Source Property in the Fact Table,” later in this appendix.)

This field stores the actual measure values. The fact table class defines bitslice indices on these values. For example:
Details for the Fact and Dimension Tables

/// Index for measure M1.
Index MxAge On MxAge [ Type = bitslice ];

/// Measure: MxAge
/// Source: Age
Property MxAge As %Integer;

• One field for each level, to indicate the level member to which this record belongs. (There is one exception; see “Reuse of a Source Property in the Fact Table,” later in this appendix.)

  – For levels in data dimensions, the fact table stores the ID of the level member, which is a pointer to a table that defines the members.

    The fact table class defines bitmap indices on these values. For example:

    /// Index for fact 1. Index DxGender On DxGender [ Type = bitmap ];
    /// Dimension: DxGender
    /// Source: Gender
    Property DxGender As Test.TestCube.DxGender;

    If this dimension is shared from another cube, then this pointer refers to a record in the applicable dimension table for that other cube. See “Defining a Shared Dimension,” earlier in this book.

  – For a time- or age-type level, the fact table stores an integer.

    The fact table class defines the corresponding property as a computed value and defines a bitmap index for it. For example:

    /// Index for fact 4. Index DxBirthDateFxYear On DxBirthDateFxYear [ Type = bitmap ];
    /// Dimension: DxBirthDateFxMonthYear
    /// Source: BirthDate Property DxBirthDateFxMonthYear As %Integer [ details omitted ];
    Property DxBirthDateFxYear As %Integer [ Calculated, SqlComputeCode = ... , SqlComputed ];

  – For a cube-to-cube <relationship>, if this cube specifies the sourceProperty or sourceExpression for the relationship, the fact table stores the ID of the corresponding row in the other fact table. That ID is a pointer to the other fact table.

• An additional field for each time-type or age-type dimension, to contain the complete value of the time or age dimension for this record. (There is one exception; see “Reuse of a Source Property in the Fact Table,” later in this appendix.)

  This value is in the form %DeepSee.Datatype.dateTime and is not indexed.

If you specified the dependsOn attribute for any levels, the fact table contains additional indices on the level combinations:

Index DxPostalCodeViaHomeCityANDd2642257510 On (DxPostalCodeViaHomeCity, DxD2642257510) [ Type = bitmap ];

If you defined any <index> elements in the <cube>, the fact table contains additional, custom indices. For example:

Index %UserIndexName On (MxAge, DxGender) [ Type = bitmap ];

These custom indices are for your own use; DeepSee does not use them.

D.1.1 Field Names

The following table summarizes how DeepSee determines the names for the measure, level, and dimension fields (in the case where you do not specify the factName attribute):
<table>
<thead>
<tr>
<th>Item and Scenario</th>
<th>Field Name in Fact Table (If Not Overridden by factName attribute)</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measure based on a source property</td>
<td>( M_{x}^{prop_name} ) where ( prop_name ) is the name of the property.</td>
<td>( M_{x}^{Age} )</td>
</tr>
<tr>
<td>Measure based on a source property in another table, via dot syntax (rare)</td>
<td>( M_{x}^{other_prop_nameVia_prop_name} ) where ( other_prop_name ) is the name of the property in the other class. However, if the resulting field name would be too long, DeepSee generates a unique number.</td>
<td>( M_{x}^{AgeViaOtherTable} )</td>
</tr>
<tr>
<td>Measure based on a source expression</td>
<td>( M_{x}^{nnnnnnnnnnT} ) where ( nnnnnnnnn ) is an integer and ( T ) indicates the measure type. (For example, ( I ) represents an integer measure.)</td>
<td>( M_{x}^{1968652733I} )</td>
</tr>
<tr>
<td>Data level based on a source property (with no range expression)</td>
<td>( D_{x}^{prop_name} )</td>
<td>( D_{x}^{Gender} )</td>
</tr>
<tr>
<td>Data level based on a source property with a range expression</td>
<td>( D_{x}^{prop_nameRgnnnnnnnnnn} ) where ( nnnnnnnnn ) is an integer.</td>
<td>( D_{x}^{AgeRg855025875} )</td>
</tr>
<tr>
<td>Data level based on a source property in another table, via dot syntax</td>
<td>( D_{x}^{other_prop_nameVia_prop_name} ) However, if the resulting field name would be too long, DeepSee generates a unique number.</td>
<td>( D_{x}^{PostalCodeViaHomeCity} )</td>
</tr>
<tr>
<td>Data level based on a source expression</td>
<td>( D_{x}^{nnnnnnnnnn} )</td>
<td>( D_{x}^{2163088627} )</td>
</tr>
<tr>
<td>Time- or age-type dimension</td>
<td>( D_{x}^{dim_name} ) where ( dim_name ) is the name of the dimension. (This field is for use by levels in this dimension.)</td>
<td>( D_{x}^{BirthDate} )</td>
</tr>
<tr>
<td>Time- or age-type level</td>
<td>( D_{x}^{dim_nameFxfunc_name} ) where ( func_name ) is the name specified in the timeFunction attribute for this level.</td>
<td>( D_{x}^{BirthDateFxYear} )</td>
</tr>
<tr>
<td>Relationship</td>
<td>( Rx^{generated_name} ), where ( generated_name ) is the name of the source property or a generated name based on the source expression.</td>
<td>( Rx^{MainCity} )</td>
</tr>
</tbody>
</table>

### D.1.2 Reuse of a Source Property in the Fact Table

If a cube contains multiple measures that use the same property (via sourceProperty), the fact table contains a field for only one of those measures (the last of those measures found in the cube definition). For example, suppose that the cube contained the following measure definitions:

```xml
<measure name="Age" sourceProperty="Age" aggregate="SUM" factName="AgeFact"/>
<measure name="Avg Age" sourceProperty="Age" aggregate="AVG" factName="AvgAgeFact"/>
```

These two measures are different only by how they aggregate across multiple records; the fact table would contain the same value for these measures, for any given record. For efficiency, the fact table would include the field AvgAgeFact, but not the field AgeFact.
The same logic applies when a cube contains multiple levels that use the same property. The logic also applies when a cube contains multiple age or time dimensions that use the same property.

For a given measure, level, or dimension, if you use `sourceExpression` and access the property via `%source.propertyname`, DeepSee always generates a separate field for that value.

### D.2 Dimension Tables

When you compile a cube definition class, DeepSee also generates a table for each level, other than the age- and time-type levels. These tables are in the same package as the fact table.

DeepSee updates the dimension tables whenever you build the cube or update it incrementally.

The dimension table for a level contains one row for each member of that level. The dimension tables are created dynamically as DeepSee processes records in the base table. For a given level, each time a new unique value is discovered, that value is added as a new row to the appropriate dimension table. This means that DeepSee automatically adds rows to the dimension tables when needed; no intervention is required.

#### D.2.1 Name of Dimension Table

If the cube definition specifies the `factName` attribute for the corresponding level, the applicable dimension table uses that value as its name.

Otherwise, the name of the dimension table has the following form:

```
Star_generated_name
```

Where `generated_name` is the corresponding field name in the fact table, without the `Dx` at the start. For example, suppose that in the fact table, the field name for the Home City level is `DxPostalCodeViaHomeCity`. In this case, the corresponding dimension table is named `StarPostalCodeViaHomeCity`.

#### D.2.2 Columns in a Dimension Table

The columns in this row are as follows:

- **ID** — ID of this row, assigned when the row is created.
- One column to contain the key for this member. The field name is the same as the column name field in the fact table that corresponds to this level; see the previous section.
- One column for each property of this level, to contain the actual property value for this member.
  
  The field name starts with `Dx` and is based either on the source property name or is generated as a unique number, as described earlier.

  If you use `linkClass` and `linkProperty` to define both the property and the level to which it belongs, the field names for the property and the level would be identical. In this scenario, DeepSee appends `_Link` to the end of the field name for the property.

- One column for the parent level of this level, to contain the ID of the parent of this member.
  
  The field name is the same as the column name in the fact table that corresponds to the parent level.

Depending on how the level is defined, you can find the member names as follows:

- By default, the key is used as the name, and the name is not stored separately.
• If the level includes a property that is defined with `isName="true"`, then the member names are stored in the column that contains that property (with one exception). The exception is when the property is also defined with `isReference="true"`; in this case, the field is computed at runtime.