Advanced Modeling for InterSystems IRIS Business Intelligence

Version 2024.1
2024-07-02

Advanced Modeling for InterSystems IRIS Business Intelligence
InterSystems IRIS Data Platform   Version 2024.1   2024-07-02
Copyright © 2024 InterSystems Corporation
All rights reserved.

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

1
Defining Computed Dimensions

Computed dimensions are a powerful but advanced InterSystems IRIS Business Intelligence model option that enable you to define members at runtime via queries. This page describes how to define them.

For a comparison of computed dimensions with more basic Business Intelligence model elements, see Summary of Model Options

Important: Computed dimensions do not have any association with calculated members. A computed dimension is specific to Business Intelligence. A calculated member is a standard concept in MDX.

1.1 Defining SQL Computed Dimensions

An SQL computed dimension contains members that are determined at runtime by SQL queries. You specify the query for each member.

Note: The null replacement option does not have any effect on computed dimensions.

1.1.1 Example Computed Dimensions

The Patients cube includes the `ComputedD` dimension, which defines two members (`member 1` and `member 2`); each of these members is defined by an SQL query that retrieves the IDs of specific patients. The following shows these members as seen in the left area of the Analyzer:

▼ **ComputedD**
    ▼ Computed
        🗋 member 1
        🗋 member 2
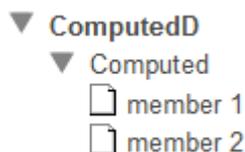
You can use this dimension, the `Computed` level, and the members in the same way that you use other elements of the cube.

Similarly, the HoleFoods cube includes the `Comments` dimension, which defines two members: `Complaints` and `Compliments`. These members are also defined by SQL queries.

1.1.2 Defining a SQL Computed Dimension

You can use the following procedure to define a SQL 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.

   When you have completed these changes, select **Save**.

   This step creates a shell that you can easily edit in an IDE. Also, the Architect initializes the display names to be the same as the names, which is convenient.

2. In an IDE, 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, the system 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">
   </level>
   ```

5. Just before the `</level>` line, add a line like the following:

   ```
   <member name="" displayName=""  spec="" />
   ```

   Edit this as follows:

Attribute	Value
name	Name of this member.
displayName	Optional display name of this member.
description	Optional text to display as a tooltip in the Analyzer, in the left area.
spec	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 Details for the Fact and Dimension Tables. Also see Variations for spec.

For example:

```
member name="example member 1" displayName="member 1"
spec="select ID from BI_Model_PatientsCube.Fact WHERE MxAge&lt;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.

1.1.3 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 BI_Model_PatientsCube.Fact WHERE MxAge&lt;50 AND DxHomeCity->DxHomeCity='Elm Heights'
    ```

    This query uses the fact table for the cube; for details, see 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 `$$$TABLE`, which is replaced by the actual name of the fact table. Unlike the previous query, this query is not valid in the Management Portal, which does not have information to convert the `$$$TABLE` token.

The following pivot table shows these members:

Computed	Avg Age	Patient Count
member 1	24.50	801
member 2	40	18

For example, in this case `member 1` consists of 801 patients whose average age is 24.50 years.

1.1.4 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 indexes 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, the system 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 `$$$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 `$$$FACT` to refer to this field. Be sure to include spaces around this token so that the system parses it appropriately.

  For an example, see the members in the `By Allergy Count` level in the example computed dimension later in this section.

- Instead of using a SELECT statement or a WHERE clause, you can a CALL statement that calls a stored procedure. For example:

  ```
  spec="CALL MyStoredProcedure()"
  ```

  For the stored procedure, be sure to specify both of the method keywords SqlProc and ReturnResultsets as true. For example:

  ```
  ClassMethod GetPatientIsDiabetic() As %Integer [ ReturnResultsets, SqlProc ]
  {
      // Note that %sqlcontext is created when SQLPROC is defined
      Try {
          Set rc = 1

          Set sql = "SELECT f.id id FROM HSAA_PatientCurrentConditionsCube.Fact f, HSAA.Diagnosis d
   "
                  _" WHERE d.Patient = f.%sourceid and d.Diagnosisgroup in ('249', '250', '253')"
          Set st = ##class(%SQL.Statement).%New()
          Set sc = st.%Prepare(sql)
          If ($$$ISERR(sc)) {
                  Set rc = 0
                  Quit
          }
          Set rs = st.%Execute()
          Do %sqlcontext.AddResultSet(rs)
      }
      Catch(ex) {
          Set %sqlcontext.%SQLCODE = ex.AsSQLCODE()
          Set %sqlcontext.%Message = ex.SQLMessageString()
          Set rc = 0
      }

      Quit rc
  }
  ```

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.

```
<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="select ID from $$$TABLE 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 $$$FACT > 2">
    </member>
    <member name="Not allergic"
    spec="select ID from $$$TABLE WHERE $$$FACT = 0">
    </member>
   </level>
  </hierarchy>
</dimension>
```

1.2 Defining MDX Computed Dimensions

An MDX computed dimension contains members that are determined at runtime by MDX queries. You specify the query for each member.

Note: The null replacement option does not have any effect on computed dimensions.

1.2.1 Defining an MDX Computed Dimension

You can use the following procedure to define an MDX 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 an IDE. Also, the Architect initializes the display names to be the same as the names, which is convenient.

2. In an IDE, 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 following to the `<dimension>` element:

        ```
        dimensionClass="MDX"
        ```

        The `dimensionClass` attribute refers to a helper class. If you do not specify a full package and class name, the system assumes that this class is in the package %DeepSee.ComputedDimension.

    For example:

    ```
    <dimension name="AgeBuckets" type="computed" dimensionClass="MDX">
        <hierarchy name="H1">
            <level name="Years"/>
        </hierarchy>
    </dimension>
    ```

4. Define members in either of the following ways:

    *   As a child of `<level>`, add `<member>` elements as in the following example:

        ```
        <member name="Boston" spec="[OUTLET].[CITY].[BOSTON]"/>
        ```

For an example of where to place the <member> element, see Defining a SQL Computed Dimension.

- Define the **%OnGetComputedMembers()** callback of the cube class. See the next subsection.

5. Save and recompile the cube class.

    As soon as you do so, the new dimension and its members are available for use.

1.2.2 %OnGetComputedMembers()

The **%OnGetComputedMembers()** callback has the following signature:

```
ClassMethod %OnGetComputedMembers(pDimName As %String,
                                  pHierName As %String,
                                  pLevelName As %String,
                                  ByRef pMemberList,
                                  pRollupKey As %String = "",
                                  ByRef pRange As %String = "") As %Status
```

Where *pDimName*, *pHierName*, and *pLevelName* are the dimension, hierarchy, and level name, respectively. The method should return, by reference, *pMemberList*. This is a multidimensional an array of the following form:

Array Node	Array Value
pMemberList(*i*) where *I* is an integer	A $LISTBUILD list that defines a member of the given level. This list must contain the following items, in order: • The MDX expression that defines the member • The display name of the member • The key of the member

The following shows an example:

Class Member

```
ClassMethod %OnGetComputedMembers(pDimName As %String, pHierName As %String, pLevelName As %String,
      ByRef pMemberList, pRollupKey As %String = "", ByRef pRange As %String = "") As %Status
{
    If (pDimName="AgeBuckets") {
        If (pLevelName="Years") {
            // $LB(MDX,Name,Key)
            Set pMemberList($I(pMemberList)) =
              $LB("%OR([DateOfSale].[Actual].[DaySold].[NOW-1y-1d]:[NOW])","1 year(s)","1")
            Set pMemberList($I(pMemberList)) =
              $LB("%OR([DateOfSale].[Actual].[DaySold].[NOW-2y-1d]:[NOW-1y])","2 year(s)","2")
            Set pMemberList($I(pMemberList)) =
              $LB("%OR([DateOfSale].[Actual].[DaySold].[NOW-3y-1d]:[NOW-2y])","3 year(s)","3")
            Set pMemberList($I(pMemberList)) =
              $LB("%OR([DateOfSale].[Actual].[DaySold].[NOW-4y-1d]:[NOW-3y])","4 year(s)","4")
            Set pMemberList($I(pMemberList)) =
              $LB("%OR([DateOfSale].[Actual].[DaySold].[NOW-5y-1d]:[NOW-4y])","5 year(s)","5")
            Set pMemberList($I(pMemberList)) =
              $LB("%OR([DateOfSale].[Actual].[DaySold].[NOW-6y-1d]:[NOW-5y])","6 year(s)","6")
            Set pMemberList($I(pMemberList)) =
              $LB("%OR([DateOfSale].[Actual].[DaySold].[NOW-7y-1d]:[NOW-6y])","7 year(s)","7")
        }
    }
    Quit $$$OK
}
```

2

Defining Shared Dimensions and Compound Cubes

This page describes how to define shared dimensions and compound cubes for use in Business Intelligence.

You can define compound cubes in the Architect, but for shared dimensions, you must use an IDE.

For background information, see Summary of Model Options.

Also see Accessing the BI Samples.

2.1 Overview

This section provides an overview of shared dimensions and compound cubes.

2.1.1 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.

2.1.2 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:

ZIP	Doctor Count	Patients Per Week	Avg Patient Allergy Count	Avg Patient Test Score
32006	5	600	1.05	73.65
32007	9	1,035	1.11	76.48
34577	12	1,473	0.98	75.15
36711	5	747	1.03	76.01
38928	9	987	1.14	75.44

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.

  The system 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:

Doctor Type	Doctor Count	Avg Patients Per Week	Patient Count	Patient Avg Age
Allergist	3	137.33	1,000	35.90
Anesthesiologist	1	148	1,000	35.90
Dermatologist	1	148	1,000	35.90
Emergency Physician	1	159	1,000	35.90
Gastroenterologist	4	153.75	1,000	35.90
General Physician	7	146	1,000	35.90
Internist	4	163	1,000	35.90
OB/GYN	6	141.50	1,000	35.90
Pediatrician	8	150.29	1,000	35.90
Radiologist	3	150.67	1,000	35.90
Surgeon	2	165.50	1,000	35.90

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.

2.2 Defining a Formally Shared Dimension

To share a dimension formally:

1. Define the dimension as usual in one cube definition.

   When that cube is built, the system 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, the system adds additional members for any levels, in the usual way.

2. Open the other cube definition in the Architect and add a shared dimension as follows:

   a. Select **Add Element**.

   b. Select **Shared Dimension**.

   c. In the first drop-down list, select the cube that defines the dimension.

   d. In the second drop-down list, select the dimension.

   e. Select **OK**.

   For all levels in this dimension, the fact table for this cube points to the dimension tables of the other cube.

3. Optionally, in the second cube, override the source data definition for the levels of the shared dimension.

   By default, the shared dimension uses the same source properties or source expressions that are used in the Dependent Cube. To override these, edit the class in an IDE, find the applicable `<dimension>` element, and add child `<hierarchy>` and `<level>` elements as needed; see Reference Information for Cube Classes. In this case, the dimension name, hierarchy names, and level names must be the same as in the other cube.

The following restrictions apply:

- The cube that owns the dimension must be built first. This process creates the tables for the dimensions defined in that cube. When you build the other cube, as the system processes records, it adds records to the dimension tables of any shared dimensions.

- Unless you override the source data definitions in the second cube, 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.

The HoleFoodsBudget and CompoundCube/Doctors cubes both contain examples of shared dimensions. These examples are not related to each other.

2.3 Defining an Informally Shared Dimension

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.

2.3.1 Examples

The Patients cube (in the class BI.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 BI.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`).

- The `ZIP` level uses the `City.PostalCode` source property (rather than `HomeCity.PostalCode`).

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 BI.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.

2.4 Defining Compound Cubes

To create compound cubes, you must use an IDE. 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 sample subject area `CompoundCube/CompoundCube`, **Base cube** is as follows:

  ```
  CompoundCube/Patients,CompoundCube/Doctors,CompoundCube/CityRainfall
  ```

  Also edit the **Depends On** option in the Details Area on the right. 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:

  ```
  <cube xmlns="http://www.intersystems.com/deepsee"
  name="Doctors"
  displayName="Doctors"
  sourceClass="BI.Study.Doctor"
  countMeasureName="DoctorCount"
  countMeasureCaption="Doctor Count">
  ...
  ```

  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.

2.4.1 Detail Listings for Compound Cubes

To define detail listings for a compound cube, define identical detail listings in all the participating cubes. The system generates an SQL UNION of these listings.

Note that the listings must be directly based on SQL; detail listings via data connectors will not work for compound cubes.

2.4.2 Example Compound Cube

To see an example of a compound cube, see the class BI.Model.CompoundCube.CompoundCube. This class is defined as follows:

Class Definition

```
Class BI.Model.CompoundCube.CompoundCube Extends %DeepSee.SubjectArea
[ DependsOn = (BI.Model.CompoundCube.Patients, BI.Model.CompoundCube.Doctors,
BI.Model.CompoundCube.CityRainfall) ]
{

/// 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 BI.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:

Dimension	CompoundCube/Patients cube	CompoundCube/Doctors cube	CompoundCube/CityRainfall cube
BirthD	✓		✓
DocD	✓	✓	
DocTypeD	✓	✓	
HomeD	✓	✓	✓

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:

✕ – + Pivot table using dimension shared by all cubes

ZIP-	City	Patient Count	Doctor Count	Avg Monthly Rainfall Inches
32006	Juniper	119	7	1.58
	Spruce	116	4	1.61
32007	Redwood	99	4	1.61
34577	Cypress	99	5	1.59
	Magnolia	125	2	1.63
	Pine	115	2	1.60
36711	Centerville	118	4	1.61
38928	Cedar Falls	101	6	1.59
	Elm Heights	108	6	1.59

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:

✕ – + Pivot table using dimension shared by two cubes

Decade	Patient Count	Doctor Count	Avg Monthly Rainfall Inches
1900s		40	1.61
1910s	7	40	1.61
1920s	19	40	1.63
1930s	60	40	1.63

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:

✕ – + Pivot table using dimension shared by two cubes

Doctor Type	Patient Count	Doctor Count	Avg Monthly Rainfall Inches
None	153		1.60
Allergist	44	2	1.60
Anesthesiologist	44	2	1.60
Cardiologist	11	1	1.60

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).

3

Defining Cube-Cube Relationships

This page describes how to define relationships between cubes for use in Business Intelligence.

For background information, see Summary of Model Options.

For information on how cube versions interact with relationships, consult Cube Versions and Relationships.

Also see Accessing the BI Samples.

3.1 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 indexes.

The Patients sample provides five related cubes, in a folder named `RelatedCubes`. The following figure summarizes how these cubes are related:
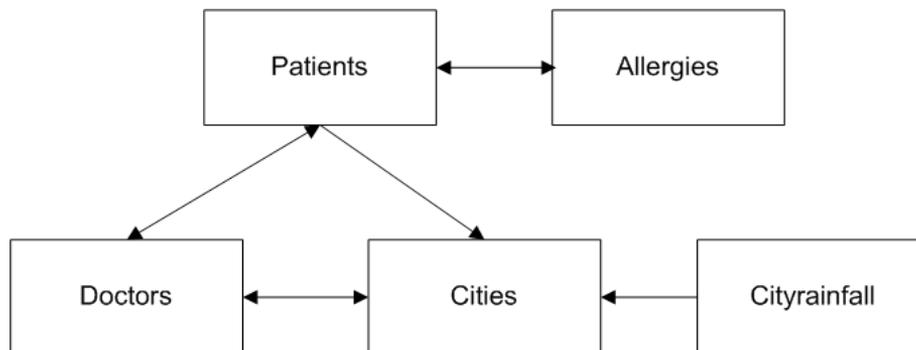
The one-way arrows represent one-way relationships; the cube that has the arrow pointing *away* can see the levels of the other cube. The two-way arrows represent two-way relationships.

3.1.1 A Look at a One-Way Relationship

In this section, we examine a working one-way relationship.

The `RelatedCubes/Patients` cube has a one-way relationship to the `RCities` cube. If you open the `RelatedCubes/Patients` cube in the Analyzer, you see the following cube contents:
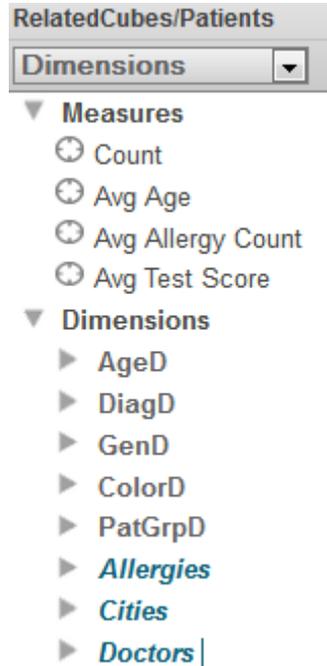
The `Cities` folder is a relationship to the `RelatedCubes/Cities` cube. If you expand it, you see the levels defined in that cube:
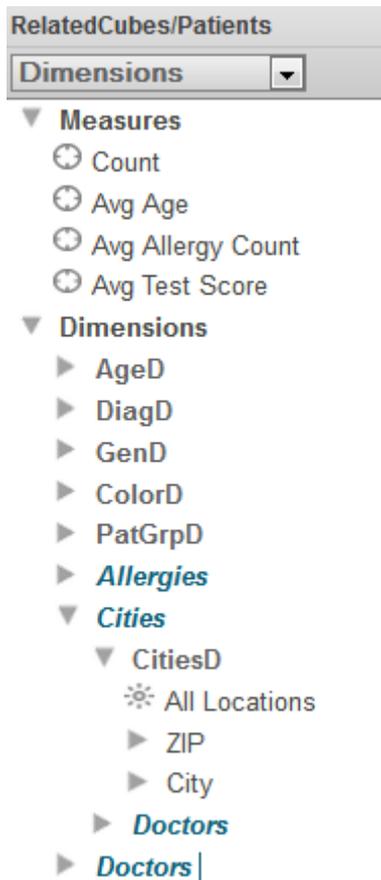
When you work with the `RelatedCubes/Patients` cube, you can use all these levels in the same way that you use the levels that are defined directly in this cube. For example, you could drag and drop `ZIP` to **Rows**.

The `Doctors` folder within `Cities` is a relationship from the `RelatedCubes/Cities` cube to the `RelatedCubes/Doctors` cube; it is not recommended that you use relationships of relationships, because the result quickly becomes confusing.

3.1.2 A Look at a Two-Way Relationship

There is a two-way relationship between the `RelatedCubes/Cities` cube and the `RelatedCubes/Doctors` cube. If you open the `RelatedCubes/Cities` cube, you see the following cube contents:
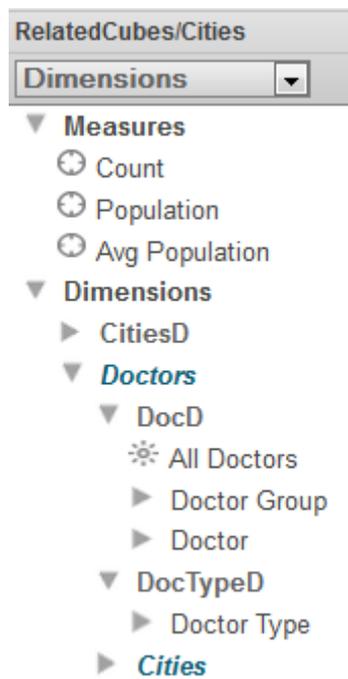
Similarly, if you open the `RelatedCubes/Doctors` cube, you see the following cube contents:
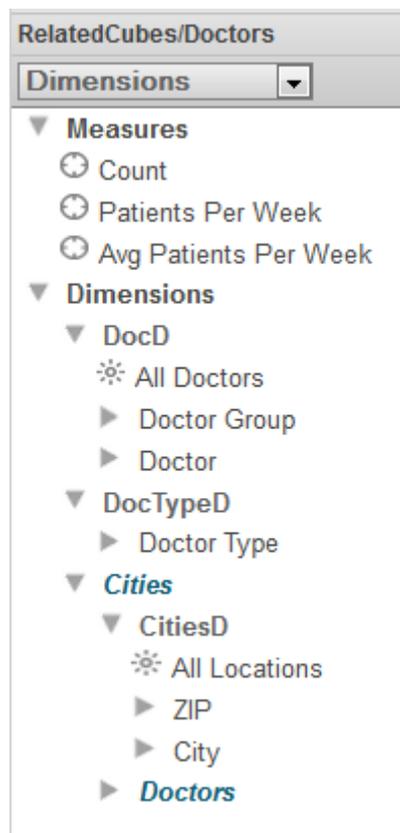
3.2 Considerations for Defining Cube Relationships

Prior to defining any new relationships for cubes, you must take into account whether or not one or both of the affected cubes are already members of a group that is registered in the Cube Manager.

Attempting to add relationships to cubes in registered groups may result in an error and suspended Cube Manager tasks until the error is resolved. In order to prevent this behavior, you must unregister from the Cube Manager any groups which the affected cubes belong to prior to adding the relationships. Once the relationships have been added, reregister the groups and recreate the settings that were previously in place for them.

InterSystems recommends that all such changes to relationships be made on development systems, that the Cube Registry be verified for proper setup, and that the cube definitions and Cube Registry class then be imported onto the production system together.

3.3 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:

1. Select **Add Element**.

    The system displays a dialog box.

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. Select **Relationship**.

4. Select **OK**.

5. In the Details Area on the right, specify the following values:

Attribute	Purpose
Display Name	Display name for this relationship. It is useful for this to be the same as the display name of the other cube.
Property	Specify one of these. The value must be the ID of a record in the base class used by the other cube.
Expression	
Cardinality	`one`
Related cube	Logical name of the other cube.
Null replacement string	(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.

Do not specify **Inverse**.

6. Select the cube definition in the Architect to select it. Then edit the **Depends On** option in the Details Area on the right. This option specifies the class or classes that must be runnable before this class can be compiled.

    By default, the system sets the DependsOn keyword equal to the name of the source class for the cube when a new cube is created.

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 an IDE 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 **Depends on** option as described in Defining Dependencies Between Levels in Different Hierarchies. 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 **Depends on** attribute is completely unrelated to the DependsOn compiler keyword.

3.4 Defining a Two-Way Relationship

To define a two-way relationship between cubes, you define two complementary `<relationship>` elements, one in each cube. One of these cubes is the dependent cube and the other is the independent cube.

Let us consider cube A (which based on the records of source class A) and cube B (which is based on the records of source class B). To define a two-way relationship between these cubes, use the following procedure:

1. Evaluate the relationship between the source classes. Use this to determine which cube is the dependent cube, as follows:

   • Does one record in class A correspond to multiple records in class B?

     If so, cube B is the dependent cube.

   • Does one record in class B correspond to multiple records in class A?

     If so, cube A is the dependent cube.

   • Is there a one-to-one relationship between the classes?

     If so, either cube can be the dependent cube.

2. In the Architect, make the following changes to the dependent cube:

   a. Select **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. Select **Relationship**.

   d. Select **OK**.

   e. In the Details Area on the right, specify the following values:

Attribute	Purpose	Example
Display Name	Display name for this relationship. It is useful for this to be the same as the display name of the other cube.	`Independent Cube Display Name`
Property	Specify one of these. The value must be the ID of a record in the base class used by the other cube.	
Expression		
Cardinality	`one`	
Inverse	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.	`DependentCubeName`
Related cube	Logical name of the other cube.	`IndependentCubeName`
Null replacement string	(Optional) Specifies the string to use as the member name if the source data for a relationship is null. There is no default null replacement for relationships.	`None`

3. Select the cube definition in the Architect; that is, select the top line in the middle area. Then edit the **Depends On** option in the Details Area on the right. This option specifies the class or classes that must be runnable before this class can be compiled.

    By default, the system sets the DependsOn keyword equal to the name of the source class for the cube when a new cube is created.

    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)]
    ```

4. In the Architect, make the following changes to the independent cube::

    a. Select **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. Select **Relationship**.

    d. Select **OK**.

    e. In the Details Area on the right, specify the following values:

Attribute	Purpose	Example
Display Name	Display name for this relationship. It is useful for this to be the same as the display name of the other cube.	`Dependent Cube Display Name`

Attribute	Purpose	Example
Cardinality	`many`	
Inverse	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.	`IndependentCubeName`
Related cube	Logical name of the other cube.	`DependentCubeName`

    f.   Optionally edit the cube class in an IDE 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 **Depends on** option as described in Defining Dimensions, Hierarchies, and Levels. 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 **Depends on** option is completely unrelated to the DependsOn compiler keyword.

3.5 Building Cubes That Have Relationships

When you build cubes that have relationships, 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 cubes in the appropriate order.

For a production system, InterSystems recommends that you use the Cube Manager, which creates automated tasks that build or synchronize cubes according to your specifications. For details, see Using the Cube Manager.

3.5.1 Determining the Build Order for Related Cubes

If you are not using the Cube Manager, it is necessary for you to determine the correct build order. To do so, use the rules described previously or use the **GetCubeGroups()** method of %DeepSee.CubeManager.Utils. The first argument, returned by reference, is an array that indicates the natural grouping of the cubes in the given namespace. For example:

```
SAMPLES>d ##class(%DeepSee.CubeManager.Utils).GetCubeGroups(.groups)

SAMPLES>zw groups
groups=11
groups(1,"CITIES")=1
groups(2,"CITYRAINFALL")=1
groups(3,"COMPOUNDCUBE/CITYRAINFALL")=1
groups(4,"COMPOUNDCUBE/DOCTORS")=1
groups(5,"COMPOUNDCUBE/PATIENTS")=1
groups(6,"CONNECTORCUBE")=1
groups(7,"HOLEFOODS")=1
groups(8,"HOLEFOODSBUDGET")=1
groups(9,"PATIENTS")=1
groups(10,"PATIENTSQUERYCUBE")=1
groups(11,"RELATEDCUBES/ALLERGIES")=5
groups(11,"RELATEDCUBES/CITIES")=1
groups(11,"RELATEDCUBES/CITYRAINFALL")=2
groups(11,"RELATEDCUBES/DOCTORS")=3
groups(11,"RELATEDCUBES/PATIENTS")=4
```

For this array:

- The first subscript identifies a unique group of cubes in this namespace. The first one is numbered 1, the second is numbered 2, and so on. The numbers are arbitrary.

- The second subscript identifies a cube in this namespace. This cube is in the group identified by the first subscript.

- The value of this node is the build order of the given cube, within this group.

In this example, one group consists of the cubes RELATEDCUBES/ALLERGIES, RELATEDCUBES/CITIES, and RELATEDCUBES/CITYTRAINFALL, RELATEDCUBES/DOCTORS, and RELATEDCUBES/PATIENTS. In this group, RELATEDCUBES/CITIES should be built first, then RELATEDCUBES/CITYTRAINFALL, then RELATEDCUBES/DOCTORS, then RELATEDCUBES/PATIENTS, and last RELATEDCUBES/ALLERGIES//.

3.5.2 Errors If You Build Related Cubes in the Wrong Order

If you build related cubes in the wrong error, **%BuildCube()** generates an error like the following:

```
ERROR #5001: Missing relationship reference in RelatedCubes/Patients: source ID 1 missing reference to
 RxHomeCity 4
```

For general information on build errors and where you can see them, see Build Errors.

3.6 Using Model Browser

The system provides a supplemental tool (the Model Browser) that you can use to view cube definitions. The Model Browser is particularly useful for cubes that have relationships because it enables you to view and navigate the cube relationships.

To access this tool, select **Analytics**, then **Tools**, and then select **Model Browser**.

Once you access the Model Browser, use the drop-down list on the right and select the name of a cube. The Model Browser then displays a diagram like the following:
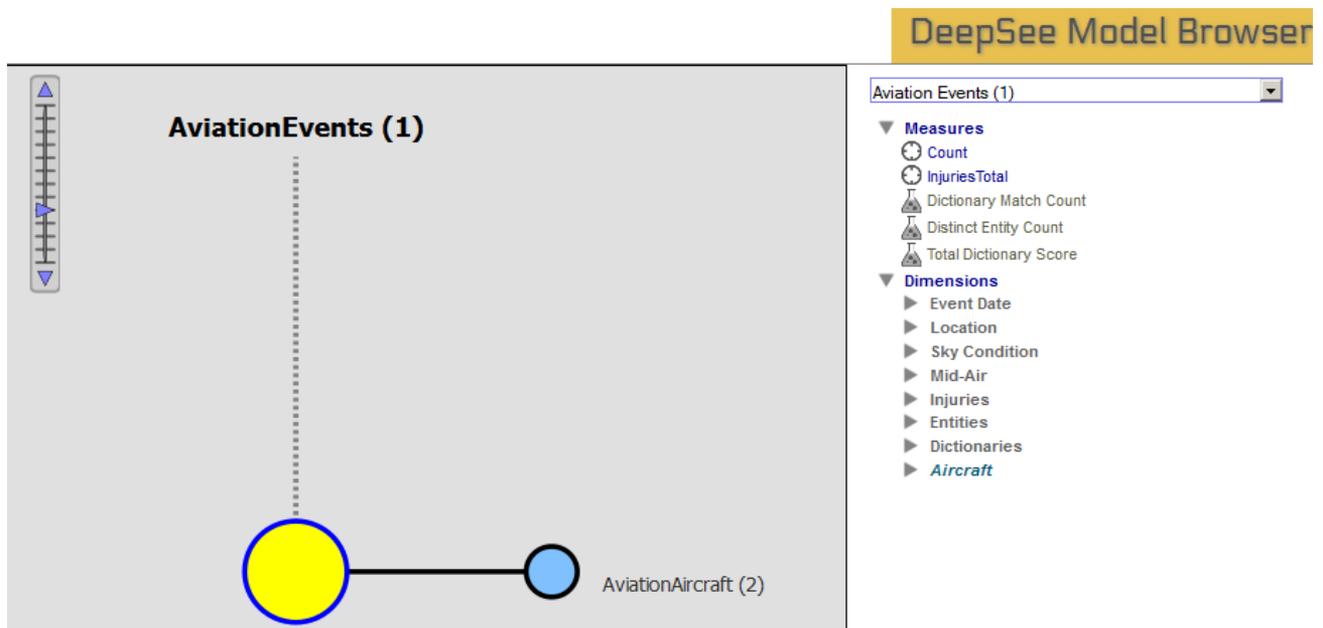
The yellow circle at the center represents the cube that you selected. The label on the top of the diagram gives the name of this cube (in this case, `AviationEvents`), followed by the number of related cubes (1) in parentheses. Each of the circles around the central circle represents one cube that is related to the selected cube. For these circles, the label indicates the cube name and the number of related cubes. For example, the circle labeled `AviationAircraft` represents the Aviation-Aircraft cube. There are two cubes that are related to Referrals.

When you select any circle, the diagram changes so that the center shows the newly selected cube, and the rest of the diagram is refreshed accordingly.

The right side of the page displays details for the selected cube. This area displays the contents of the cube in exactly the same way as the left area of the Analyzer.

3.7 Removing Relationships

If you later decide to remove a relationship, do so as follows:

1. Remove the relationship from both cube definitions.

2. Recompile both cube definitions.

3. Rebuild all modified cubes in the correct order.

4

Using Text Analytics in Cubes

Important: The integration of text analytics within InterSystems IRIS Business Intelligence uses InterSystems IRIS Natural Language Processing (NLP), which InterSystems has deprecated. As a result, this feature may be removed from future versions of InterSystems products. The following documentation is provided as reference for existing users only. Existing users who would like assistance identifying an alternative solution should contact the WRC.

This page describes how to include text analytics within Business Intelligence cubes: analytics that examine text data (called unstructured data here).

Also see Generating Secondary Cubes for Use with Text Analytics.

Also see Accessing the BI Samples.

4.1 Overview of Integration with Text Analytics

The Analytics engine includes the ability to analyze text data (*unstructured data*), which is written as text in a human language such as English or French. For a general introduction of this ability, see Conceptual Overview (NLP).

You can use unstructured data within 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 NLP (natural language processing) 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 page, 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 KPIs that expose text analytics queries via the InterSystems IRIS Business Intelligence KPI mechanism. See KPIs and Dashboards (NLP).)

4.1.1 Terminology

When analyzing unstructured data, the Analytics engine identifies the *entities* in it, identifying the words that belong together and their roles in the sentence. An entity is a minimal logical unit of text — a word or a group of words. Example entities are `clear skies` and `clear sky` (note that these are distinct entities, because the engine does not perform stemming). The 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.

The engine transforms each sentence into a logical sequence of concepts and relationships, thereby identifying the words that belong together (as entities) and their roles within the sentence. This is a much more structured form of data and serves the basis for analysis, including relevance metrics, automated summarizing and categorization, and matching against existing taxonomies or ontologies.

For its core activities of identifying concepts and relationships (as well as measuring the relevance of these entities), the engine does not require information about the topics discussed in the text. Therefore, the engine is a true bottom-up technology that works well with any domain or topic, because it is based on an understanding of the language rather than of the topic.

However, if you have some knowledge about the topics discussed in the text, you can let the engine find any *matches* for these known terms, called *dictionary terms*. Because the engine understands the role and extent of entities, it can also judge whether an entity (or sequence of entities) encountered in the text is a good or a poor match with a known term and come up with a match score. For example, for the dictionary term `runway`, the entity `runway` (occurring in the text) is a full match, but `runway asphalt` is only a partial match and therefore gets a lower score.

Dictionary terms are grouped in *dictionary items*, which represent the unique things you wish to identify. For example `runway` and `landing strip` might be two terms for a generic dictionary item that covers any mentions of runways.

For a broader discussion of dictionaries, see Smart Matching: Creating a Dictionary and Smart Matching: Using a Dictionary.

4.1.2 About NLP (Natural Language Processing) Measures and Dimensions

Unlike other kinds of measures, an NLP measure is not shown in the Analyzer, and you do not directly use it in pivot tables. You define an NLP measure for each property that you want the engine to process as text data. Then you can use the measure as the basis of an NLP dimension.

An NLP dimension *is* like other 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.

There are two kinds of NLP dimensions:

- *Entity dimensions*. An entity dimension contains a single level. Each member of that level corresponds to an entity that the engine found in the unstructured data.

  The members of this level are sorted in decreasing order by *spread* (number of records 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. That member consists of all records that match any item of the given dictionary.

    For example, the `weather conditions` member consists of all records that match any item in the dictionary `weather conditions`. This includes items such as `winter`, `rain`, `clouds sky`, and so on.

  – The optional lower level, the *item level*, contains one member for each dictionary item. That member consists of all records that match any term of the given dictionary item.

    For example, the `winter` member consists of each record that matches any of its terms, including `snow`, `icy`, and `ice-covered`.

Note that because the source text typically includes multiple entities, any given source record is likely to belong to multiple members of a level.

4.1.3 Generated Text Analytics Domains

When you use the features described in this page, the system creates one or more *text analytics domains*. Each cube level and each measure is available as a pseudo metadata field, which you can use in text analytics queries. For a level, the equal and not equal operator are supported. For a measure, all operators are supported. For information on text analytics queries, see *Using InterSystems IRIS Natural Language Processing (NLP)*.

Also see Text Analytics Domain Management, later in this page.

4.2 Setting Up the Aviation Events Demo

This page uses the Samples-Aviation sample (https://github.com/intersystems/Samples-Aviation). InterSystems recommends that you create a dedicated namespace named SAMPLES and then load samples into that namespace. For the general process, see Downloading Samples for Use with InterSystems IRIS® data platform.

The Aviation demo, which includes several cube definitions, example term lists, and a dashboard.

For this demo, the primary cube (Aviation Events) is based on the table Aviation.Event, which 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.

4.2.1 Sample Dashboard

To see the sample dashboard:

1. In the Management Portal, access the namespace into which you loaded this sample.

2. Select **Analytics** > **User Portal**.

    The system then displays the User Portal, which lists any existing public dashboards and pivot tables in this namespace. At the top of the User Portal you can select a **View:** display option.

3. Select the **Aviation event reports** dashboard.

The **Aviation event reports** dashboard includes this pivot table:

This pivot table is defined as follows:

*   The measure is Count (count of events).

- Except for the last row, each row represents an entity that was found by the Analytics engine and matched to an item in a Text Analytics dictionary.

- The last row represents all records matched to items in dictionaries.

- 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.

This means that the rows display groupings as determined by the *unstructured* data (found by the Analytics engine in the textual report), and the columns display groupings as determined by the *structured* data (the direct classifications). You can use a pivot table like this to find any discrepancies between the unstructured data and structured data.

Consider, for example, the `None` column, which provides information on reports that are officially classified with `None` as the highest injury level. Of this set, the cells in this column provide the following information:

- The `iKnow -> none` cell indicates that in 377 reports, the Analytics engine found an entity that matches the `none` item of the Injuries dictionary. This is reasonable.

- The `iKnow -> minor` cell indicates that in four reports, the Analytics engine found an entity that matches the `minor` item of the Injuries dictionary. That is, in four reports, the unstructured data suggests that there were minor injuries (despite the fact that these reports are classified with `None` as the highest injury level).

  The value in this cell represents a discrepancy between the unstructured data and structured data. For this cell, it would be useful to investigate further and read the complete reports.

- The `iKnow -> serious` and `iKnow -> fatal` cells are empty. These cells indicate that there are no records where the Analytics engine found injury entities named `serious` or `fatal`. This is reasonable.

If we display a detail listing for the cell `iKnow -> minor` in the column `None` and then view the reports (via the 🔍 icon) for these incidents, we find that these reports were misclassified and there were minor injuries in all of them. For example, the first report includes the sentence The private pilot reported minor injuries.

Similarly, in the `Minor` column, the cells `iKnow -> serious` and `iKnow -> fatal` indicate other discrepancies. For the `Serious` column, the cell `iKnow -> fatal` indicates other discrepancies.

4.2.2 A Closer Look at the Aviation Cubes

For a closer look at these demo cubes, use the Architect and the Analyzer. The Aviation Events cube contains the following elements:

- The `Count` measure, which is a count of event reports.

- The `InjuriesTotal` measure, which is a sum of injuries.

- The `Report` measure, which is an NLP measure that uses the unstructured data. This measure is not listed in the Analyzer, because it is meant for use only by NLP dimensions.

- The `Event Date`, `Location`, `Sky Condition`, `Mid-Air`, and `Injuries` dimensions, which are standard dimensions that use the structured data.

- The `Entities` dimension, which is an entity dimension.

- The `Dictionaries` dimension, which is an dictionary dimension.

- The `Aircraft` dimension, which is a relationship to the Aircraft cube.

  The Aircraft cube provides dimensions that you can use to group records by attributes of the aircraft, such as its type and model. The Aircraft cube also includes a relationship to the Crew cube, which provides levels associated with the personnel on the aircraft.

4.3 Defining an NLP measure

To add an NLP measure:

1. Select **Add Element**.

   The system displays a dialog box.

2. For **Enter New Item Name**, type a measure name.

3. Select **Measure**.

4. Select **OK**.

5. Select the measure in the middle area of the Architect.

6. Specify the following options:

   - **Property** or **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.

   - **Type** — Select **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 Analytics engine, how to process the values specified in **Property** or **Expression**.

     The value `domain` is for advanced use; see Alternative Technique: Using an Existing Text Analytics Domain.

   As an example, the Aviation cube is based on the `Aviation.Event` class. The NLP measure `Report` is based on the `NarrativeFull` property of that class. For this measure, **iKnow Source** is `string`.

   Note that the **Aggregate** option has no effect on NLP measures.

7. Save the cube definition in the Architect.

8. If you plan to define one or more dictionary levels that use this measure, also specify the **Dictionaries** option as follows:

   a. Select the button below **Dictionaries**.

      The system displays a dialog box.

   b. Select the appropriate dictionary in the **Available Dictionaries** list and then select > to move that dictionary to the **Selected Dictionaries** list.

      If **Available Dictionaries** does not list the dictionaries that you need, see Loading and Updating a Dictionary, later in this page.

   c. Repeat as needed.

   d. Select **OK**.

   Each dictionary is actually a term list. If you follow the steps described here, the Analytics Engine automatically finds the given term lists, loads them as dictionaries, and performs matching. (If you do not add this attribute, you can instead invoke a method to perform these tasks.)

Note that NLP measures are not stored in the fact table for the cube and are not displayed in the Analyzer. The primary purpose of an NLP measure is to define an Text Analytics domain and to serve as the basis of an NLP dimension. See the next sections.

Also note that you can override the default parameters for the measure. See Specifying Text Analytics Domain Parameters for a Measure, later in this page; this option is for advanced use.

4.3.1 Alternative Technique: Using an Existing Text Analytics Domain

If you have an existing Text Analytics 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 source that corresponds to that record in the fact table.

- In an IDE, add the `iKnowDomain` attribute to the measure definition. Its value should be the name of an existing Text Analytics domain.

- Skip step 8. That is, do not specify the **Dictionaries** option.

Important: In this case, the Text Analytics domain is managed separately from the process of compiling, building, and synchronizing cubes. At build time, the Analytics Engine does not drop or load any text analytics 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), the Analytics Engine uses the Text Analytics logic. Also note that the Analytics Engine does not load automatically data during the process of compiling, building, and synchronizing cubes. To load data, specify parameters, or otherwise manage this domain, use the domain APIs directly as described in *Using InterSystems IRIS Natural Language Processing (NLP)*.

4.3.2 Alternative Technique: Retrieving Unstructured Text from Elsewhere

In some scenarios, you may need to retrieve the unstructured text from a web page. For example, you might have a table of structured information, with a field that contains the URL where additional information (such as a news article) can be found. In such a case, the easiest way to use that text as an NLP measure is as follows:

- Write a utility method to retrieve the text from the URL.

- Refer to that utility method in a source expression for the NLP measure.

For an example, suppose that we are basing a cube on a class that has summary information about news articles. Each record in the class contains the name of the news agency, the date, the headline, and a property named `Link`, which contains the URL of the full news story. We want to create an NLP measure that uses the news stories at those URLs.

To do this, we could define a method, `GetArticleText()`, in the cube class as follows:

Class Member

```
ClassMethod GetArticleText(pLink As %String) As %String
{
    set tSC = $$$OK, tStringValue = ""
    try {

        set tRawText = ..GetRawTextFromLink(pLink, .tSC)
        quit:$$$ISERR(tSC)

        set tStringValue = ..StripHTML(tRawText, .tSC)
        quit:$$$ISERR(tSC)

    } catch (ex) {
        set tSC = ex.AsStatus()
    }
    if $$$ISERR(tSC) {
        set tLogFile = "UpdateNEWSARCHIVE"
        set tMsg = $system.Status.GetOneErrorText(tSC)
```

```
            do ##class(%DeepSee.Utils).%WriteToLog("UPDATE", tMsg, tLogFile)
        }
    quit tStringValue
}
```

The `GetRawTextFromLink()` method would retrieve the raw text, as follows:

Class Member

```
ClassMethod GetRawTextFromLink(pLink As %String, Output pSC As %Status) As %String
{
    set pSC = $$$OK, tRawText = ""
    try {
        // derive server and path from pLink
        set pLink = $zstrip(pLink,"<>W")
        set pLink = $e(pLink,$find(pLink,"://"),*)
        set tFirstSlash = $find(pLink,"/")
        set tServer = $e(pLink,1,tFirstSlash-2)
        set tPath = $e(pLink,tFirstSlash-1,*)

        // send the HTTP request for the article
        set tRequest = ##class(%Net.HttpRequest).%New()
        set tRequest.Server = tServer
        set tSC = tRequest.Get(tPath)
        quit:$$$ISERR(tSC)

        set len = 32000
        while len>0 {
            set tString = tRequest.HttpResponse.Data.Read(.len, .pSC)
            quit:$$$ISERR(pSC)
            set tRawText = tRawText _ tString
        }
    } catch (ex) {
        set pSC = ex.AsStatus()
    }
    quit tRawText
}
```

The `StripHTML()` method would remove the HTML formatting, as follows:

Class Member

```
ClassMethod StripHTML(pRawText As %String, Output pSC As %Status) As %String
{
    set pSC = $$$OK, tCleanText = ""
    try {
        for tTag = "b","i","span","u","a","font","em","strong","img","label","small","sup","sub" {
            set tReplaceTag(tTag) = " "
        }

        set tLowerText = $$$LOWER(pRawText)
        set tStartPos = $find(tLowerText,"<body")-5, tEndTag = ""
        set pRawText = $e(pRawText,tStartPos,*), tLowerText = $e(tLowerText,tStartPos,*)
        for {
            set tPos = $find(tLowerText,"<")
            quit:'tPos // no tag start found

            set tNextSpace = $f(tLowerText," ",tPos), tNextEnd = $f(tLowerText,">",tPos)
          set tTag = $e(tLowerText,tPos,$s(tNextSpace&&(tNextSpace<tNextEnd):tNextSpace, 1:tNextEnd)-2)

            if (tTag="script") || (tTag="style") {
                set tPosEnd = $find(tLowerText,">",$find(tLowerText,"</"_tTag,tPos))
            } else {
                set tPosEnd = tNextEnd
            }
            if 'tPosEnd { //
                set tEndTag = $e(pRawText,tPos-1,*)
                set pRawText = $e(pRawText,1,tPos-2)
                quit
            }

            set tReplace = $s(tTag="":"", 1:$g(tReplaceTag(tTag),$c(13,10,13,10)))
            set pRawText = $e(pRawText,1,tPos-2) _ tReplace _ $e(pRawText,tPosEnd,*)
            set tLowerText = $e(tLowerText,1,tPos-2) _ tReplace _ $e(tLowerText,tPosEnd,*)
        }
        set tCleanText = $zstrip($zconvert(pRawText, "I", "HTML"),"<>=W")

    } catch (ex) {
        set pSC = ex.AsStatus()
```

```
    }
    quit tCleanText
}
```

Finally, we would create an NLP measure and base it on the following source expression:
`%cube.GetArticleText(%source.Link).`

4.4 Loading and Updating Dictionaries

This section describes how to load and update dictionaries for use with NLP measures and dimensions.

4.4.1 Loading Dictionaries

To load an dictionary into InterSystems IRIS:

1. Access the Term List Manager, as described in the Accessing the Term List Manager.

2. Define a new term list to contain the dictionary items and terms. For this term list:

   • Use a convenient name for the term list. The dictionary name is based on the term list name, with an added prefix.

   • Optionally add the *custom fields* **URI** and **language**; see the following step for details on how these fields would be used.

   Every term list has the fields **key** and **value**, so your term list will have these fields as well.

   For general information on creating term lists, see Defining Term Lists.

3. Add terms to the term list. For each term list item, specify values as follows:

   • **key** (required) is a unique term that could be found in the text.

   • **value** (required) is the corresponding dictionary item.

   • **URI** (optional) is a unique identifier for the dictionary item (the **value** column of the term list), which you can then use as a member key in MDX queries, if you need to refer to a specific dictionary item. This identifier must be unique for each combination of dictionary name and dictionary item.

   If you omit this field, the system generates a URI of the following form:

   `:dictionary_name:dictionary_item`

   Where *dictionary_name* is the name of the dictionary to define or update, and *dictionary_item* is the value in the **value** field.

   • **language** (optional) is a all-lowercase language tag (such as `en` or `es`).

The following shows an example (omitting the **language** field):

key	value	URI
broken clouds	clouds	:weather:clouds
calm winds	mild wind	:weather:wind
clear of clouds	clear	:weather:visibility
clear skies	clear	:weather:visibility
clear sky	clear	:weather:visibility
cumuliform clouds	clouds	:weather:clouds
drizzle	rain	:weather:rain
extreme turbulence	heavy wind	:weather:heavy wind

Terms

4. Save the term list.

5. Specify the **Dictionaries** option for each NLP measure that should use this term list as a dictionary. See Defining an NLP measure.

The **Dictionaries** option specifies the term lists to load as dictionaries for this NLP measure. The system automatically loads these term lists at cube build time.

4.4.2 Updating Dictionaries

If you create or change a term list that is used as a dictionary, you must update the dictionary. To do so, use the **UpdateDictionary()** method of %iKnow.DeepSee.CubeUtils:

```
classmethod UpdateDictionary(pTermList As %String,
                             pCube As %String = "",
                             pMeasure As %String = "",
                             pClearFirst As %Boolean = 0) as %Status
```

Where:

- *pTermList* is the name of term list.

- *pCube* is the name of the cube. If you omit this argument, this method is invoked for all cubes in this namespace.

- *pMeasure* is the name of NLP measure. If you omit this argument, this method is invoked for all NLP measures in the given cube (or all cubes, depending on *pCube*).

- *pClearFirst* controls whether to drop the existing dictionary before reloading it from the term list. Leave *pClearFirst* as 0 if you only appended to the term list, or use 1 if you changed or removed any existing terms.

  If *pClearFirst* is 0, this method can run significantly faster.

Note that when you build a cube, the system refreshes all dictionaries used by this cube by appending any new term lists. Deleted and renamed items are not affected. See When Text Analytics Updates Occur, later in this page.

4.5 Defining an Entity Dimension

To add a entity dimension:

1. Create an NLP measure for this dimension to use, as described earlier in this page.

You can also do this *after* defining the dimension; if so, edit the dimension later so that it refers to this measure.

2. Select **Add Element**.

   The system displays a dialog box.

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

4. Select **iKnow Dimension** and select **OK**.

5. Select the dimension in the middle area of the Architect.

6. Make the following changes to the dimension, if needed:

   - **iKnow type** — Select **entity**.

   - **NLP measure** — Select the NLP measure for this dimension to use.

7. Select the level in the middle area of the Architect and optionally modify **Name** and **Display Name**.

8. Optionally, to specify the members of this level manually, use an IDE 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. Note that for an entity dimension, the number of members displayed in the Analyzer is fixed at 100.

   See Manually Specifying the Members for a Level, in 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 NLP dimension, the associated NLP measure specifies the source values.

4.6 Defining a Dictionary Dimension

To add a dictionary dimension:

1. Load a dictionary into InterSystems IRIS. See Loading Dictionaries, earlier in this page.

   You can also do this *after* defining the dimension.

2. Create an NLP 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. Select **Add Element**.

   The system displays a dialog box.

4. For **Enter New Item Name**, type a dimension name.

5. Select **iKnow Dimension** and select **OK**.

6. Select the dimension in the middle area of the Architect.

7. Make the following changes if needed:

   - **iKnow type** — Select **dictionary**.

   - **NLP measure** — Select the NLP 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. Select each level in the middle area of the Architect and optionally modify **Name** and **Display Name**.

10. Save the cube definition in the Architect.

11. Open the cube class in an IDE 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):

```
<dimension name="MyDictionaryDimension" disabled="false"
` hasAll="false" allCaption="MyDictionaryDimension" allDisplayName="MyDict"
  type="iKnow" iKnowType="dictionary"  nlpMeasure="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:

```
<dimension name="MyDictionaryDimension" disabled="false"
  hasAll="false" allCaption="MyDictionaryDimension" allDisplayName="MyDict"
  type="iKnow" iKnowType="dictionary"  nlpMeasure="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 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 dictionaries, all dictionaries are used.

   For each dictionary to use, add the following between the `<level>` element and the `</level>`:

```
<member name="dictionary name" />
```

   Where *dictionary name* is the name of a dictionary.

   For example, with a single dictionary:

```
<dimension name="MyDictionaryDimension" disabled="false"
  hasAll="false" allCaption="MyDictionaryDimension" allDisplayName="MyDict"
  type="iKnow" iKnowType="dictionary"  nlpMeasure="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.

Note that it is not necessary to specify anything for **Source Values**, either for the dimension or for the level. For an NLP dimension, the associated NLP measure specifies the source values.

4.7 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:

```
<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 Dictionaries, earlier in this page.

List the `<member>` elements in the desired sort order. For example:

```
<level name="ReportDictInjuriesDimItem" displayName="Injuries" >
   <member name=":injuries:none" displayName="not injured" />
   <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.

4.8 Adding Measures That Use Plug-ins

A plug-in is essentially a query, and the system provides plug-ins that perform specialized text analytics queries. You can use these plug-ins to add calculated measures that provide information on entity occurrences and on matching results. The following sections give the details.

4.8.1 Adding Measures to Quantify Entity Occurrences

You can easily add measures that provide information on entity occurrences (indicate the total number, average number per record, and so on). For an example, see the calculated measure Distinct Entity Count in the Aviation Events cube.

To add your own measures, follow the steps in Defining a Calculated Measure. For **Expression**, use the following expression:

```
%KPI("%DeepSee.iKnow","Result",1,"aggregate","total","%CONTEXT")
```

This expression returns the total number of *distinct* entities, in any given context.

Instead of `"total"`, you can use any of the following:

- `"sum"` — In this case, the expression returns the total number of entities (as opposed to distinct entities), in the given context. That is, entities might be counted more than once.

- `"average"` — In this case, the expression returns the average number of entities per record, in the given context.

- `"max"` — In this case, the expression returns the maximum number of entities in any given record, in the given context.

- `"min"` — In this case, the expression returns the minimum number of entities in any given record, in the given context.

This expression uses the %KPI MDX function and the plug-in class %DeepSee.PlugIn.iKnow. For details on the function, see *InterSystems MDX Reference*. For details on the class, see the class reference.

Important: If you omit the `"%CONTEXT"`, then in all cases, your calculated measures ignore any context and return results for your entire data set.

4.8.2 Adding Measures to Quantify Matching Results

You can easily add measures that provide information on dictionary matching results (indicate the total number, average matching score per record, and so on). For examples, see the calculated measures Dictionary Match Count and Total Dictionary Score in the Aviation Events cube.

To add your own measures, follow the steps in Defining a Calculated Measure. For **Expression**, use one of the following expressions:

- To get counts of matching results (results that match dictionary items):

    ```
    %KPI("%DeepSee.iKnowDictionary","MatchCount",1,"aggregate","sum","%CONTEXT")
    ```

    This expression returns the total number of matching results, in any given context.

    Instead of `"sum"`, you can use the alternative aggregation types listed in the previous section.

- To get scores for matching results:

    ```
    %KPI("%DeepSee.iKnowDictionary","MatchScore",1,"aggregate","sum","%CONTEXT")
    ```

    This expression returns the total score for matching results, in any given context.

    Instead of `"sum"`, you can use the alternative aggregation types listed in the previous section.

These expressions use the %KPI MDX function and the plug-in class %DeepSee.PlugIn.iKnowDictionary. For details on the function, see *InterSystems MDX Reference*. For details on the class, see the class reference.

Important: If you omit the `"%CONTEXT"`, then in all cases, your calculated measures ignore any context and return results for your entire data set.

4.9 Including Text Analytics Results in Listings

You can include Text Analytics results in listings as follows:

- You can include a summary field in a listing

- You can add a link from the listing to the complete unstructured text

- You can define a specialized listing for use in the Content Analysis plug-in

4.9.1 Including a Text Analytics 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[nlpMeasure,summarylength] As Report
```

Where *nlpMeasure* is the name of the NLP measure to summarize and *summary_length* is the number of sentences to include in the summary (the default is five). You can omit *nlpMeasure* if there is only one NLP measure in the cube.

The `As` clause specifies the title of the column; in this case, the title is `Report`.

The `$$$IKSUMMARY` token 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>
```

Internally `$$$IKSUMMARY` uses the **GetSummary()** method of %iKnow.Queries.SourceAPI.

You can also use the `$$$IKSUMMARY` token to refer to an NLP measure in a related cube, if there is a many-to-one relationship between the listing cube and the related cube. In this case, use *relationshipname.nlpMeasure* instead of *nlpMeasure* as the first argument to `$$$IKSUMMARY`. For example, suppose that the `Observations` cube has a relationship called `Patient`, which points to the `Patients` cube. Also suppose that the `Patients` cube has an NLP measure named `History`. Within the `Observations` cube, you can define a listing that includes `$$$IKSUMMARY[Patient.History]`.

You can refer to a relationship of a relationship in the same way. For example:
`$$$IKSUMMARY[Relationship.Relationship.Measure]`.

4.9.2 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[nlpMeasure]
```

Where *nlpMeasure* is the name of the NLP measure to display. You can omit *nlpMeasure* if there is only one NLP measure in the cube.

You can also use the `$$$IKLINK` token to refer to an NLP measure in a related cube, if there is a many-to-one relationship between the listing cube and the related cube. In this case, use *relationshipname.nlpMeasure* instead of *nlpMeasure* as the first argument to `$$$IKLINK`. You can refer to a relationship of a relationship in the same way. For example:
`$$$IKLINK[Relationship.Relationship.Measure]`.

See the examples in the previous subsection.

4.9.3 Creating a Specialized Listing for Use in Content Analysis Plug-in

The Analyzer provides advanced analysis options, which include the Content Analysis plug-in. This option uses a detail listing to display the five most typical and five least typical records. By default, the plug-in 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 plug-in uses that listing instead.

In either case, the plug-in adds a **Score** column to the right of the columns defined in the listing.

For details on this analysis option, see Text Analytics Content Analysis.

4.10 Text Analytics Domain Management

When you use the features described in this page, the system creates one or more *Text Analytics domains*. These Text Analytics domains are managed by Business Intelligence (unlike Text Analytics domains that you create directly as described in *Using InterSystems IRIS Natural Language Processing (NLP)*). To modify them, you should use only the APIs described in this page.

InterSystems IRIS manages these domains in a way that requires little or no intervention. If you are familiar with Text Analytics domains, however, you might be interested in the details.

The system creates one Text Analytics domain for each NLP 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 NLP measure.

InterSystems IRIS 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, the Analytics engine processes the text in the NLP 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 NLP measure (rather than considering the logical name of the NLP measure). Therefore, when you rename an NLP measure, the system reuses the existing Text Analytics domain.

- When you remove an NLP measure and recompile the cube, the system deletes the corresponding Text Analytics domain and all associated engine results.

- When you delete the cube, the system deletes the Text Analytics domains and removes all associated engine results.

4.11 Advanced Topics

This section discusses the following advanced topics:

- How to override the default Text Analytics domain parameters
- How to load skiplists for use with the cube
- How to update skiplists

4.11.1 Specifying Text Analytics Domain Parameters for a Measure

In rare cases, you might want to override the default Text Analytics domain parameters used for a given NLP measure. To do so, edit the cube class in an IDE and add the following to the definition of the applicable NLP measure:

```
iKnowParameters="parameter::value;parameter::value"
```

For *parameter*, use the parameter name or the macro that represents the parameter. Use two colons between a parameter and its value. Use a semicolon to separate each name/value pair from the next pair in the list.

The following example overrides the default values for the `DefaultConfig` and `MAT:SkipRelations` parameters:

```
iKnowParameters="DefaultConfig::Spanish;MAT:SkipRelations::0"
```

For details on Text Analytics domain parameters, see Alternatives for Creating an NLP Environment.

4.11.2 Loading Skiplists

A skiplist is a list of entities that you do not want a query to return. To load skiplists for use with Business Intelligence:

1. Create a term list that consists of the skiplist items. For information on creating term lists, see Defining Term Lists.

2. Edit an NLP measure to use this skiplist. To do this, edit the measure in an IDE and specify the `iKnowParameters` attribute. This attribute contains one or more name/value pairs, where the names are Text Analytics domain parameters and the values are the corresponding values. For general information on specifying `iKnowParameters`, see Specifying Text Analytics Domain Parameters for a Measure, earlier in this page.

    In this case, the Text Analytics domain parameter is `$$$IKPDSSKIPLIST` and its value is the term list name.

3. Either rebuild the cube or manually load the term list as a skiplist.

    To manually load the term list as a skiplist, use the following class method of %iKnow.DeepSee.CubeUtils:

    ```
    classmethod LoadTermListAsSkipList(pCube As %String,
                                       pMeasure As %String,
                                       pTermList As %String) as %Status
    ```

    Where:

    - *pCube* is the name of the cube that uses this NLP measure.

    - *pMeasure* is the name of NLP measure that uses this skiplist.

    - *pTermList* is the name of term list.

The system uses this skiplist for the following purposes:

- To filter entities returned by the entity dimension, in the Analyzer or directly through MDX.

- To exclude entities from the derivation of top groups as shown in the Entity Analysis screen (described in Using the Pivot Analysis Window). Entries on skiplists (or their standardized form) will not be a group by themselves, but still contribute to the scores of other groups. For example, if `pilot` is on a skiplist, `helicopter pilot` still contributes to the group `helicopter`.

- To filter entities shown in the Entities detail tab in the Entity Analysis screen.

The skiplist does not affect the assisted text entry for the **Analyze String** option in the Entity Analysis screen.

4.11.3 Updating Skiplists

To update a skiplist, edit the corresponding term list and then either rebuild the cube or manually load the term list as a skiplist as described in the previous subsection.

4.12 When the NLP Updates Occur

The following table summarizes when the Analytics engine updates dictionaries, skiplists, and matching results:

Action	Automatic NLP Updates by the Analytics Engine
Compiling a cube	None
Building a cube	• Refreshes all dictionaries used by this cube by appending any new term lists (deleted and renamed items are not affected) • Completely refreshes all skiplists for term lists used by the cube • Fully updates the matching results
Synchronizing a cube	• No refresh of dictionaries or skiplists • Creates results for any *new* records in the fact table
Updating a term list via the API or the Management Portal	None
Updating a term list via the **UpdateDictionary()** method of %iKnow.DeepSee.CubeUtils	• Completely refreshes the dictionary represented by the specified term list • Fully updates the matching results with respect to the specified term list

5

Defining Term Lists

Term lists provide a way to customize an InterSystems IRIS® data platform Business Intelligence model without programming. This page describes how to define them.

Also see Accessing the BI Samples.

5.1 Overview of Term Lists

Term lists provide a way to customize a Business Intelligence model without programming. A term list is a simple (but extendable) list of **key** and **value** pairs. The key values must be unique within the list. You can extend the term list by adding custom fields. You can use term lists in the following ways:

*   You can use a term list to define additional data to use in the cube. In this case, the term list is used when the cube is built. For example, you can use a term list to define a set of values to use as properties of a level. Or you could use a term list to provide an alternative set of names for the level members.

*   You can use a term list at run time to access additional values. (In this case, you use the LOOKUP and %LOOKUP functions.)

*   You can use a term list at run time to build a set of members, typically for use in a filter. In this case, you must specify a term list pattern. You use the term list with the %TERMLIST function.

Tip: You can use term lists to define matching dictionaries for use with unstructured data. See Using Unstructured Data in Cubes.

5.2 Accessing the Term List Manager

To define and modify term lists, you use the Term List Manager. To access this tool:

1.   Select the InterSystems Launcher and then select **Management Portal**.

    Depending on your security, you may be prompted to log in with an InterSystems IRIS username and password.

2.   Switch to the appropriate namespace as follows:

    a.   Select the name of the current namespace to open the list of available namespaces.

    b.   From the list, select the appropriate namespace.

3. Select **Analytics** > **Tools** > **Term List Manager**.

The **Term List Manager** page initially looks like this:
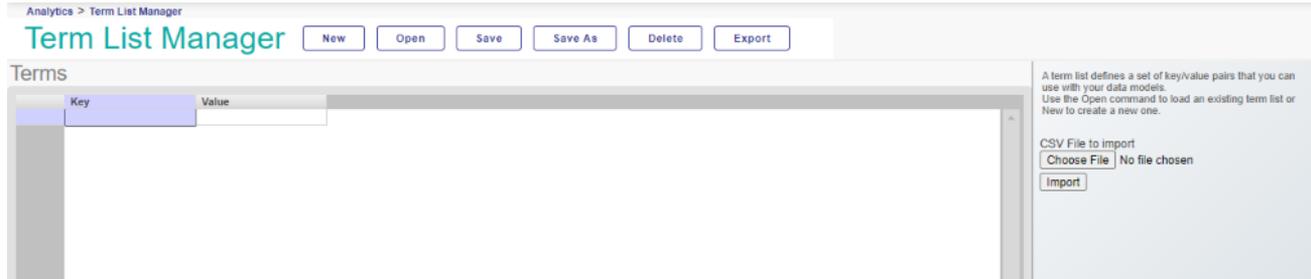
On this page you can do the following:

- Create new term lists. Select **New** and continue as described in Defining a Term List.

- Open and modify existing term lists. Select **Open**, select a term list, and continue as described in Defining a Term List.

- Export and import term lists. See Exporting and Importing Term Lists.

5.3 Defining a Term List

To define a term list, select **New**. The system displays a dialog box that asks for the name of the new term list. Enter a name and select **OK**. Then do the following tasks as needed:

- Enter descriptive information about the term list into the following optional fields on the **Details** tab:

    – **Caption** — Optional caption for the term list

    – **Subject Area** — Select the subject area where this term list will be used

    – **Description** — Longer description of the term list

    For information on **Pattern**, see Specifying the Pattern for a Term List.

- Add terms. To do so, select **Add Term** on the right. Or select the New Term button .

    A new row is added to the bottom of the term list.

- Add custom fields. To do so, enter a value into **Field Name** and then select the **Add Field** button on the right. Or select the New Field button .

    A new column is added on the right side of the term list.

- Enter values into cells. To enter a value, select a cell and type a value. Then move the cursor to another cell, press **Enter**, or select **OK**.

- Remove a term. To do so, select the row for the term and select **Remove Term** on the **Details** tab.

- Remove a custom field. To do so, select the column for the custom field and select the **Remove Field** button on the right.

Then save the term list. To do so, select **Save** or **Save As**. If you select **Save As**, specify a new name and then select **OK**.

5.4 Specifying the Pattern for a Term List

The %TERMLIST function returns a set that, by default, consists of members that are identified by the keys in the term list, in combination with the term list pattern.

If you intend to use a term list with the %TERMLIST function, then define the term list as follows:

- For each term, specify the **value** field as a member name or the numeric key.

- For **Pattern**, specify an expression of one of the following forms:

  - If the **value** field contains member names, use the form:

    ```
    [dimension name].[hierarchy name].[level name].[*]
    ```

  - If the **value** field contains member keys, use the form:

    ```
    [dimension name].[hierarchy name].[level name].&[*]
    ```

In either case, *dimension name*, *hierarchy name*, and *level name* are the names of the dimension, hierarchy, and level to which the member belongs. The asterisk represents the value in the **value** field. For an example, see %TERMLIST.

5.5 Exporting and Importing Term Lists

You can export term lists to comma-separated files and you can import term lists from such files.

5.5.1 Exporting a Term List

To export a term list:

1. Display the term list in the Term List Manager.

2. Select **Export**. The system generates a file.

   Depending on your browser settings, the browser then either saves the file automatically (in the default download directory for this browser) or opens the file in a default application.

5.5.2 Sample Term List File

The following shows the contents of an example term list file:

```
%%NAME,My Term List
%%CAPTION,My Caption
%%DESCRIPTION,My Description
%%SUBJECTAREA,HoleFoods
%%MODDATE,2014-06-06 11:31:10
Atlanta,Braves
Boston,Red Sox
New York,Yankees
```

As you can see, the first lines of the export contain items that start with %%; these lines contain descriptive information about the term list.

5.5.3 Importing a Term List

To import a term list:

1. In the right area, for **CSV File to import**, select **Browse...** and navigate to the file.

2. Select **Import**.

    InterSystems IRIS examines the name of the term list as contained in this file. Then:

    *   If the system already has a saved term list with the same name, the system asks if you want to overwrite that term list. If you *potentially* want to overwrite the term list, select **Yes**. Otherwise, select **Cancel** to cancel the import action.

        If you select **Yes**, the system imports the file and displays the new definition of the term list, but does not save the change.

    *   If the system does not yet contain a saved term list with the same name, the system imports the file and displays the new term list, but does not save the new term list.

3. Optionally save the new or changed term list. To do so, select **Save** or **Save As**. If you select **Save As**, specify a new name and then select **OK**.

5.6 Deleting a Term List

To delete a term list:

1. Display the term list in the Term List Manager.

2. Select **Delete**.

3. Select **OK**.

5.7 Accessing Term Lists Programmatically

You can use the %DeepSee.TermList class to access term lists programmatically. This class provides class methods like the following:

*   **%ExportCSV()**

*   **%GetTermLists()**

*   **%GetValueArray()**

*   **%Lookup()**

*   **%TermListExists()**

*   And others

See the *InterSystems Class Reference*.

6
Defining Worksheets

The Management Portal provides the **Worksheet Builder** option, which is no longer documented. Note that worksheets are now deprecated.

7

Defining Quality Measures

This page describes how to define quality measures for use in Business Intelligence. Quality measures are similar to calculated measures (see Defining Calculated Members).

For a comparison of quality measures with more basic Business Intelligence model elements, see Summary of Model Options.

7.1 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 an InterSystems IRIS® data platform class definition, specifically a subclass of %DeepSee.QualityMeasure.QualityMeasure.

You can define quality measures either in the Quality Measure Manager or in an IDE. This page discusses the Quality Measure Manager.

You can use quality measures as follows:

- Published quality measures 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 quality measures 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*.)

You can also refer to a group (*group name*) in a quality measure as follows:

```
[%QualityMeasure].[catalog/set/qm name/group name]
```

7.2 Introduction to the Quality Measure Manager

The Quality Measure Manager displays all compiled 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, select **Analytics**.

2. Select **Tools** > **Quality Measures**.

Here you can do the following:

*   View summary information for the existing quality measures in this namespace. To do so, select **Browse**.

    The middle area then displays something like the following:

*   View the definition of a quality measure. To do so, select **Open**, expand the folders until you find the quality measure, and then select 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:

Measure

[Numerator]/[Denominator]

Numerator

Numerator

    ⭐ **Numerator Cohort**

    [tests].[h1].[leadscr].[yes]

Denominator

Denominator

    ⭐ **Denominator Cohort**

    ([patgrp].[h1].[medicaid].[yes],[age].[h1].[0 to 2])

You can then edit the measure or remove it.

- Create a quality measure. To do so, select **New** and then continue as described in the next section.

7.3 Creating a Quality Measure

To create a quality measure:

1. Select **New**.

   The system 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. Select **OK**.

   The system closes the dialog box, and the middle area of the page now displays the initial definition of the quality measure, as follows:

Measure

[Numerator]/[Denominator]

Numerator
Numerator

⭐ **Numerator Cohort**
    100

Denominator
Denominator

⭐ **Denominator Cohort**
    100

Every quality measure is expressed as a formula as shown here in the **Measure** field. In this case, the formula is:

```
[Numerator]/[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. Select **Edit**.

   Now you can modify the formula and make other changes. See the following sections.

5. When you are done, select either **Save** to save the definition or select **Save As** to save it with a new name.

   When you save the definition, the system automatically compiles the class and writes the quality measure definition into a system global.

7.4 Specifying the Expression for a Quality Measure

Except for linked quality measures, when you display a quality measure in edit mode, you can edit the overall expression, which looks like this initially:

Here you can do the following:

- Change the overall expression. To do so, select the edit button ⬜ . Then specify a new expression and select **OK**. In your formula:

  – Use square brackets around the name of any group that is used in the formula. For example, use `[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, select 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, select the edit button ⬜ . Then specify new values for **Name** and **Description** as needed, and select **OK**.
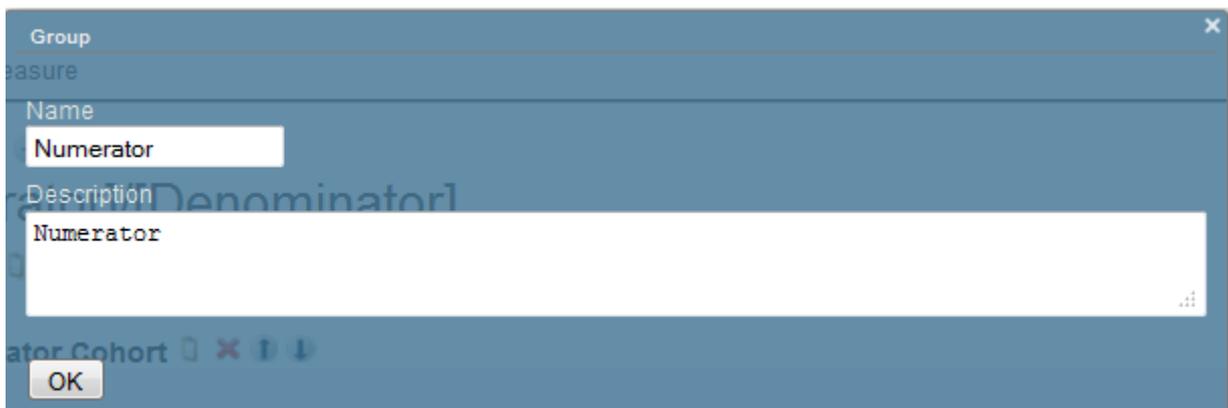
- Add an element to a group. To do so, select the add button ⊕ next to the group name. Then edit the new element.

- Edit a element in a group. To do so, select the edit button ⬜ next to the element name. Then specify new values for **Name** and **MDX Expression** as needed, and select **OK**.
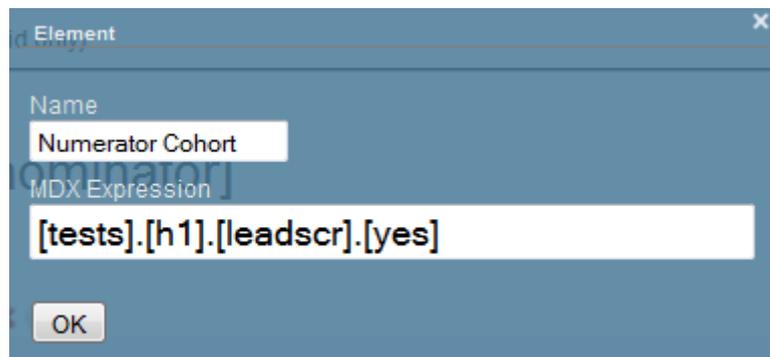
For **MDX Expression**, see Allowed MDX Expressions.

- Change the order of groups within the measure definition or change the order of elements within a group. To do so, select the up and down arrow next to the item that you want to move.

- Delete a group or an element. To do so, select the X button next to the item you want to delete. The system prompts you to confirm this action.

7.4.1 Allowed MDX Expressions

For **MDX Expression**, specify one of the following kinds of expressions:

- A reference to a member, as shown in the example. This is the most common kind of expression used in quality measures. A member is a set of records, and a quality measure usually consists of comparisons between different sets of records.

- Most other kinds of member expressions.

- Numeric literals such as 42 and other kinds of numeric expressions. You can, for example, use functions such as AVG and COUNT.

- String literals such as "my label" and some other kinds of string expressions. The PROPERTIES and IIF functions are not supported here.

- A tuple expression.

MDX Expression cannot be a set expression. You can, however, use %OR to enclose a set and return it as a single member.

Search expressions are not supported within quality measures.

For information on creating MDX expressions, see *Using InterSystems MDX* and the *InterSystems MDX Reference*.

7.4.2 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, Business Intelligence 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.

Note that a quality measure can define groups that are not used by the formula. This can be useful if you are using the special [%QualityMeasure] dimension to refer to groups, as described earlier in this page.

7.5 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, select the edit button next to that item and then specify the details.

- Add an item. To do so, select **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 an IDE.

  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. The system does not use these items.

7.6 Defining a Linked Quality Measure (Quality Measure Alias)

A linked quality measure is an alias for another quality measure (the master measure). The master measure may or may not be published, and can belong to a different subject area.

To define a linked quality measure:

1. Select **New**.

The system 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. Select **OK**.

    The system closes the dialog box, and the middle area of the page now displays the initial definition of the quality measure.

4. Select **Edit**.

5. For **Linked To**, select the parent measure and select **OK**.

    If you specify **Linked To** for an existing quality measure, the editor discards values for any optional properties of the measure.

6. Optionally specify the following options:

    •   **Caption**

    •   **Subject Area**

    •   **Published**

    Other options are inherited from the master measure.

7. When you are done, select either **Save** to save the definition or select **Save As** to save it with a new name.

    When you save the definition, the system automatically compiles the class and writes the quality measure definition into a system global.

7.7 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])

7.8 Deleting a Quality Measure

To delete a quality measure:

1. Display its definition.

2. Select **Remove**.

3. Select **OK**.

8

Defining Basic KPIs

This page introduces key performance indicators (KPIs) in Business Intelligence and describes how to define KPIs that use hardcoded queries.

For a comparison of KPIs with other kinds of model elements, see Summary of Model Options.

Also see Defining KPIs with Filters and Listings and Defining Advanced KPIs.

For information on defining KPIs based on text analytics, see KPIs and Dashboards.

Also see Accessing the BI Samples.

8.1 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.)

8.1.1 Ways to Use KPIs

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. As a consequence, you can define calculated members that are based on KPIs.

To access the values of a KPI value from ObjectScript, use the **%GetKPIValueArray()** method of the KPI class. For an example, see Defining a KPI with a Hardcoded Query.

8.1.2 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.

For additional differences and similarities between KPIs and pivot tables, see Summary of Model Options.

8.1.3 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 Defining Advanced KPIs.

Also note that if you display the KPI in a meter, only the first row of the KPI is used.

8.2 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:

PatientID	Patient Count	Age	Allergy Count
SUBJ_100301	1	2	
SUBJ_100302	1	79	
SUBJ_100303	1	17	2
SUBJ_100304	1	72	
SUBJ_100305	1	66	1
SUBJ_100306	1	44	
SUBJ_100307	1	66	1

In this pivot table, each row represents one row in the source table. The equivalent SQL query would be faster.

8.3 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 page). Each series has a name, which is shown here in the first column.

KPI Values

Series	PatCount	Population
Cedar Falls	1153	90000
Centerville	1120	49000
Cypress	1170	3000
Elm Heights	1163	33194
Juniper	1035	10333
Magnolia	1147	4503
Pine	1147	15060
Redwood	1086	29192
Spruce	1084	5900

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.

8.4 Defining a KPI with a Hardcoded Query

To create a simple KPI that uses a hardcoded query, do the following in Studio:

1. Select **File > New**, select the **Custom** tab, and then select **New Business Intelligence 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.

    *   **Domain** — Localization domain to which this KPI belongs; for details, see Implementing InterSystems Business Intelligence.

    *   **Resource** — Resource that secures this KPI. For information on how this is used, see Setting Up Security.

    *   **Source Type** — Specifies the source of the data for this KPI. Select either **mdx** or **sql**. (For information on **manual**, see 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 Defining KPIs with Filters. 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. Type each action name on a separate line.

    You can edit all these values later as well.

4. Select **Finish**.

The wizard generates a class definition like this:

```
Class MyApp.KPI.MyKPI Extends %DeepSee.KPI
{

Parameter DOMAIN = "MyAppDomain";

Parameter RESOURCE = "KPI_Resource";

/// 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" columnNo="1"/>
<property name="AvgAge" displayName="AvgAge" columnNo="2"/>
</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 Defining KPIs with Filters and Defining Advanced KPIs.

5. Within the `<kpi>` element, add one of the following attribute specifications:

```
mdx="MDX query"
```

Or:

```
sql="SQL query"
```

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:

```
<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.

For requirements, see Requirements for KPI Queries, earlier in this page.

For information on MDX, see *Using InterSystems MDX* and *InterSystems MDX Reference*.

6. Optionally specify class parameters, as described in the next section.

7. Compile the class.

8. Select **View > Web Page**.

You then see something like the following:

KPI Test Page

KPI

Class	MyApp.KPI.MyKPI
Name	MyKPI
Caption	MyCaption

Filters 0 filter(s)

[Submit Query]

Query: mdx

```
SELECT {MEASURES.[%COUNT],MEASURES.[Avg Age]} ON 0,
HomeD.H1.City.MEMBERS ON 1 FROM patients
```

KPI Values 9 series

Series	PatCount	AvgAge
Cedar Falls	110	33.536363636363636
Centerville	105	36.89523809523809524
Cypress	106	33.98113207547169811
Elm Heights	121	38.45454545454545455
Juniper	110	39.69090909090909091
Magnolia	107	36.672897119626168224
Pine	108	37.61111111111111111
Redwood	118	39.08474576271186441
Spruce	115	35.8

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.

The KPI test page provides a convenient way to test the KPI before using it. You can also use the **%GetKPIValueArray()** method of the KPI class. For example:

```
SAMPLES>set
status=##class("HoleFoods.KPIYears").%GetKPIValueArray("HoleFoods.KPIYears",.pValues,$LB("Value"))

SAMPLES>w status
1
SAMPLES>set
status=##class("HoleFoods.KPIYears").%GetKPIValueArray("HoleFoods.KPIYears",.pValues,$LB("Value"))

SAMPLES>w status                                                          1
SAMPLES>zw pValues                                            pValues(1)=$lb("2010")
pValues(2)=$lb("2011")
pValues(3)=$lb("2012")
pValues(4)=$lb("2013")
pValues(5)=$lb("2014")
pValues(6)=$lb("2015")
```

For details, see the class reference for %DeepSee.AbstractKPI.

8.5 Specifying Class Parameters

You can specify some or all of the following class parameters in your KPI class:

DOMAIN

Class Member

```
Parameter DOMAIN = "MyAppDomain";
```

Specifies the localization domain to which this KPI belongs; for details, see *Implementing InterSystems Business Intelligence*.

FORCECOMPUTE

Specifies whether the system should always recompute the values in this KPI when this KPI is used within an MDX query (that is, via the %KPI function). The default is false; when that query is rerun, the system uses cached values instead.

If the KPI uses external data, it may be useful to set *FORCECOMPUTE* equal to true.

LABELCONCAT

Specifies the character used to concatenate labels for an MDX-based KPI that uses CROSSJOIN or NONEMPTY-CROSSJOIN for rows. The default is a slash (/).

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.

RESOURCE

Class Member

```
Parameter RESOURCE = "KPI_Resource";
```

Specifies the resource that secures this KPI. For information on how this is used, see Setting Up Security.

For the *ASYNC* class parameter, see Defining Advanced KPIs.

8.6 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:

```
<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):
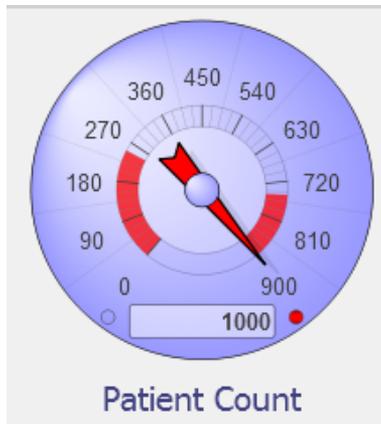
Notice the value box in the speedometer displays the actual KPI value (1000), even though it is higher than the value of `rangeUpper`.

You can also set the range and threshold values programmatically, which is useful if hardcoded values are not appropriate. See 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.

8.7 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:

```
Method %GetMDXContextFilter() As %String
{
    Quit ""
}
```

9

Defining KPIs with Filters and Listings

This page describes options for defining Business Intelligence KPIs that include filters and listings.

Before using this page, see Defining Basic KPIs. For more advanced KPIs, see Defining Advanced KPIs.

KPIs can also define actions. See Defining Custom Actions.

Also see Accessing the BI Samples.

9.1 Introduction to Filters

Typical dashboards include one or more filter controls, with which the user interacts with the widgets on the dashboard. Each filter control is typically displayed as a drop-down list. A filter can filter the data displayed by a widget, or it could affect the data source in some other different manner.

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 (or %FILTER, for an MDX query) that restricts the records used by the query. Then when the user refreshes the dashboard, the system 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 (or %FILTER) 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.

9.2 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.

9.2.1 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.

	Scenario	Syntax	Example
Filter Name	All	MDX level identifiers	`[Region].[H1].[Country]`
Filter Value	User selects a single member	MDX key for a member of that level, which is an expression of the following form: `&[keyval]` where *keyval* is the key value for the member.	`&[USA]`
Filter Value	User selects a range of members	Expression of the following form: `&[keyval1]:&[keyval2]`	`&[2006]:&[2009]`
Filter Value	User selects multiple members	Expression of the following form (for example): `{&[keyval1],&[keyval2]}`	`{&[2006],&[2007],&[2008]}`
Filter Value	User selects a single member and selects **Exclude**	Expression of the following form: `%NOT&[keyval1]`	`%NOT &[2010]`
Filter Value	User selects multiple members and selects **Exclude**	Expression of the following form (for example): `%NOT{&[keyval1],&[keyval2]}`	`%NOT {&[2006],&[2007]}`

Other than the optional `%NOT` string, the filter names and filter values are not case-sensitive.

9.2.2 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 sample dashboard `Demo Filter Interoperability`. 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.

For details on how this works, see Additional MDX KPI Examples, later in this page.

9.3 Defining Filters in an MDX-Based KPI

To define filters in an MDX-based KPI, do the following:

1. *Specify the list of filters.* Each filter can have a logical name (required) and a display name (the same as the logical name by default).

   One option is to hardcode the list of filters. To do so, add a set of `<filter>` elements to the `<kpi>` element. For example:

   ```
   <kpi name="sample KPI"...>
    <filter name="[aged].[h1].[age group]" displayName="Age Group"/>
    <filter name="[gend].[h1].[gender]" displayName="Gender"/>
    <filter name="[homed].[h1].[zip]" displayName="ZIP Code"/>
   ...
   ```

   In this example, the logical filter names are MDX level identifiers. It is convenient, but not required, to specify the logical names in this way.

   You could instead specify filter names at runtime. See Other Options for Defining Filter Names and Items, later in this page.

2. If you add `<filter>` elements, specify the following options as wanted:

   - You can enable users to select only one element from a filter list (instead of multiple elements, which is the default). To do so, add `multiSelect="false"` to the `<filter>` element. For example:

     ```
      <filter name="[homed].[h1].[zip]" displayName="ZipCode" multiSelect="false" />
     ```

     When multiselect is disabled, the **Exclude** option is also disabled.

   - If the filter items are days, you can display a calendar control rather than the default drop-down list. To do so, add `searchType="day"` to the `<filter>` element. For example:

     ```
      <filter name="[birthd].[h1].[day]" displayName="Day"
     filterProperty="Day" searchType="day"/>
     ```

   - A filter can depend upon another filter. To indicate this, use the optional `dependsOn` attribute. For example:

     ```
     <filter name="[homed].[h1].[zip]" displayName="ZIP Code"/>
     <filter name="[homed].[h1].[city]" displayName="City" dependsOn="[homed].[h1].[zip]"/>
     ```

3. *Define the filter items for each filter.* Each filter item can have a logical name (required) and a display name (the same as the logical name by default).

   It is useful if the logical names are the same as MDX member keys; otherwise, more work is needed to construct filter clauses, as described later in this page.

   One option is to implement the **%OnGetFilterMembers()** method. If the logical names of the filters are MDX level identifiers, you can use a version of the following simple implementation:

   Class Member

   ```
   ClassMethod %OnGetFilterMembers(pFilter As %String, Output pMembers As %List,pSearchKey As %String
    = "") As %Status
   {
       set status = $$$OK

       try {
           do ..%GetMembersForFilter("Patients.cube",pFilter,.pMembers,pSearchKey)
       }
       catch(ex) {
           set status = ex.AsStatus()
       }

       quit status
   }
   ```

   Simply replace `Patients` with the name of your cube. For example:

   ```
    do ..%GetMembersForFilter("YourCube.cube",pFilter,.pMembers,pSearchKey)
   ```

   For details on **%OnGetFilterMembers()**, see the first subsection. The example shown here uses the **%GetMembersForFilter()** method; for details, see the second subsection.

   You could instead hardcode the filter items; see Other Options for Defining Filter Names and Items, later in this page.

4. *Modify the query to use the value or values selected by the user.* See Modifying an MDX Query to Use Filter Values, later in this page.

9.3.1 %OnGetFilterMembers() Details

In your KPI class, if you implement the **%OnGetFilterMembers()** method, use the following signature:

```
classmethod %OnGetFilterMembers(pFilter As %String,                         Output pMembers
As %List,                                    pSearchKey As %String = "") as %Status
```

Where:

- *pFilter* is the logical name of a filter.

- *pMembers* specifies the members in a $LISTBUILD list. This list contains both the logical names and the display names. For details, see the class reference for %DeepSee.KPI.

- *pSearchKey* is the search key entered by the user.

Tip: You can use this method in any KPI, regardless of the type of query it uses.

If the logical names of the filters are MDX level identifiers, you can use a simple implementation as shown previously. If not, it is necessary to do more work before getting the members. For examples, see Using Custom Logic to Build the List of Filter Items at Runtime, later in this page.

9.3.2 %GetMembersForFilter() Details

In your KPI class, you can use the **%GetMembersForFilter()** method, which has the following signature:

```
classmethod %GetMembersForFilter(pCube As %String,                          pFilterSpec As
%String,                          Output pMembers,                          pSearchKey
 As %String = "") as %Status
```

Where:

- *pCube* is the logical name of a cube, with `.cube` appended to it.

- *pFilterSpec* is a [MDX level identifier](#) (for example, `"[DateOfSale].[Actual].[YearSold]"`).

- *pMembers*, which is returned as an output parameter, is a list of members in the form required by **%OnGetFilterMembers()**. In this list, the filter items are member keys.

- *pSearchKey* is the search key entered by the user.

This method is more useful for MDX-based KPIs than for SQL-based KPIs. For an SQL-based KPI, you would need to convert the filter values to a form suitable for use in your query.

9.4 Defining Filters in an SQL-Based KPI

To define filters in an SQL-based KPI:

1. *Specify the list of filters.* Each filter can have a logical name (required) and a display name (the same as the logical name by default).

   One option is to specify the `<filter>` element of the `<kpi>` element; see the [previous section](#).

   Or you could instead [specify filter names at runtime](#). See [Other Options for Defining Filter Names and Items](#), later in this page.

2. Within each `<filter>` element, specify the following `multiSelect`, `searchType`, and `dependsOn` attributes as wanted. See the [previous section](#).

3. *Define the filter items for each filter.* Each filter item can have a logical name (required) and a display name (the same as the logical name by default).

   One option is to implement the **%OnGetFilterMembers()** method, described [earlier](#) in this page.

   You could instead hardcode the filter items; see [Other Options for Defining Filter Names and Items](#), later in this page.

4. *Modify the query to use the value or values selected by the user.* See [Modifying an SQL Query to Use Filter Values](#), later in this page.

9.5 Other Options for Defining Filter Names and Items

This section describes other options for defining filter names and filter items. It discusses the following topics:

- [How to specify filter names at runtime](#)

- [How to hardcode the list of filter items in the valueList attribute](#)

- [How to specify the list of filter items via the sql attribute](#)

- [How to use custom logic to create the list of the filter items at runtime](#)

9.5.1 Specifying the Filter Names at Runtime

The easiest way to define the filter names is to hardcode them as described [earlier in this page](#). Another option is to define them at runtime. To do so, override the **%OnGetFilterList()** method of your KPI class. This method has the following signature:

```
classmethod %OnGetFilterList(Output pFilters As %List,pDataSourceName As %String = "") As %Status
```

Where *pFilters* is an array with the following nodes:

- *pFilters* — Specifies the number of filters.

- *pFilters(n)* — Specifies the details for the *n*th filter. This is a $LISTBUILD list that consists of the following items:

  - A string that equals the logical name of the filter.

  - A string that equals the display name of the filter.

  - A string that equals the filter property; the default is the logical name of the filter.

  - 1 or 0 to indicate whether multiselect is enabled for this filter. Use 1 to enable multiselect or 0 to disable it.

pDataSourceName is for future use.

For example, the following **%OnGetFilterList()** adds a filter named `New Filter`:

Class Member

```
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

}
```

For another example, the following **%OnGetFilterList()** uses a utility method to define as filters all the levels in the Patients cube. The utility method **%GetFiltersForDataSource()** (in %DeepSee.Dashboard.Utils) returns a list of filters in the format needed by **%OnGetFilterList()**:

Class Member

```
ClassMethod %OnGetFilterList(Output pFilters As %List, pDataSourceName As %String = "") As %Status
{
    set tSC = ##class(%DeepSee.Dashboard.Utils).%GetFiltersForDataSource("patients.cube",.tFilters)
    quit:$$$ISERR(tSC)

    set i = ""
    for {
        set i = $order(tFilters(i), 1, data)
        quit:i=""

        set pFilters($i(pFilters)) = $lb($lg(data,2), $lg(data,1),,1)
    }
    quit $$$OK
}
```

9.5.2 Hardcoding the List of Filter Items via the valueList Attribute

Another way to build the list of filter items is to specify the `valueList` attribute of the `<filter>` element. Use a comma-separated list of logical names for the filter items. The order of this list determines the order in which filter controls list the filter items.

For example:

XML

```
<filter name="ZipCode"
valueList="&amp;[36711],&amp;[34577],&amp;[38928],&amp;[32006],&amp;[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 names. If you do not specify this attribute, the logical names are also used as the display names.

9.5.3 Retrieving the List of Filter Items via the sql Attribute

Another way to build the list of filter items is to 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 logical names for the filter items. The second column, if included, provides the corresponding display names. If you do not include a second column, the logical names are also used as the display names.

For example:

XML

```
<filter name="ZipCode1" sql="SELECT DISTINCT PostalCode FROM BI_Study.City"/>
```

If you specify the `sql` attribute, do not specify the `displayList` or `valueList` attributes (see the previous subsection).

9.5.4 Using Custom Logic to Build the List of Filter Items at Runtime

Another way to build the list of filter items is to implement **%OnGetFilterMembers()** and your own logic to create the list. For example:

Class Member

```
ClassMethod %OnGetFilterMembers(pFilter As %String, Output pMembers As %List, pSearchKey As %String =
"") As %Status
{
    Set status = $$$OK

    Try {
        If (pFilter = "AgeGroup") {
                set pFilterSpec="[AgeD].[h1].[Age Group]"
        } Elseif (pFilter="Gender") {
                set pFilterSpec="[GenD].[h1].[Gender]"
        } Elseif (pFilter="ZipCode") {
                set pFilterSpec="[HomeD].[h1].[ZIP]"
        }
        do ##class(%DeepSee.KPI).%GetMembersForFilter("Patients",pFilterSpec,.pMembers,pSearchKey)
    }
    Catch(ex) {
        Set status = ex.AsStatus()
    }

    Quit status
}
```

For another approach, you can query the dimension table that holds the level members. (For details on the dimension tables, see Details for the Fact and Dimension Tables.) The following shows an example:

Class Member

```
ClassMethod %OnGetFilterMembers(pFilter As %String, Output pMembers As %List, pSearchKey As %String =
 "") As %Status
{
  set status = $$$OK

  try {
    if (pFilter = "AgeGroup") {
      //get values from level table
      set sql = "SELECT DISTINCT DxAgeGroup FROM BI_Model_PatientsCube.DxAgeGroup"
      set stmt = ##class(%SQL.Statement).%New()

      set status = stmt.%Prepare(sql)
      if $$$ISERR(status) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(status) quit}

      set rs = stmt.%Execute()
      if (rs.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", rs.%SQLCODE, ": ", rs.%Message
quit}

      while(rs.%Next()) {
        set display=rs.DxAgeGroup
        set actual="&[""_display_""]"
        set pMembers($I(pMembers)) = $LB(display,actual)
      }
      if (rs.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rs.%SQLCODE, ": ", rs.%Message quit}

    }
  }
  catch(ex) {
    set status = ex.AsStatus()
  }

  quit status
}
```

Note that this example does not use the *pSearchKey* argument.

9.6 Modifying an MDX Query to Use Filter Values

For an MDX-based KPI, to modify the query to use filters, implement the **%OnGetMDX()** method. If you use the KPI wizard in Studio and you choose the source type as MDX, the generated KPI class includes the following stub definition for this method, which you can use as a starting point:

Class Member

```
/// Return an MDX statement to execute.
Method %OnGetMDX(ByRef pMDX As %String) As %Status
{
    Quit $$$OK
}
```

The variable *pMDX* contains the MDX query specified in the mdx attribute of <kpi>. (If you did not specify that attribute, *pMDX* is null.)

When you implement this method, do the following:

1. If there is no initial query, specify an initial value of the variable *pMDX*. This should be an MDX SELECT query.

    There is no initial query if you do not specify a query in the mdx attribute, as described in the Defining Basic KPIs.

2. Examine each applicable filter and get its value.

    To do this, use the %filterValues property of your KPI instance, as described in the first subsection.

3. Parse each filter value and convert it as needed to a form that can be used in an MDX %FILTER clause (if that is how you intend to use the filter). For a table that lists the suitable forms, see the second subsection.

4. Modify the variable *pMDX* to use the filter values, as appropriate for your needs. In the most common case, append a %FILTER clause for each non-null filter, as follows:

    ```
    %FILTER mdx_expression
    ```

    Where *mdx_expression* is the expression for that filter, as created in the previous step.

    Or create an MDX tuple expression that combines the expressions and then add a single %FILTER clause like this:

    ```
    %FILTER tuple_expression
    ```

    In more exotic cases, you could use the filter values to rewrite the query string in other ways.

5. Return $$$OK (or 1).

The last subsection provides an example.

9.6.1 Accessing Filter Values

When a user selects filter items, the system sets the %filterValues property of your KPI instance. This property is an object (specifically an instance of %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 `filter1`, `filter2`, and `filter3`, then the `%filterValues` property of your KPI instance has properties named `filter1`, `filter2`, and `filter3`.

- The properties of %filterValues contain values shown in the following table:

User Selection in Filter	Value of %filterValues.FilterName	Notes
None	null	
One item	*item*	
Multiple items	{*item1* , *item2*}	Not applicable if multiselect is off
One item and the **Exclude** option	%NOT *item*	Not applicable if multiselect is off
Multiple items and the **Exclude** option	%NOT {*item1* , *item2*}	Not applicable if multiselect is off
Range	*item1* : *item2*	Applies only to a filter defined in a pivot table, which can affect the KPI

9.6.2 Converting Filter Values to MDX Expressions for Use in %FILTER

The following table lists appropriate MDX expressions to use in %FILTER clauses, depending on user selections.

User Selection in Filter	Appropriate MDX %FILTER Expression
None	*none* (do not apply a %FILTER clause in this case)
One item	[dimension].[hierarchy].[level].[member]
One item and the **Exclude** option	[dimension].[hierarchy].[level].[member].%NOT
item1 and *item2*	{item1,item2,...} where each item has the form [dimension].[hierarchy].[level].[member]
Multiple items and the **Exclude** option	EXCEPT([dimension].[hierarchy].[level].MEMBERS,{item,item,item,...}) where each item has the form [dimension].[hierarchy].[level].[member]
A range	[dimension].[hierarchy].[level].[member]:[member]

For an example, see the method **BuildString()** in the sample class BI.Utils.MDXAutoFiltersKPI.

Note that if multiselect is disabled, EXCLUDE is also disabled, and there are fewer possible forms of filter values. In this case, query construction is comparatively simple.

9.6.3 Example

The following implementation of **%OnGetMDX()** applies to a KPI in which multiselect is disabled:

Class Member

```
Method %OnGetMDX(ByRef pMDX As %String) As %Status
{
 if (..%filterValues."[aged].[h1].[age group]"'="")
 {
    set pMDX = pMDX _ " %FILTER [aged].[h1].[age group]." _..%filterValues."[aged].[h1].[age group]"
 }

 if (..%filterValues."[gend].[h1].[gender]"'="")
 {
    set pMDX = pMDX _ " %FILTER [gend].[h1].[gender]." _..%filterValues."[gend].[h1].[gender]"
 }

 if (..%filterValues."[homed].[h1].[zip]"'="")
 {
    set pMDX = pMDX _ " %FILTER [homed].[h1].[zip]." _..%filterValues."[homed].[h1].[zip]"
 }

 quit $$$OK
}
```

Because this method uses delimited property names, it can be confusing to read. Notes:

- `"[aged].[h1].[age group]"` is a valid property name.

- `..%filterValues."[aged].[h1].[age group]"` is a reference to the `"[aged].[h1].[age group]"` property of the %filterValues property of the KPI instance.

9.7 Modifying an SQL Query to Use Filter Values

For an SQL-based KPI, to modify the query to use filters, implement the **%OnGetSQL()** method. If you use the KPI wizard in Studio and you choose the source type as SQL, the generated KPI class includes the following stub definition for this method, which you can use as a starting point

Class Member

```
/// Return an SQL statement to execute.
Method %OnGetSQL(ByRef pSQL As %String) As %Status
{
    Quit $$$OK
}
```

The variable *pSQL* contains the SQL query specified in the `sql` attribute of `<kpi>`. If you did not specify that attribute, *pSQL* is null.

When you implement this method, do the following:

1. If there is no initial query, specify an initial value of the variable *pSQL*. This should be an SQL SELECT query.

    There is no initial query if you do not specify a query in the `sql` attribute, as described in Defining Basic KPIs.

2. Examine each applicable filter and get its value. The options are as follows:

    *   Use the **%GetSQLForFilter()** method, which returns values in a convenient format for use in the WHERE clause of an SQL query. See the first subsection.

    *   Use the `%filterValues` property of your KPI instance, as described in the previous section.

3. Modify the variable *pSQL* to use the filter values, as appropriate for your needs. In the most common case, modify the query to include an SQL WHERE clause.

    In more exotic cases, you could use the filter values to rewrite the query string in other ways.

    The second and third subsections provide examples.

4. Return `$$$OK` (or 1).

Note: An SQL-based KPI cannot have more than 1000 rows; the system automatically limits the number of rows returned.

9.7.1 %GetSQLForFilter()

For an SQL-based KPI, you can use a method to access filter values in a format that is useful for including in your query:

```
method %GetSQLForFilter(sql_field_reference,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 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.

Scenario	Value Returned by Method
User selects `PINE`	`City->Name = 'PINE'`
User selects a range, starting with `MAGNOLIA` and ending with `PINE`	`City->Name = ('MAGNOLIA':'PINE')`
User selects `MAGNOLIA` and `PINE`	`City->Name IN ('MAGNOLIA','PINE')`
User selects `PINE` and selects **Exclude**	`City->Name <> 'PINE'`
User selects `MAGNOLIA` and `PINE` and selects **Exclude**	`City->Name NOT IN ('MAGNOLIA','PINE')`

9.7.2 SQL KPI Example 1

The following example is from the sample class BI.Model.KPIs.DemoSQL. In this case, the filter adds GROUP BY and ORDER BY clauses to the SQL query.

Class Member

```
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 BI_Study.Patient "

    Set where = ""
    //look at %filterValues to see if a filter has been applied to this KPI instance
    If $IsObject(..%filterValues) {
        If (..%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
}
```

9.7.3 SQL KPI Example 2

The following example is from the sample class HoleFoods.KPISQL:

Class Member

```
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 clase 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 Defining Basic KPIs. The **%OnGetSQL()** method modifies that query.

9.8 Additional MDX KPI Examples

This section discusses some other MDX KPI examples.

9.8.1 DemoMDXAutoFilters KPI

The sample class BI.Model.KPIs.DemoMDXAutoFilters KPI is simple but uses a special superclass:

Class Definition

```
Class BI.Model.KPIs.DemoMDXAutoFilters Extends BI.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 BI.Utils.MDXAutoFiltersKPI, which defines the filters and rewrites the query.

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 page.

- 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()** appends the %FILTER clause to the hardcoded query.

9.8.2 DemoInteroperability KPI

The sample class BI.Model.KPIs.DemoInteroperability KPI 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`:

Class Member

```
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 page. This method is as follows:

Class Member

```
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","[profd].[h1].[profession]")
        set pMembers($I(pMembers))=$LB("Diagnoses","[diagd].[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.

Class Member

```
Method %OnGetMDX(ByRef pMDX As %String) As %Status
{
 set yaxis=", NON EMPTY [profd].[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
}
```

9.9 Defining a Listing for a KPI

You can define a KPI so that it includes a listing option. In this case, if the KPI also includes filters, the listing definition must consider the filter selections.

To define the listing, you implement the **%OnGetListingSQL()** method in your KPI class. This method 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:

Node	Node Value
pFilters(filter_name) where *filter_name* is the name of a filter defined in this KPI	Current value of this filter

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:

Node	Node Value
pSelection(`"selectedRange"`)	Currently selected cells in the pivot as a string in the form "startRow,startCol,endRow,endCol" (1-based).
pSelection(`"rowValues"`)	Comma-separated list of the values for the selected rows. In these values, any comma is presented as a backslash (\). If no property of the KPI is configured as the value, then this node contains the series name instead.
pSelection(`"sortColumn"`)	Specifies the number of the column to use for sorting the listing. Use 0 for no sorting.
pSelection(`"sortDir"`)	Specifies the sort direction, `"ASC"` or `"DESC"`

The method should return an SQL SELECT query. In this query, you can also use 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.

9.9.1 Example

The following example is from HoleFoods.KPISQL:

Class Member

```
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
}
```

10

Defining Advanced KPIs

This page describes how to implement Business Intelligence KPIs that use advanced features.

Also see Defining Basic KPIs and Defining KPIs with Filters and Listings.

Also see Accessing the BI Samples.

10.1 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

10.1.1 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.
- %filterValues — Contains the values of any filters. For details, see Defining KPIs with Filters and Listings.

10.1.2 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.

10.1.2.1 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!
        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 "_tROWS_"%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.

10.1.3 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.

10.1.3.1 Example

The following shows a simple example with hardcoded values:

Class Member

```
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
}
```

10.2 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.

  ```
  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.

  ```
  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, the system uses the cached value. Otherwise, the system 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.

Note that specifying the *FORCECOMPUTE* parameter will prevent KPI caching even if *CACHEABLE* is set.

10.3 Defining Asynchronous KPIs

Except for plug-ins, KPIs are executed synchronously.

To modify a KPI so that it executes asynchronously, make the following changes to the KPI class:

- Specify the *ASYNC* class parameter as 1.

- Also modify the KPI so that its results are cached. See the previous section.

  This is required so that the system 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.

11

Defining Plug-ins

This page describes how to define plug-ins, which are a specialized form of Business Intelligence KPI.

Note: Before reading this page, be sure to read all other articles on KPIs.

For a comparison of plug-ins with other kinds of model elements, see Summary of Model Options.

Also see Accessing the BI Samples.

11.1 Introduction

A plug-in is a class that defines one or more computations to use in the Analyzer and in queries. A plug-in has the following features:

- In any given context, the plug-in instance has access to the lowest-level data.

- It can accept parameters.

- It executes asynchronously. When the plug-in is used in a pivot table, the system can display the plug-in 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 plug-in are cached.

Plug-ins 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 plug-in would be suitable in this case.

11.1.1 How Plug-ins Can Be Used

Depending on the plug-in class, you can use it in some or all of the following ways:

- With the MDX %KPI function (which also enables you to specify values for any parameters). This is possible in all cases.

  This means that in all cases, you can define a calculated member that uses the plug-in. (For information on defining calculated members, see Defining Calculated Members.)

- 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 plug-in that cannot be directly used in the Analyzer or in widgets, specify *PLUGINTYPE* as `"Aggregate"`. Or specify *PUBLIC* as 0.

11.1.2 Available Plug-in Classes

The %DeepSee.PlugIn package provides several plug-in classes for you to use in calculated measures. These classes are as follows:

- %DeepSee.PlugIn.Distinct — Gets the count of distinct values for a given level in a given cell.

  %DeepSee.PlugIn.Median — Gets the median value for a given measure, across all the lowest-level records used in a cell.

- %DeepSee.PlugIn.Percentile — Gets a percentile value for a given measure, across all the lowest-level records used in a cell.

  These plug-in classes are defined with *PLUGINTYPE* as `"Aggregate"` and so cannot be directly used in the Analyzer or in widgets. For more details on them, see %KPI in the reference MDX Functions.

- Other classes — More advanced plug-ins for use with Text Analytics. These are used by the Pivot Analysis screens in the Analyzer.

Another sample plug-in class is BI.Model.KPIs.PluginDemo. This plug-in class is defined with *PLUGINTYPE* as `"Pivot"` and thus can be used directly.

11.1.3 Samples That Demonstrate Plug-ins

Within the BI samples, see the dashboards in the `KPIs & Plug-ins` folder:

- The dashboard `HoleFoods Plug-ins` uses the calculated measures `Median Revenue` and `90th Percentile Revenue`, which are defined in the HoleFoods cube. These measures use the %KPI function to retrieve values from the sample plug-in classes %DeepSee.PlugIn.Median and %DeepSee.PlugIn.Percentile.

- The dashboard `Patients Plug-ins` has a pivot table that uses the calculated measures `Median Test Score` and `90th Percentile Test Score`. These calculated members are defined in the Patients cube, in a similar manner to the ones in the previous bullet.

  This dashboard contains another pivot table, which directly uses the plug-in defined by the sample class BI.Model.KPIs.PluginDemo.

11.2 Requirements for a Simple Plug-in

To define a simple plug-in, 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:

Class Member

```
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 Reference Information for KPI and Plug-in Classes.

- Specify the *BASECUBE* class parameter. For a simple plug-in, specify the logical name of a single cube or subject area. (But also see Creating a Plug-in for Multiple Cubes, later in this page.)

- Specify the base MDX query to use. Either specify the mdx attribute of <kpi> or implement the **%OnGetMDX()** method in the following generic way:

```
Method %OnGetMDX(ByRef pMDX As %String) As %Status
{
    Set pMDX = "SELECT FROM "_..#BASECUBE
    Quit $$$OK
}
```

InterSystems IRIS automatically applies context information (row, column, and filter) to this base query.

- Specify the fields that need to be available to the **%OnCompute** method. These can be fields in the source table of the cube or can be fields in the fact table. You can specify a hardcoded list, or you can use a callback to define the list of fields.

  To specify these fields:

  – If the fields you want to use are in the fact table, specify the *LISTINGSOURCE* class parameter as "FactTable". (For details on the fact table, see Details for the Fact and Dimension Tables.)

    If you omit this parameter or specify it as "SourceTable", the plug-in queries the source table of the given cube.

  – If you want to specify a hardcoded list of field names, specify the *LISTINGFIELDS* class parameter. Specify a comma-separated list of fields to use.

    For example:

Class Member

```
Parameter LISTINGFIELDS = "Field1, Field2, Field3";
```

You can specify an alias for any field. For example:

Class Member

```
Parameter LISTINGFIELDS = "Field1, Field2 as FieldAlias, Field3";
```

You can also use arrow syntax and SQL functions, as with other listings.

    If you use arrow syntax, be sure to specify an alias for the field.

  – Or if you want to define the list of fields programmatically, implement the **%OnGetListingFields()** method. For example, the following method causes the plug-in to retrieve a single field:

Class Member

```
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.

Note: For a plug-in, the *LISTINGFIELDS* parameter and the **%OnGetListingFields()** do not define a detail listing or any listing fields. These options only define the fields that are available to the **%OnCompute()** method.

• Implement the **%OnCompute()** method. The following section provides details on this task.

• Optionally specify the *PLUGINTYPE* and *PUBLIC* class parameters. See How Plug-ins Can Be Used.

11.3 Implementing %OnCompute()

For each pivot table cell where the plug-in is used, the plug-in performs either a DRILLTHROUGH or DRILLFACTS query (depending on the value of *LISTINGSOURCE*) and returns the fields specified by *LISTINGFIELDS* or **%OnGetListingFields()** (as applicable). It then passes the field values to the **%OnCompute()** method. This method has the following signature:

```
Method %OnCompute(pSQLRS As %SQL.StatementResult, pFactCount As %Integer) As %Status
```

Where:

• *pSQLRS* is an instance of %SQL.StatementResult that contains the fields specified by *LISTINGFIELDS* or **%OnGetListingFields()**.

For information on using this class, see Using Dynamic 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, **%OnGetListingFields()** retrieves a single field, MxTextScore. In this case, *pSQLRS* has a property named MxTextScore.

3. Perform the desired computations.

4. Set the properties of the plug-in instance, as described in Defining Advanced KPIs. At a minimum, set the following properties:

   • %seriesCount — Specifies the number of series (rows) in this plug-in.

   InterSystems recommends that plug-ins have only one series. (For plug-ins with *PLUGINTYPE* equal to "Pivot", when a user drags and drops a plug-in 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:

ObjectScript

```
// 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()) {
  set n = n + 1

  set testscore = pSQLRS.MxTestScore
  if (testscore>95) {
    Set highcount = highcount + 1
  }
 if (pSQLRS.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", pSQLRS.%SQLCODE, ": ", pSQLRS.%Message
quit}

}
set ..%data(1,"HighScoreCount") = highcount
```

This is an extract from the sample class BI.Model.KPIs.PluginDemo, which is available in the Analyzer for use with the Patients cube.

11.4 Indicating State of Completion

Plug-ins are executed asynchronously. When a query containing plug-ins is executed, the query can be complete before the plug-ins have completed execution. In this case, there are cells whose results are pending. Within these cells, you can display the plug-in 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:

ObjectScript

```
// 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.

11.5 Creating a Plug-in for Multiple Cubes

The previous sections describe how to create a plug-in that can be used with a single cube or subject area. You can also create a plug-in 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 plug-in 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 cubes and subject areas can use the plug-in.

- 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 plug-in within the Analyzer (if applicable), the system passes the name of the current cube or subject area to this filter. Similarly, when you use this plug-in within an MDX query, the FROM clause of the query determines the value of this filter.

- Implement the **%OnGetMDX()** method so that it uses the value of the %cube filter. For example:

  ### Class Member

```
Method %OnGetMDX(ByRef 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.

  – Programmatically determine the fields to use.

For examples, see %DeepSee.PlugIn.Median and %DeepSee.PlugIn.Percentile.

11.5.1 Determining the Listing Fields Programmatically

If the query for the plug-in 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.

- Implement the **%OnGetListingSQL()** method as follows:

  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.

•   Optionally implement the **%OnGetListingOrderBy()** and **%OnGetListingMaxRows()** callbacks. For details, see the class reference for %DeepSee.KPIPlugIn.

For examples, see %DeepSee.PlugIn.Median and %DeepSee.PlugIn.Percentile. Also see the class reference for the **%DeepSee.Utils** class.

11.6 Filtering the Listing

Plug-ins provide a feature that is not available in other scenarios: namely, the ability to specify which records to use when a detail listing is displayed. By default, when a user requests a detail listing for a given cell or set of cells in the results, the system displays a listing that shows all the records associated with those cells. In some cases, however, it is preferable to show a subset of them. For example, the sample class BI.Model.KPIs.PluginDemo has a plug-in property called HighScoreCount. The following shows an example MDX query that uses this plug-in property as a measure:

```
SELECT NON EMPTY {[Measures].[%COUNT],%KPI("PluginDemo","HighScoreCount",,"%CONTEXT")} ON 0,
NON EMPTY [AllerSevD].[H1].[Allergy Severities].Members ON 1 FROM [PATIENTS]
```

```
                         Patient Count       HighScoreCount
1 Nil known allergi            158                      12
2 Minor                        113                       7
3 Moderate                     103                       5
4 Life-threatening             133                       9
5 Inactive                     122                       8
6 Unable to determi            119                       6
7 No Data Available            385                      29
```

Consider the row for Nil known allergies. If you display a listing for either cell, by default, the system displays a listing that consists of 158 records, because there are 158 patients with no known allergies. But the purpose of the HighScoreCount measure is to count the patients with scores above a given threshold, so when we display the detail listing for the cell HighScoreCount in this row, we might prefer to see only the patients with scores above that threshold.

To apply this sort of filtering to a plug-in, include the following logic in your implementation of **%OnCompute()**, for any source class ID that *should* be shown in the listing:

```
    set ..%data("IDLIST",pluginProperty,sourceClassID) = ""
```

Where *pluginProperty* is the name of the plug-in property that should use this filtering, and *sourceClassID* is the ID in the source class. (The ID should be a source class ID even if plug-in otherwise uses the fact class. To make the source class ID available to the plug-in, add %sourceId to the field list.)

For a given plug-in property, if %data("IDLIST",*pluginProperty*) is not defined, the listing shows all the records associated with the given cell or cells.

11.6.1 Example

To see an example, edit the sample class BI.Model.KPIs.PluginDemo as follows:

1. Change *LISTINGFIELDS* to be the following:

```
    Parameter LISTINGFIELDS As STRING = "%sourceId,MxTestScore";
```

2. Find the part of **%OnCompute()** that sets the *highcount* variable, and modify it as follows:

```
if (testscore>95) {
    Set highcount = highcount + 1
    Set tHighScoreId = pSQLRS.sourceId
    Set ..%data("IDLIST","HighScoreCount",tHighScoreId)=""
}
```

3. Save and recompile the class.

Then, in the Analyzer, create a pivot table that uses both properties of this plug-in (for purposes of comparison). Select a cell that displays the `HighScoreCount` property, display a listing, and notice that only patients with a high score are shown. For contrast, select a cell that displays the `PatientCount` property and display a listing for that. In this case, you will see patients with all scores.

11.7 Available Error Logging

If a plug-in encounters an error, the system writes to the error log file in the manager's directory. The name of this file is DeepSeeTasks_*namespace*.log.

11.8 Defining a Calculated Member That Uses a Plug-in

For any plug-in (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 described in Defining a Calculated Measure.

- For **Expression**, specify an MDX expression of the following form:

```
%KPI(pluginname,propertyname,seriesname,"%CONTEXT")
```

Where *pluginname* is the name of the plug-in, *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 plug-in.

`"%CONTEXT"` is a special parameter that provides row, column, and filter context to the plug-in; this information is passed to the base MDX query used by the plug-in.

For example (for a plug-in with only 1 series):

```
%KPI("PluginDemo2","Count",,"%CONTEXT")
```

For plug-ins with *PLUGINTYPE* equal to `"Pivot"`, when a user drags and drops a plug-in property, the Analyzer automatically uses syntax like this in the underlying MDX query that it generates.

For additional options, see %KPI.

12

Using Cube Inheritance

In some cases, it is necessary to define multiple similar cubes. Business Intelligence provides a simple inheritance mechanism to enable you to define these cubes more easily. This page describes how to use this mechanism. In some of the procedures described, you cannot use the Architect but must instead use an Integrated Development Environment (IDE) with support for ObjectScript: Visual Studio Code or Studio.

12.1 Introduction to Cube Inheritance

To use the cube inheritance mechanism:

1. Define one cube class according to the procedure described in Defining a Cube. This cube should contain the core items expected in all the similar cubes, and will serve as the parent cube.

2. *Optionally*, you can mark this cube as abstract so that it cannot be used directly.

    a. On your BI instance, open a an IDE session and ensure you are in your analytics namespace.

    b. In the XData definition, locate the `<cube>` element and set `abstract="true"` in its list of attributes. This prevents the compiler from validating the cube, the Analyzer from displaying it, and the user from executing queries against it.

    c. Save your changes.

3. Define the child cubes in their own cube classes. This procedure deviates from the procedure used to define the parent cube.

    a. On your BI instance, log in to the Management Portal as a user with administrative privileges.

    b. Navigate to `Home > Analytics > Architect`.

    c. Click the **New** button.

    d. For **Definition Type**, select **Cube**.

    e. Enter a **Cube Name** and, optionally, a **Display Name** of your choosing.

    f. For **Cube Source**, select **Cube**.

    g. In the **Base Cube** field, enter the logical name of the parent cube created previously.

    h. Enter a **Class Name for the Cube** and, optionally, a **Class Description**.

    i. Click **OK** to create the child cube. By default, cubes created in this way inherit all definitions from their parent cubes. Each cube may only have one Base Cube, but Base Cubes may themselves inherit from other cubes.

4. Make sure that the parent cube is compiled before any of its child cubes. In the **Details** pane to the right of the model viewer, enter the logical name of the parent cube into the **Depends On** field.

5. Optionally redefine any dimension, measure, or other *top-level* element specified in the parent cube. To do so, specify a definition in the child cube and use the same logical name as in the parent cube. The new definition completely replaces the corresponding definition from parent cube. Also see Hiding or Removing Items.

6. Optionally specify additional definitions (dimensions, measures, listings, and so on) in the child cubes.

Note the cube inheritance mechanism has no relationship with class inheritance. Each subcube has its own fact table and indexes, and (at run time) these are independent of the parent cube. The cube inheritance mechanism is used only at build time, and affects only the definitions in the cubes.

12.2 Cube Inheritance and the Architect

When you display a subcube in the Architect, you can view the inherited elements, override the inherited elements, and define new elements. The left area displays the source class for the cube, so that you can drag and drop items from this class for use as elements in the subcube. The middle area of the Architect displays both inherited items (in italic) and locally defined items (in plain font).

When you select an inherited item in the middle area of the Architect, the **Details** pane displays the following at the top:
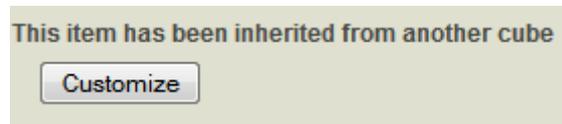
The rest of the **Details** pane is read-only.

The following subsections describe how to redefine inherited items, remove overrides, and add local elements. For information on compiling and building cubes, see Compiling and Building Cubes; subcubes are handled in the same way as other kinds of cubes.

12.2.1 Redefining an Inherited Element

To redefine an inherited element, click the element in the middle area and then click the **Customize** button in the **Details** pane on the right. If the element you want to customize is not a top-level element, then click the top-level element that contains it, and then click the **Customize** button in the **Details** pane on the right.

When you click **Customize**, the Architect copies the definition from the parent cube to the subcube, creating a local definition that overrides the inherited definition. You can now edit the local definition.

12.2.2 Removing an Override

To remove a local definition that overrides an inherited definition:

1. Click the X in the row for that item, in the middle area of the Architect.

2. Click **OK** to confirm the action.

12.2.3 Adding a Local Element

To add a local element to the subcube, either use the **Add Element** button or use the standard drag-and-drop actions, as described in Adding Items to a Cube.

12.3 The %cube Shortcut

If the parent cube contains any source expressions that use the variable *%cube*, the system cannot build the cube unless you do either of the following:

- Modify the child cube class that it extends the parent cube class. (That is, use class inheritance.)

- Override the elements that use *%cube*. In your local definitions, replace *%cube* with the usual full syntax (`##class(package.class)`).

12.4 Hiding or Removing Items

To hide a measure, dimension, calculated member, or calculated measure, add an override for that item and select the **Hidden** option in your override. The item is still created but is not shown in the Analyzer.

To hide levels or hierarchies, redefine the dimension that contains them. To do so, add an override for the dimension. In your override, define all the hierarchies and levels that this dimension *should* contain. This new dimension completely replaces the inherited dimension. The new dimension can, for example, define fewer or more levels than the corresponding dimension in the parent cube.

12.5 Inheritance and Relationships

If the parent cube has any relationships, note that those relationships are inherited, but necessarily not in a useful way, because the relationships always point to the original cubes.

For example, suppose that two cubes (Patient and Encounter) are related to each other, and you create subcubes (Custom-Patient and CustomEncounter) for each of them. By default, CustomPatient has a relationship that points to the original Encounter cube. Similarly, the CustomEncounter cube has a relationship that points to the Patient cube. If you want a relationship between CustomPatient and CustomEncounter, you must define that relationship explicitly in the subcubes.

13

Defining Intermediate Expressions

In some cases, you might have multiple measures or dimensions in Business Intelligence 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.

Also see Accessing the BI Samples.

13.1 Defining and Using Intermediate Expressions in an IDE

To define an expression in an IDE:

1. In an IDE, open the cube class.

2. Find the `<cube>` starting element:

    ```
    <cube name="Patients" displayName="Patients"
       owner="_SYSTEM"
        sourceClass="BI.Study.Patient"
        nullReplacement="None"
        defaultListing="Patient details">
    ```

3. After the greater than sign (`>`), add one or more `<expression>` elements. For example:

    ```
    <expression name="patDetails" sourceExpression='%cube.GetPatientDetails(%source.PatientID)'  />
    ```

    At a minimum, the `<expression>` element must have the following attributes:

Attribute	Value
name	Name of this expression.
sourceExpression	Optionally specify an ObjectScript expression that returns a single value for any given source record. If you specify this, do not specify sourceProperty. For an <expression> 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 <expression>). An <expression> element can refer to another <expression> element.
sourceProperty	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.

For additional options for sourceExpression and sourceProperty, see Specifying the Source Values for a Dimension or Level and Details for Source Expressions.

4. Within the definition of a measure, dimension, level, or another <expression>, use the following syntax to refer to an expression:

```
%expression.expressionName
```

5. Save and recompile the class.

6. Rebuild the cube or perform a selective build of any cube elements that use the expression.

13.1.1 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:

```
%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 BI_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 is built, 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 an editor, open the Patients cube class, BI.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:

Class Member

```
ClassMethod GetPatientDetails(patientID As %String) As %String
{
    New SQLCODE
    &sql(SELECT Profession->Industry,Profession->Profession,FavoriteColor
    INTO :Industry,:Profession,:FavoriteColor
    FROM BI_Study.PatientDetails
    WHERE PatientID=:patientID)

    If (SQLCODE'=0) {
        Set Industry="",Profession="",FavoriteColor=""
    }
    Set ReturnValue=$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 or perform a selective build of the redefined levels.

13.2 Defining Intermediate Expressions in Architect

To define an expression in the Architect:

1. Open a cube definition in the Architect.

2. Select **Add Element**.

   The system displays a dialog box.

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

4. Select **Expression**.

5. Select **OK**.

6. In the **Details** tab on the right, specify values for each of the fields.

14

Other Options

This page describes how to use other options for Business Intelligence cubes and subject areas.

Also see Accessing the BI Samples.

14.1 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.

Or you can specify the `maxFacts` attribute as follows:

1. In an IDE, open the cube class.

2. Find the `<cube>` element:

   ```
   <cube name="HoleFoods"
   caption="HoleFoods Sales"
   defaultListing="Listing"
   nullReplacement="Missing Value"
   actionClass="HoleFoods.KPIAction"
   sourceClass="HoleFoods.Transaction">
   ```

3. Add the `maxFacts` attribute to this element:

   ```
   <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.

14.2 Restricting the Records Used in the Cube

By default, the system 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 this in either or both of the following ways:

- By specifying the **Build restriction** option for the cube definition, as described in Defining Models for InterSystems Business Intelligence.

  This option has no effect if the cube is based on a data connector.

- By defining the **%OnProcessFact()** callback of the cube class.

14.2.1 %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:

```
classmethod %OnProcessFact(pID As %String, ByRef pFacts As %String,        Output pSkip As %Boolean,
 pInsert As %Boolean) as %Status
```

The system 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:

Node	Value
`pFacts(`*factName*`)` where *factName* is the name of the level or measure in the fact table, as specified by the **Field name in fact table** option.	Value of that level or measure. For example, `Magnolia` or 47.

- *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:

Class Member

```
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 **Field name in fact table** option as `DxColor` for the favorite color level.

Note: If your **%OnProcessFact()** evaluates records based on the levels or measures you specify using the *pFacts* array, you must include the corresponding cube elements whenever you perform a Selective Build for the cube. In the preceding example, the user must include `DxColor` in any Selective Build for the cube. Otherwise, every record will yield an error because the value of `pFacts("DxColor")` will be undefined.

For another example, the following callback would cause the cube to ignore all records whose ID was less than 1000000:

Class Member

```
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
}
```

14.3 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 an IDE as follows:

1. In an IDE, open the cube class.

2. Find the section that defines the level:

    ```
    <level name="MyLevel" displayName="MyLevel" ... >
    </level>
    ```

3. Just before the `</level>` line, add a set of `<member>` elements like the following:

    ```
    <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. In `<member>`, `name` specifies the name of the member, `displayName` specifies an optional localized display name for the member, and `spec` specifies an optional key specification for this member.

    The rules for `name` and `spec` are as follows:

Scenario	Requirement	Result
Specify only `name`	The value of `name` must exactly match a value of the source property or source expression, including case.	The value of `name` becomes the key.
Specify both `name` and `spec`	The value of `spec` must exactly match a value of the source property or source expression, including case.	The value of `spec` becomes the key.

    Additional 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).

- • A `<member>` cannot have the same `name` as a `<property>` within the same level.

4. Save and recompile the class.

5. Rebuild the cube or perform a selective build of the level.

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.

14.3.1 XML Reserved Characters

When you edit a cube class in an IDE, you cannot use XML reserved characters in the values of `name`, `displayName`, or other attribute values. Instead substitute as follows:

Reserved Character	Use This Instead
<	<
&	&
"	"
'	'

It is not an error to use >, but you can use `>` instead of that character.

When you type these characters into input fields in the Architect, the Architect automatically makes these substitutions. If you examine a cube class in an IDE, you might notice that the Architect also makes other substitutions. The preceding table lists only the substitutions that are necessary.

14.4 Adding Custom Indexes to the Fact Table

The system automatically defines all the indexes that it needs. However, you can use the fact table directly for your own purposes, and if you do, you might need additional indexes. To add them, edit the cube class in an IDE and add `<index>` elements as needed.

See Reference Information for Cube Classes.

14.5 Customizing Other Cube Callbacks

The class %DeepSee.CubeDefinition provides callback methods that you can override to customize the cube behavior further. This section describes some commonly overridden methods.

Also see %OnProcessFact() Callback and Filtering a Cube or Subject Area Dynamically, elsewhere in this page.

See the class reference for %DeepSee.CubeDefinition for additional options.

14.5.1 %OnAfterProcessFact() Callback

The **%OnAfterProcessFact()** callback enables you to add custom code that runs *after* any given record is added or updated in the fact table of the cube.

```
ClassMethod %OnAfterProcessFact(pID As %String, ByRef pFactArray As %String, pUpdateStatus As %Status)
 as %Status
```

The system passes the following information to this callback:

- *pID* is the id of the row in the source data being processed.

- *pFacts* is an array containing the values that will be used for the row, subscripted by fact name.

- *pUpdateStatus* is the status about to be returned by **%ProcessFact()**. If an error is passed in, this error will already be logged in the Business Intelligence logs and the **^DeepSee.BuildErrors** global.

The **%ProcessFact()** method ignores the value returned by this method.

14.5.2 %OnGetDefaultListing() Callback

The **%OnGetDefaultListing()** callback enables you to programmatically specify the name of the listing to use, in the case where the system uses the default listing. The signature of this method is as follows:

```
ClassMethod %OnGetDefaultListing() as %String
```

This callback is called when a user requests the default listing and has no effect when a specific listing is requested. The following shows an example:

```
ClassMethod %OnGetDefaultListing() As %String {     Quit "Listing By Product" }
```

You could use this, for example, to check the role to which a user belongs and display a suitable listing for that role.

14.5.3 %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:

```
ClassMethod %OnExecuteListing(pSQL As %String) as %Status
```

The system calls this method immediately before it executes a listing query. When the system calls this method, it passes the value *pSQL*, which is the listing query that will be executed.

14.5.4 %OnAfterBuildCube() Callback

The **%OnAfterBuildCube()** callback, if defined, is called after the cube is built.

```
ClassMethod %OnAfterBuildCube(pBuildStatus As %Status, pBuildErrors As %Boolean = 0) As %Status
```

The system calls this as the last step in the **%BuildCube** method in %DeepSee.Utils. The system passes the following information to this callback:

- *buildStatus* is the status of the cube build thus far

- *factErrorCount* is the count of any facts for which there were build errors.

Your implementation should return an instance of %Status.

This method is called before the lock on the cube build is released, which means that only one process can execute the callback at once.

14.6 Filtering a Cube or Subject Area Dynamically

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. This callback is also available in cube classes. Thus you can filter both cubes and subject areas dynamically.

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. For example:

Class Member

```
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 any filtering if the user belongs to the **%All** role:

Class Member

```
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:

Class Member

```
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

}
```

A

Reference Information for KPI and Plug-in Classes

This page contains reference information for Business Intelligence KPI and plug-in classes.

You must recompile a KPI class after making any change.

A.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 `<KPI>` and this element must follow the requirements described in this page.

- The class can define several class parameters. See Class Parameters for the KPI Class.

The requirements for a plug-in 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 Plug-ins.

A.2 Common Attributes in a KPI or Plug-in

Most of the elements in a KPI or plug-in have the following attributes, which are listed here for brevity:

Attribute	Purpose
name	Logical name of the element.

Attribute	Purpose
`displayName`	(Optional) Localized name of this element for use in user interfaces. If you do not specify this attribute, the system instead uses the value specified by the `name` attribute. For details, see Performing Localization.
`description`	(Optional) Description of this element.
`disabled`	(Optional) Controls whether the compiler uses this element. If this attribute is `"true"` then the compiler ignores it. By default, this attribute is `"false"`

A.3 <kpi>

The `<kpi>` element is the root element in the XData block in a KPI or plug-in class. This element contains the following items:

Attribute or Element	Purpose
`name`, `displayName`, `description`, `disabled`	See Common Attributes in a KPI, earlier in this page.
`caption`	Not used.
`sourceType`, `mdx`, `sql`	See Specifying the Query for the KPI.
`rangeLower`, `rangeUpper`, `thresholdLower`, `thresholdUpper`	See Specifying Ranges and Thresholds for Speedometers.
`actionClass`	(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.
`<property>`	(Optional) You can include zero or more `<property>` elements, each of which corresponds to a column of the query that the KPI uses. You cannot display the query results unless you specify `<property>` elements.
`<action>`	(Optional) You can include zero or more `<action>` elements, each of which is available for use when you create a scorecard based on this KPI.
`<filter>`	(Optional) You can include zero or more `<filter>` elements, each of which is available for use when you create a scorecard based on this KPI.

A.4 <property>

Within `<kpi>`, a `<property>` element contains the following attributes:

<filter>

Attribute	Purpose
`name`, `displayName`, `description`, `disabled`	See Common Attributes in a KPI, earlier in this page. 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.
`columnNo`	Number of the column in the query that contains the data for this property. The first data column is 1.
`format`	(Optional) Default numeric format for this property, when this KPI is displayed in a pivot table widget. For example: `format="##.###"`
`style`	(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="color:red"`
`headerStyle`	(Optional) CSS style to apply to the corresponding header cells, when this KPI is displayed in a pivot table widget. For example: `style="color:red;font-style:italic"`
`defaultValue`	Not used.

Note that the KPI test page ignores the `format`, `style`, and `headerStyle` attributes. These attributes affect only pivot table widgets.

A.5 <filter>

Within `<kpi>`, a `<filter>` element contains the following attributes:

Attribute	Purpose
`name`, `displayName`, `description`, `disabled`	See Common Attributes in a KPI, earlier in this page.
`filterProperty`	Logical name of the `<property>` element that this filter controls. This option is for use when retrieving the filter values programmatically.
`defaultValue`	(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 `<filter>` element.
`sql`, `valueList`, `displayList`	See Defining KPI Filters.
`searchType`	(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 `"day"`
`dependsOn`	(Optional) Specifies the name of another filter on which this filter depends. For example, if your KPI has filters named `State` and `City`, you might include `dependsOn="State"` in the definition of the `City` filter.

A.6 <action>

Within `<kpi>`, an `<action>` element contains the following attributes:

Attribute	Purpose
`name`, `displayName`, `description`, `disabled`	See Common Attributes in a KPI earlier in this page. The `name` attribute cannot be any of the following (not case-sensitive): `applyFilter`, `setFilter`, `refresh`, `showListing`, `viewDashboard`, `navigate`, `newWindow`, `rowCount`, `rowSort`, `colCount`, `colSort`.

Note that the `<action>` element does not define the actions themselves, which you define within the **%OnDashboardAction()** callback method of the class; see Defining the Available Actions.

The `<action>` element is necessary to notify the Dashboard Designer which actions are defined in a KPI or plug-in, so that you can select them for use.

B

Generating Secondary Cubes for Use with Text Analytics

Important: The integration of text analytics within InterSystems IRIS Business Intelligence uses InterSystems IRIS Natural Language Processing (NLP), which InterSystems has deprecated. As a result, this feature may be removed from future versions of InterSystems products. The following documentation is provided as reference for existing users only. Existing users who would like assistance identifying an alternative solution should contact the WRC.

This page assumes that you have added an NLP measure to a Business Intelligence cube (and have created NLP dimensions that use that measure), as described in Using Text Analytics in Cubes. This page describes how to generate secondary cubes that analyze entity occurrences and dictionary matching results. Always build the main cube before building these cubes.

The approach in this page is an alternative to using plug-ins as described in Adding Measures to Quantify Entity Occurrences and Adding Measures to Quantify Matching Results, in Using Text Analytics in Cubes. Plug-ins are a better approach, because the secondary cubes must be rebuilt manually whenever the main cube is rebuilt or synchronized.

Also see Accessing the BI Samples.

B.1 Entity Occurrence Cube

To generate a cube that represents entity occurrences, use the following command in the Terminal:

ObjectScript

```
d ##class(%iKnow.DeepSee.CubeUtils).CreateEOCube(cubename,measurename)
```

Where *cubename* is the name of the cube that contains an NLP measure, and *measurename* is the name of the NLP measure.

This method generates a read-only class that provides access to the 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 NLP 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 *BaseCubemeasurename*EO where *BaseCube* is the logical name of the base cube and *measurename* is the name of the NLP 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 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 concepts and relations. For information on these terms, see Logical Text Units Identified by NLP.

- 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:

	Airplane	Balloon	Glider	Gyrocraft	Helicopter	Ultralight
airplane	4,980		15		1	5
pilot	4,322	22	98	9	478	4
engine	1,438		8	1	121	1
accident	1,181	4	22	3	164	
flight	1,178	2	21	2	156	1
time	887	2	13	3	118	
runway	829		20	1	8	
helicopter	47				701	
examination	601		6	2	92	

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.

B.2 Matching Results Cube

To generate a cube that represents matching results, use the following command in the Terminal:

ObjectScript

```
d ##class(%iKnow.DeepSee.CubeUtils).CreateMRCube(cubename,measurename)
```

Where *cubename* is the name of the cube that contains an NLP measure, and *measurename* is the name of the NLP measure.

This method generates a read-only class that provides access to the 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 NLP 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 *BaseCubemeasurename*MR where *BaseCube* is the logical name of the base cube and *measurename* is the name of the NLP 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 entities, CRCs, paths, and sentences. For information on these terms, see Logical Text Units Identified by NLP.

  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:

		Accident		Incident	
		Count	Score	Count	Score
Injuries	fatal	106	85.31		
	minor	152	151.50	1	1
	none	713	340.52	24	12.85
	serious	93	91.51		

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.