InterSystems®
Caché

Using iKnow

Version 2018.1
2024-04-03

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

List of Tables

About This Book

This book describes how to use the iKnow semantic analysis engine to access and analyze unstructured data. Commonly, unstructured data consists of a large number of text sources, such as a collection of newspaper articles or a collection of doctors' notes. You load this text data into iKnow, and then use iKnow to retrieve meaningful information. iKnow operates on data loaded from source texts; it does not modify source texts. iKnow can perform semantic analysis on texts in Dutch (nl), English (en), French (fr), German (de), Japanese (ja), Portuguese (pt), Russian (ru), Spanish (es), Swedish (sv), and Ukrainian (uk).

The book addresses a number of topics:

- Introductory Chapters:

    – A Conceptual Overview, which describes the iKnow approach to unstructured data and iKnow architecture. It describes both what iKnow is and what iKnow is not, so that users can determine whether iKnow is the best fit for their text access application.

    – iKnow Implementation, which describes the implementation of iKnow software in the ObjectScript environment and describes data source considerations. Data sources can be text files, SQL records, globals, RSS feeds, or any other type of text source.

- iKnow Interfaces:

    – iKnow Architect, which describes the Management Portal interface for creating iKnow components and populating them with data.

    – REST Interface, which describes the REST API interface for performing many iKnow operations.

    The operations performed through these interfaces are described in greater detail in the following chapters, which are specific for that type of operation.

- Setting Up an iKnow Environment and Loading Text Data into an iKnow Domain which describe how to write programs to get unstructured data into iKnow.

- iKnow Queries describes how to write programs that can be run against data loaded into iKnow. Attributes allow iKnow queries to distinguish Negation and Sentiment attributes; terms flagged with these attributes affect query interpretation of a path or sentence. Dominance and Proximity provide more sophisticated calculations for comparing text elements, and Custom Metrics gives the user the means to extend and customize the calculations used for comparing text elements in queries.

- Filtering Sources describes how to use various filters to limit the scope of queries to a subset of the data sources loaded into iKnow. Text Categorization describes how to automatically assign data sources to categories based on an analysis of the contents of the source. Once assigned, this category metadata value can be used to filter sources or when querying sources.

- The two Smart Matching chapters describe how to create a dictionary, and then how to use a dictionary to match against data sources loaded into iKnow.

- The User Interfaces chapter describes several sample GUI interfaces for retrieving information. These interfaces use the queries, filters, and dictionaries described in the prior chapters.

- iKnow Tools describes the Terminal iKnow Shell interface for displaying iKnow components and data. This is a useful tool, but provides no additional iKnow functionality. This chapter also describes the Data Upgrade utility for use on iKnow domains created under an earlier version of iKnow.

- The iKnow KPIs and DeepSee Dashboards chapter describes how to use iKnow ObjectScript queries as data sources for KPIs (key performance indicators) and how to display these KPIs on DeepSee dashboards.

- The Web Services chapter describes using iKnow with Internet data.

- The final two chapters describe advanced topics. Customizing iKnow describes how to create additional iKnow text processing facilities to supplement those supplied with iKnow. Language Identification describes how to work with source texts in more than one language and with texts in languages that need special processing.

- The Domain Parameters appendix provides a comprehensive list of available domain parameters. You can set these parameters to customize a domain, or set them for all domains systemwide.

- The DeepSee Cube Integration (Deprecated Form) appendix describes the deprecated form of integration between iKnow and DeepSee cubes.

This book also describes iFind, an SQL facility for performing text search operations.

For a detailed outline, see the Table of Contents.

For information on using the newer form of iKnow/DeepSee cube integration, see "Using Unstructured Data in Cubes" in the *Advanced DeepSee Modeling Guide*.

For general information, see *Using InterSystems Documentation*.

1

Conceptual Overview

The iKnow semantic analysis engine is used to analyze unstructured data, data that is written as text in a human language such as English or French. By providing the ability to rapidly access and analyze this type of data, iKnow allows you to work with *all* of your data. iKnow does not require you to have any prior knowledge of the contents of this data, or even know what language it is written in, so long as it is one of the languages that iKnow supports.

Commonly, unstructured data consists of multiple source texts, often a very large number of texts. A source text is commonly running text divided by punctuation into sentences. A source text can be a file, a record in a SQL result set, or a web source such as a series of blog entries.

1.1 A Simple Use Case

To see how iKnow handles unstructured data, consider the following sentence:

General Motors builds their Chevrolet Volt in the Detroit-Hamtramck assembly plant.

iKnow first divides a text into sentences; it then analyzes each sentence semantically. It does not need to "understand" or look up the words in the sentence. iKnow then indexes the resulting semantic entities from this sentence (normalizing letters to lower case):

[general motors] {builds} (their) [chevrolet volt] {in} the [detroit-hamtramck assembly plant]

From the initial sentence iKnow has identified the following entities:

- 3 Concepts: `[general motors] [chevrolet volt] [detroit-hamtramck assembly plant]`.

- 2 Relations: `{builds} {in}`.

- 1 Path-relevant: `(their)`. (Path-relevents are considered in path analysis, but are not indexed.)

- 1 Non-relevant: `the`. (Non-relevants are discarded from further iKnow analysis.)

Note: For the purpose of illustration, this example shows each Concept delimited by square brackets, each Relation delimited by curly braces, and each path-relevant delimited by parentheses; within iKnow such delimiter characters are not used.

iKnow assigns each entity a unique Id. iKnow also identifies sequences of these entities that follow the pattern Concept-Relation-Concept (CRC). In this example there are two CRCs:

[general motors] {builds} [chevrolet volt]
[chevrolet volt] {in} (their) [detroit-hamtramck assembly plant]

iKnow assigns each CRC a unique Id.

iKnow also recognizes that this sentence contains a continuous sequence of entities (in this case, the sequence CRCRC). A sequence of entities within a sentence that together express a single statement is known as a Path. The entire sentence may be a single Path, or may contain multiple Paths. iKnow assigns each Path a unique Id.

iKnow has now identified the sentences in the text, the relevant Entities in each sentence, which Entities are Concepts and which are Relations, which of these form CRC sequences, and which sequences of entities form a Path. By using these semantic units, iKnow can return many types of meaningful information about the contents of the source texts.

1.2 What is iKnow?

iKnow provides access to unstructured data by dividing up text into relational and associated entities and producing an index of these entities. It divides a text into sentences, then divides each sentence into a sequence of Concepts and Relations. It performs this operation by identifying the language of the text (for example, English), then applying the corresponding iKnow language model.

- A Relation is a word or group of words that join two Concepts by specifying a relationship between them. iKnow contains a compact language model that is able to identify the Relations in a sentence.

- A Concept is a word or group of words that is associated by a Relation. By determining what is a Relation, iKnow can identify associated Concepts. Thus the iKnow analysis engine can identify Concepts semantically without "understanding" their content.

Note: For the purpose of explanation, verbs are commonly Relations and nouns with their associated adjectives are commonly Concepts. However, the linguistic model of Relations and Concepts is significantly more inclusive and more sophisticated than the distinction between verbs and nouns.

Thus iKnow divides a sentence into Concepts (C) and Relations (R). The language model uses a relatively small and fixed dictionary of relationship words and a set of context rules to identify Relations. Anything not identified as a Relation is considered a Concept. (iKnow also identifies non-relevant words, such as "the" and "a", and discards them from further analysis.)

Relations and Concepts are collectively known as Entities. However, a Relation is almost never meaningful without an associated Concept. For this reason, iKnow entity analysis emphasizes Concepts and sequences that contain Concepts associated by a Relation.

Because iKnow analyzes text using a small and stable language model focused on identifying Relations, iKnow can rapidly index texts containing any subject matter. iKnow does not need to use a dictionary or ontology to identify Concepts.

Once iKnow has identified the Concepts and Relations in each sentence in a text, or (more commonly) in many texts, this information can be used to perform the following types of operations:

- Smart Indexing: provides insight into what's relevant, what's related, and what's representative from a large body of unstructured text.

- Smart Matching: provides a means to associate entities in the source texts with external items such as lists or dictionaries. These lists can contain words, phrases, or sentences for full (identical) matching and partial matching, and can contain templates for matching by format.

1.2.1 What iKnow Isn't

iKnow is not a search tool. Search tools enable you to locate only those things that you already believe are in the text. InterSystems provides the iFind tool as a search tool for unstructured text data in SQL tables. iFind uses many of the features of iKnow to provide intelligent text search.

iKnow is a content analysis tool. iKnow enables you to use the entire contents of the text data, including texts whose content is wholly unknown to you.

iKnow is not a dictionary-based content analyzer. Unlike dictionary-based tools, it does not break up sentences into individual words then attempt to "understand" those words and reconstruct context. iKnow simply identifies Entities semantically. It does not need to look up these Entities in a dictionary or ontology. For this reason its language model is compact, stable, and general-purpose; you do not have to specify any information about the type of texts being analyzed (medical, legal, etc.), or provide a separate dictionary of relevant terms. While iKnow can be extended by associating a dictionary or ontology of terms, its essential functions do not require one. Thus it does not require the creation, customizing, or periodic updating of a dictionary.

iKnow supports stemming, but is not by default a stemming tool. By default it does not reduce relations or concepts to stem forms. Instead, it treats each element as a distinct entity, then identifies its degree of similarity to other elements. iKnow supports stemming as an optional feature; it is recommended primarily for use with Russian and Ukrainian text sources. Caché also provides a set of classes you can use to perform stemming: the %Text package, as described in the *InterSystems Class Reference*. %Text and %iKnow are wholly independent of each other and are used for different purposes.

1.3 Logical Text Units Identified by iKnow

1.3.1 Sentences

iKnow uses a language model to divide the source text into sentences. In general, iKnow defines a sentence as a unit of text ending with a sentence terminator (usually a punctuation mark) followed by at least one space or line return. The next sentence begins with the next non-whitespace character. Capitalization is not required to indicate the beginning of a sentence.

Sentence terminators are (for most languages) the period (.), question mark (?), exclamation mark (!), and semi-colon (;). A sentence can be terminated by more than one terminator, such as ellipsis (...) or emphatic punctuation (??? or !!!). Any combination of terminators is permitted (...!?). A blank space between sentence terminators indicates a new sentence; therefore, an ellipsis containing spaces (. . .) is actually three sentences. A sentence terminator must be followed by either a whitespace character (space, tab, or line return), or by a single quote or double quote character, followed by a whitespace character. For example, `"Why?" he asked.` is two sentences, but `"Why?", he asked.` is a single sentence.

A double line return acts as a sentence terminator with or without a sentence terminator character. Therefore, a title or a section heading is considered to be a separate sentence, if followed by a blank line. The end of the file is also treated as a sentence terminator. Therefore, if a source contains any content at all (other than whitespace) it contains at least one sentence, regardless of the presence of a sentence terminator. Similarly, the last text in a file is treated as a separate sentence, regardless of the presence of a sentence terminator.

A period followed by a blank space usually indicates a sentence break, though iKnow language models recognize exceptions to this rule. For example, the English language model recognizes common abbreviations, such as "Dr." and "Mr." (not case-sensitive) and removes the period rather than performing a sentence break. The English language model recognizes "No." as an abbreviation, but treats lowercase "no." as a sentence terminator.

You can use the UserDictionary option of your Configuration to cause or avoid sentence endings in specific cases. For example, the abbreviation "Fr." (Father or Friar) is not recognized by the English language model. It is treated as a sentence

break. You can use a UserDictionary to either remove the period or to specify that this use of a period should not cause a sentence break. A UserDictionary is applied as a source is loaded; already loaded sources are not affected.

1.3.2 Entities

An entity is a minimal logical unit of text. It is either a word or a group of words that iKnow logically groups together into either a concept or a relation. Other logical units, such as a telephone number or an email address, are also considered entities (and are treated as concepts).

Note: Japanese text cannot be divided into concepts and relations. Instead iKnow analyzes Japanese text as a sequence of entities with associated particles. The definition of an "entity" for Japanese is roughly equivalent to a Concept in other iKnow languages. For a description of iKnow Japanese support (written in Japanese) refer to iKnow Japanese.

iKnow normalizes entities so that they may be compared and counted. It removes non-relevant words. It translates entities into lower case letters. It removes most punctuation and some special characters from entities.

By default, iKnow restricts its analysis of entities to Concepts. By default, Relations are only analyzed because of their role in linking Concepts together. This default can be overridden, as described in the "Limiting by Position" section of the iKnow Queries chapter.

1.3.2.1 Path-relevant Words

iKnow identifies certain words in each language as being an essential part of its analysis of sentences and paths, but otherwise not relevant. Outside of the context of a sentence or path, these words have little informational content. The following are typical path-relevant words:

- Pronouns of all types: definite, indefinite, possessive.

- Indefinite expressions of time, frequency, or place. For example, "then", "soon", "later", "sometimes", "all", "here".

A word is only considered a path-relevant if it is not part of a Concept or a Relation. For example: "He said this was his" contains path-relevants; "His teacher said this signature was his name" does not contain path-relevants.

Path-relevant words are not considered Concepts, nor are they counted in frequency or dominance calculations. Path-relevant words may be negation or time attribute markers. Path-relevant words are not stemmed.

1.3.2.2 Non-relevant Words

iKnow identifies certain words in each language as being non-relevant, and excludes these words from iKnow indexing. There are several kinds of non-relevant words:

- Articles (such as "the" and "a") and other words that the iKnow language model identifies as having little or no semantic importance.

- Prefatory words or phrases at the beginning of a sentence, such as "And", "Nevertheless", "However", "On the other hand".

- Character strings over 150 characters that are unbroken by spaces or sentence punctuation. A "word" of this length is highly likely to be a non-text entity, and is thus excluded from iKnow indexing. Because in rare cases (such as chemical nomenclature or URL strings) these 150+ character words are semantically relevant, iKnow flags them with the attribute "nonsemantic".

Non-relevant words are excluded from iKnow indexing, but are preserved when sentences are displayed.

1.3.3 CRCs and CCs

Once iKnow divides a sentence into Concepts (C) and Relations (R), it can determine several types of connections between these fundamental entities.

- CRC is a Concept-Relation-Concept sequence. A CRC is handled as a Master Concept - Relation - Slave Concept sequence. Whether an entity is a Master, Relation, or Slave is known as its *position*. In some cases, a CRC may have an empty string value for one of the sequence members (CR or RC); this can occur, for example, when the Relation of the CRC is an intransitive verb: "Robert slept."

- CC is a Concept + Concept pair. iKnow retains the position of each Concept, but ignores the Relation between the two Concepts. A CC can either be handled as two associated Concepts, or as a Master Concept/Slave Concept sequence. You can use CC pairs to identify associated Concepts without regard to their master/slave positions or the linking Relation. This is especially useful when determining a network of Concepts — what Concepts have a connection to what other Concepts. You can also use CC pairs as a master/slave sequence.

Note: Japanese cannot be analyzed semantically in terms of CRCs or CCs because iKnow does not divide Japanese entities into concepts and relations.

1.3.4 Paths

A Path is a meaningful sequence of Entities through a sentence. In Western languages, Paths are commonly based on sequential CRCs, thus resulting Paths have the entities (Concepts & Relations) in their original sentence order. Commonly, though not exclusively, this takes the form of a continuous sequence of CRCs. For example, in a common path sequence the Slave Concept of one CRC becomes the Master Concept of the next CRC. This results in a path consisting of five entities: C-R-C-R-C. Other meaningful sequences of Concepts and Relations are also treated as paths, such as a sequence that contains a path-relevant pronoun as a stand-in for a Concept.

In Japanese, Paths cannot be based on the sequence of Entities in the original sentence. iKnow nevertheless does identify Paths as meaningful sequences of Entities within Japanese text. iKnow semantic analysis of Japanese uses an entity vector algorithm to create Entity Vectors. When iKnow converts a Japanese sentence into an Entity Vector it commonly lists the Entities in a different order than the original sentence to indicate which Entities are linked to each other and how strong the link between them is. The resulting Entity Vector is used for Path analysis.

A Path must contain at least two Entities. Not all sentences are paths; some very short sentences may not contain the minimum number of Entities to qualify as a path.

A path is always contained within a single sentence. However, a sentence may contain more than one path. This can occur when iKnow identifies a non-continuous sequence within the sentence. Once identified, the entities that comprise a path sequence are demarcated and normalized, and the path is assigned a unique Id. Paths are useful when an analysis of just CRCs is not large enough to identify some meaningfully associated entities. Paths are especially useful when returning some smaller linguistic unit in a wider context.

1.4 Smart Indexing

Smart indexing is the process of translating unstructured text into a relational network of Concepts and Relations. You can index the contents of multiple unstructured texts, then analyze the resulting indexed entities according to user-defined query criteria, such as listing concepts in order of frequency. Each indexed entity can reference its source text, source sentence, and relational entities, such as its position in a CRC sequence. As part of smart indexing, iKnow assigns two values to each indexed concept, specifying the total number of appearances of the concept in the texts (its frequency), and the number of different texts in which the concept appears (its spread).

Once you have performed smart indexing on multiple texts, iKnow can use this information to analyze the source texts. For example, iKnow can perform intelligent content browsing. From any selected iKnow indexed item, you can browse to other items based on the degree of similarity between these items. Intelligent browsing can be performed within a source text or across all indexed source texts.

Once texts are indexed, iKnow can generate summaries of individual texts. The user specifies the length of the summary as a percentage of the original text. iKnow returns a summary text consisting of those sentences of the original text that are most relevant to the whole, based on index statistics. For example, if a text consists of 100 sentences, and the user specifies a 50% summary, iKnow generates a summary text consisting of the 50 most relevant sentences from the original.

1.5 Smart Matching

Once iKnow has indexed a collection of texts, it is possible to match items found in the texts with one or more user-defined match lists and to tag these matches. Smart matching performs high-precision tagging of concepts and phrases based on a semantic understanding of the complete context. Thus matches can occur between similar concepts or phrases, as well as full (identical) matches. Because this tagging is based on finding semantic matches, smart matching does not require any understanding of the text contents.

Once tagged, each appearance of a matched phrase in the texts remains associated with the tag text. These phrases can be matched as a single entity, a CRC, or a path. For example, the user could supply a list of the names of countries, so that each appearance of a country name in the texts is tagged for rapid access. You can build a dictionary of company names that you can match against analyst reports, allowing you to quickly find the latest news about the companies you're interested in. You can create a dictionary in which each appearance of specified medical procedure (phrased in various ways) is matched to a medical diagnostic code.

This dictionary matching is not limited to simple entities, but extends to CRCs and/or paths if the terms in the dictionary span more than one entity themselves. Because iKnow indexes dictionary terms in the same way that it indexes source texts, a dictionary entry may be as long as a sentence. It may be useful to match a dictionary entry sentence against sources to locate similar information.

2
iKnow Implementation

The iKnow semantic analysis engine is a fully-integrated component of Caché. No separate installation is required. No configuration changes are needed.

Your ability to use iKnow is governed by the Caché license. Most standard Caché licenses provide access to iKnow. All Caché licenses that support iKnow provide full and unlimited use of all components of iKnow.

iKnow is provided as a collection of APIs containing class object methods and properties which may be invoked from ObjectScript programs. APIs are provided to invoke iKnow operations from Caché (API classes). Equivalent APIs are provided to invoke iKnow operations from SQL (QAPI classes) and SOAP web services (WSAPI classes). These APIs are described in the %iKnow package in the *InterSystems Class Reference*. iKnow is a core Caché technology and therefore does not have application-like interfaces. However, iKnow does provide a few generic, sample output interfaces in the %iKnow.UI package.

To use iKnow you must define an iKnow domain within a Caché namespace. You can create multiple iKnow domains; a Caché namespace can contain multiple iKnow domains. All iKnow processing occurs within a specified domain. A set of iKnow indexed text sources is created within a domain. All iKnow queries and other text processing must specify the domain in which to access this data.

2.1 A Note on Program Examples

Many of the program examples in this manual begin by deleting a domain (or its data) and then loading all data from the original text files into an empty domain. For the purpose of these examples, this guarantees that the iKnow indexed source data is an exact match with the contents of the file(s) or SQL table(s) from which it was loaded.

This delete/reload methodology is not recommended for real-world applications processing large numbers of text sources. Instead, you should perform the time-consuming load of all sources for a domain once. You can then add or delete individual sources to keep the indexed source data current with the contents of the original text files or SQL tables.

2.2 A Note on %Persistent Object Methods

iKnow support standard %Persistent object methods for creating and deleting object instances such as domains, configurations, and so forth. These %Persistent method names begin with a % character, such as **%New()**. Use of %Persistent object methods is preferable to using older non-persistent methods, such as **Create()**. Users are encouraged to use the %Persistent object methods for new code. Program examples throughout this documentation have been revised to use these preferred %Persistent methods.

Note that the **%New()** persistent method requires a **%Save()** method. The older **Create()** method does not require a separate save operation.

2.3 A Note on %iKnow and %SYSTEM.iKnow

Throughout this documentation, all classes referred to are located in the %iKnow package. However, the %SYSTEM.iKnow class also contains a number of iKnow utilities that can be used to simplify coding of common iKnow operations. These utilities are provided as shortcuts; all of the operations performed by %SYSTEM.iKnow class methods can also be performed by %iKnow package APIs.

You can display information about %SYSTEM.iKnow class methods by using the **Help()** method. To display information about all %SYSTEM.iKnow methods, invoke %SYSTEM.iKnow.Help(""); to display information about a specific %SYSTEM.iKnow method, supply the method name to the **Help()** method, as shown in the following example:

ObjectScript

```
DO ##class(%SYSTEM.iKnow).Help("IndexDirectory")
```

For further details refer to the *InterSystems Class Reference*.

2.4 Space Requirements and iKnow Globals

iKnow globals in a namespace have the following prefix: ^ISC.IK:

- ^ISC.IK.* are the final globals, permanent globals that contain iKnow data. This iKnow data is roughly 20 times the size of the original source texts.

- ^ISC.IKS.* are the staging globals. During data loading these can grow to 16 times the size of the original source texts. Staging globals should be mapped to a non-journaled database. iKnow automatically deletes these staging globals once source loading and processing is completed.

- ^ISC.IKT.* are the temp globals. During data loading these can grow to 4 times the size of the original source texts. Temp globals should be mapped to a non-journaled database. iKnow automatically deletes these temp globals once source loading and processing is completed.

- ^ISC.IKL.* are logging globals. These are optional and their size is negligible.

CAUTION: These globals are for internal use only. Under no circumstances should iKnow users attempt to directly interact with iKnow globals.

For example, if you are loading 30Gb of source documents, you will need 600Gb of permanent iKnow data storage. During data loading you will need 1.17Tb of available space, 600Gb of which will be automatically released once iKnow indexing completes.

In addition, the cachetemp subdirectory in the Mgr directory may grow to 4 times the size of the original source texts for the duration of file loading and indexing.

These space requirements apply when you create a domain in Caché 2012.2 and load iKnow data, or when you upgrade a domain created in Caché 2012.1 (iKnow version 1) to support Caché 2012.2 (iKnow version 2) features. They do not apply to existing domains created and loaded with iKnow data in Caché 2012.1, or to any data added to a 2012.1 domain in 2012.2. Caché 2012.1 domains have smaller space requirements (and support fewer features), as described in Upgrading iKnow Data in the "iKnow Tools" chapter.

Caché 2013.1 (iKnow version 3) data space requirements are not significantly larger than those for Caché 2012.2.

You should increase the size of the Caché global buffer, based on the size of the original source texts. Refer to the "Performance Considerations when Loading Texts" chapter in this manual.

2.4.1 Batch Load Space Allocation

Caché allocates 256MB of additional memory for each iKnow job to handle batch loading of source texts. By default, iKnow allocates one job for each processor core on your system. The $$$IKPJOBS domain parameter establishes the number of iKnow jobs; generally the default setting gives optimal results. However, it is recommended that the maximum number of iKnow jobs should be either 16 or the number of processor cores, whichever is smaller.

2.5 Input Data

iKnow is used to analyze unstructured data. Commonly, this data consists of multiple text sources, often a large number of texts. A text source can be of any type, including the following:

* A file on disk that contain unstructured text data. For example, a txt file.

* A record in an SQL result set with one or more fields that contain unstructured text data.

* An RSS web feed containing unstructured text data.

* A Caché global containing unstructured text data.

iKnow does not modify the original text sources, nor does it create a copy of these text sources. Instead, iKnow stores its analysis of the original text source as normalized and indexed items, assigning an Id to each item that permits iKnow to reference its source. Separate Ids are assigned to items at each level: source, sentence, path, CRC, and entity.

iKnow supports texts in the following languages: Dutch (nl), English (en), French (fr), German (de), Japanese (ja), Portuguese (pt), Russian (ru), Spanish (es), Swedish (sv), and Ukrainian (uk). You do not have to specify what language your texts contain, nor must all of your texts or all of the sentences in an individual text be in the same language. iKnow can automatically identify the language of each sentence of each text and applies the appropriate language model to that sentence. You can define an iKnow configuration that specifies the language(s) that your texts contain, and whether or not to perform automatic language identification. Use of an iKnow configuration can significantly improve iKnow performance.

You do not have to specify a genera for text content (such as medical notes or newspaper articles); iKnow automatically handles texts of any content type.

2.5.1 File Formats

iKnow accepts source files of any format and with any extension (suffix). By default, iKnow assumes that a source text file consists of unformatted text (for example, a .txt file). It will process source files with other formatting (for example, .rtf, .doc) but may treat some formatting elements as text. To avoid this, you can either convert your source files to .txt files and load these .txt files, or you can create an iKnow converter to remove formatting from source text during iKnow loading.

You specify the list of file extensions as a Lister parameter. Only files with these extensions will be loaded. For example, this list of file extensions can be specified as an **AddListToBatch()** method parameter.

2.5.2 SQL Record Format

iKnow accepts records from an SQL result set as sources. iKnow generates a unique integer value for each record as the iKnow source Id. iKnow allows you to specify an SQL field containing unique values which iKnow uses to construct the

external Id for the source records. Note that the iKnow source Id is assigned by iKnow, it *is not* the external Id, though frequently both are unique integers. Commonly, the iKnow source text is taken from only some of the fields of the result set record, often from a single field containing unstructured text data. It can ignore the other fields in the record, or use their values as metadata to filter (include or exclude) or to annotate the source.

2.5.3 Text Normalization

iKnow maintains links to the original source text. This enables it to return a sentence with its original capitalization, punctuation, and so forth. Within iKnow, normalization operations are performed on entities to facilitate matching:

- Capitalization is ignored. iKnow matching is not case-sensitive. Entity values are returned in all lowercase letters.

- Extra spaces are ignored. iKnow treats all words as being separated by a single space.

- Multiple periods (...) are reduced to a single period, which iKnow treats as a sentence termination character.

- Most punctuation is used by the language model to identify sentences, concepts and relations, then discarded from further analysis. Punctuation is generally not preserved within entities. Most punctuation is only preserved in an entity when there are no spaces before or after the punctuation mark. However, the slash (/), and at sign (@) are preserved in an entity with or without surrounding spaces.

- Certain language-specific letter characters are normalized. For example, the German *eszett* ("ß") character is normalized as "ss".

The iKnow engine automatically performs text normalization when a source text is indexed. iKnow also automatically performs text normalization of dictionary terms and items.

You can also perform iKnow text normalization on a string, independent of any iKnow data loading, by using the **Normalize()** or **NormalizeWithParams()** methods. This is shown in the following example:

ObjectScript

```
SET mystring="Stately plump Buck Mulligan   ascended the StairHead,  bearing a shaving bowl"
SET normstring=##class(%iKnow.Configuration).NormalizeWithParams(mystring)
WRITE normstring
```

2.5.4 User-defined Source Normalization

The user can define several types of tools for source normalization:

- Converters process source text to remove formatting tags and other non-text content during loading.

- UserDictionary enables the user to specify how to rewrite or use specific input text content elements during loading. For example, UserDictionary can specify substitutions for known abbreviations and acronyms. It is commonly used to standardize text by eliminating variants and synonyms. It can also be used to specify text-specific exceptions to standard iKnow processing of punctuation.

2.6 Output Structures

iKnow creates global structures to store the results of its operations. These global structures are intended for use by iKnow class APIs only. They are not user-visible and should not be modified by the user.

iKnow indexed data is stored as Caché list structures. Each iKnow list structure contains a generated ID for that item, a unique integer value. iKnow entities can be accessed either by value or by integer ID.

iKnow preserves the relationships amongst indexed entities, so that each entity can reference the entities related to it, the path of that sequence of entities, the original sentence that contains that path, and the location of that sentence within its source text. The original source text is always available for access from iKnow. iKnow operations do not ever change the original source text.

2.7 Constants

iKnow defines constant values in the %IKPublic.inc file. After specifying this include file, you can invoke these constants using the $$$ macro invocation, as shown in the following example:

ObjectScript

```
#include %IKPublic
  WRITE "The $$$FILTERONLY constant=",$$$FILTERONLY
```

These constants include domain parameter names, query parameter values, and other constants.

2.8 Error Codes

The General Error Codes 8000-8099 are reserved for use by iKnow. For further details, refer to General Error Messages in the *Caché Error Reference*.

3

iKnow Architect

Caché provides the iKnow Architect as an interactive interface for creating and populating iKnow domains and performing analysis on the indexed data. iKnow Architect is accessed using the Caché Management Portal.

It consists of three tools:

- Architect: for creating an iKnow domain and populating it with source text data.

- Knowledge Portal: for analyzing the data in an iKnow domain by looking at specific entities.

- Indexing Results: for displaying how iKnow analyzed the text data in a source, using highlighting to show different types of entities.

All functionality provided through the iKnow Architect is also available by using ObjectScript to invoke iKnow class methods and properties.

3.1 Accessing iKnow Architect

The starting point for accessing the iKnow Architect is the Management Portal **System Explorer** option. From there you select the **iKnow** option.

All iKnow domains exist within a specific namespace. Therefore, you must specify which namespace you wish to use by selecting the **Switch** option at the top of any Management Portal interface page. This displays the list of available namespaces, from which you can make your selection.

A namespace must be iKnow-enabled before it can be used. Selecting an iKnow-enabled namespace displays the iKnow **Domain Architect** option.

Note: If selecting an iKnow-enabled namespace does not display the **Domain Architect** option, you do not have a valid iKnow license. Look at **Licensed to** in the Management Portal header. Review or activate your license key.

3.1.1 Enabling a Namespace

A namespace must be iKnow-enabled before it can be used with iKnow Architect.

- If no namespaces are iKnow-enabled, the **iKnow** option displays the greyed out (disabled) message "No iKnow-enabled namespaces found".

- If the current namespace is not iKnow-enabled, the **iKnow** option displays a list of iKnow-enabled namespaces. You can select one of these displayed namespaces, and then select it from the Namespace Chooser window.

To enable a namespace for iKnow from the Management Portal, select **System Administration**, **Security**, **Applications**, **Web Applications** (**[System] > [Security Management] > [Web Applications]**). This displays a list of web applications; the third column indicates if a listed item is a namespace ("Yes") or not. Select the desired namespace name from the list. This display the **Edit Web Application** page. Select the **Enabled** check box for **iKnow**. Click the **Save** button.

You cannot enable the %SYS namespace. This is because you cannot create iKnow domains in the %SYS namespace.

You can set your Management Portal default namespace. From the Management Portal select **System Administration**, **Security**, **Users** (**[System] > [Security Management] > [Users]**). Select the name of the desired user. This allows you to edit the user definition. From the General tab, select a **Startup Namespace** from the drop-down list. Click **Save**.

3.2 Creating a Domain

From the iKnow Architect press the **New** button to define a domain. You specify the following domain values (in the specified order):

- **Domain name**: The name you assign to a domain must be unique for the current namespace (not just unique within its package class). A domain name may be of any length and contain any typeable characters, including spaces (the % character is valid, but should be avoided). Domain names are not case-sensitive. However, because iKnow Architect uses the domain name to generate a default domain definition class name, it is recommended that you follow class naming conventions when naming a domain, unless there are compelling reasons to do otherwise.

- **Definition class name**: the domain definition package name and class name, separated by a period. If you first specified the domain name, clicking on the **Definition class name** generates default names for the domain definition package and class. The package name defaults to User. The class name defaults to the domain name, stripped of non-alphanumeric characters. You can accept or modify this default.

  The package name and the class name can contain only alphanumeric characters, and are case-sensitive. Specifying a package name that differs from an existing package name only in lettercase results in an error. Within a package, specifying a class name that differs from an existing class name only in lettercase results in an error.

- **Allow Custom Updates**: optionally select this box if you wish to enable adding data or dictionaries to this domain manually; the default is to not allow custom updates.

Click the **Finish** button to create the domain. This displays the Model Elements selection screen.

You must **Save** and **Compile** a newly created domain before exiting that domain.

Both domain names and domain definitions must be unique within the namespace. If a duplication occurs, iKnow Architect performs the following operations:

Duplicate domain: if you create two domain names in the namespace that are the same but have different domain definitions, the iKnow Architect will appear to create both domains. However, when attempting to compile the domains, compilation will fail for the second domain name.

Duplicate domain definitions: if you create two domains that have the different names but have the same domain definition, the iKnow Architect will overwrite the first domain with the second domain. This is a delete and replace operation, not a rename. iKnow Architect issues no message when performing this overwrite.

For other ways to create a domain, refer to iKnow Domains. Note that iKnow Architect is the only domain creation interface that allows you to define a domain definition package name and class name.

3.2.1 Opening a Domain

Creating a domain using the Management Portal interface immediately opens the domain, allowing you to begin immediately to manage this new domain.

To manage an existing domain, click the **Open** button to list all existing domains in the namespace. This display lists the packages that contain domains. Select a package to display its domains. Select an existing domain. This displays the Model Elements selection screen.

3.2.2 Changing the Domain Name and Check Boxes

Creating or opening a domain displays the Model Elements window. If you click on the domain name in this window, the Details tab displays the **Domain Name** field, the **Domain Tables Package** field, and the **Allow Custom Updates** and **Disabled** check boxes. You can modify these characteristics of the domain. Changing the **Domain Name** does not change the **Definition class name**.

Checking the **Allow Custom Updates** check box allows the manual loading of data sources and dictionaries into this domain using interfaces other than iKnow Architect.

Checking the **Disabled** check box prevents the loading of all data (source data, metadata, dictionary matching data) during the Build operation. Each of these types of data also has its own **Disabled** check box that allows you to disable loading of each types of data separately.

You must **Save** and **Compile** a renamed domain before exiting that domain.

3.2.3 Deleting a Domain

To delete the current domain, click the **Delete** button. This displays the Drop domain data window. you can either delete just the domain contents or delete the domain definition. Click **Drop domain & definition class** to delete the domain and its associated class definition, including the specifications of data sources, blacklists, and other model elements.

3.3 Model Elements

After creating a domain, or opening an existing domain, you can define model elements for the domain. To add or modify model elements, click on the expansion triangle next to one of the headings. Initially, no expansion occurs. Once you have defined some model elements, clicking the expansion triangle shows the model elements you have defined.

To add a model element, click the heading. Then click the Add button shown in the Details tab on the right side. Specify the name and values. The model element is automatically generated when you leave the Details area. Model elements are listed in the order of their creation, with the most-recently-created element at the top of the list; modifying a model element does not change its position in the list.

To modify a model element, expand the heading, then click a defined model element. The current values are shown in the Details tab on the right side. Modify the name and/or values as desired. The model element is automatically re-generated when you leave the Details area.

Once you have created model elements, clicking on the **Expand All** button (or one of the expansion triangles) displays these defined values. The Element Type column shows the type of each model element. Clicking on the red "X" deletes that model element.

The **Save** button saves all changes. The **Domain Architect** page heading is followed by an asterisk (*) if there are unsaved changes. Click **Save** to save your changes.

The **Undo** button reverses the most recent unsaved change. You can click **Undo** repeatedly to reverse unsaved changes in the reverse order that they were made. Once changes are saved, this button disappears.

The following Model Elements are provided:

- Domain Settings

- Metadata Fields

- Data Locations

- Blacklists

- Matching

3.3.1 Domain Settings

This model element allows you to modify the characteristics of the domain. All Domain Settings are optional and take default values. Domain Settings provides the following options:

- **Languages**: select one or more languages that you wish iKnow to identify in the text data. If you check more than one language, automatic language identification is activated. This increases the processing required for texts. Therefore, you should not select multiple languages unless there is a real likelihood that texts in the selected language will be part of the data set. The default language is English.

- **Add Parameter**: this button allows you to specify a domain parameter value. Specify the domain parameter name and the new value. Domain parameter names are case-sensitive. For example, Name=SortField, Value=1. No validation is performed. All unspecified domain parameters take their default values. To view the parameters that you have added, expand the Domain Settings heading.

- **Maximum Concept Length**: the largest number of words that should be indexed as a concept. This option is provided to prevent a long sequence of words from being indexed as a concept. The default (0) uses the language-specific default for the maximum number of words. This default should be used unless there are compelling reasons to modify it.

- **Manage User Dictionary**: this button displays a "Manage User Dictionary" box that allows you to specify one or more strings to the user dictionary. Each specified string either specifies a string that will rewrite to a new string, or specifies a string to which you assign an attribute label from a drop-down list.

3.3.2 Metadata Fields

Add Metadata: this button allows you to specify a source metadata field. For each metadata field you specify the field name, the data type (String, Number, or Date), the supported operators, and the storage type. After creating a domain, you can optionally specify one or more metadata fields that you can use as criteria for filtering sources. A metadata field is data associated with an iKnow data source that is not itself iKnow indexed data. For example, the date and time that a text source was loaded is a metadata field for that source. Metadata fields must be defined *before* loading text data sources into a domain.

Case Sensitive check box: By default, a metadata field is not case-sensitive; you can select this check box to make it case-sensitive.

Disabled check box: You can select the Disabled check box to disable all metadata fields, or you can select the Disabled check box displayed with an individual metadata field to disable just that metadata field. A disabled field is not loaded during the Build operation.

The metadata fields that you specify here appear in the Data Locations **Add data from table** and **Add data from query** details under the title "Metadata mappings".

3.3.3 Data Locations

Specifies the source for adding data. Option are **Add data from table**, **Add data from query**, **Add data from files**, **Add RSS data**, and **Add data from global**.

- The **Drop existing data before build** check box allows you to specify whether source text data already indexed in this iKnow domain should be deleted before adding the source text data specified here. To use this check box to drop data, data loading must not be disabled. To drop existing data without loading new data, use the **Delete** button **Drop domain contents only** option.

- The **Disabled** check box allows you to disable source indexing; disabled source data is not loaded during the Build operation. If data loading is disabled, the **Drop existing data before build** check box is ignored.

  A Build operation for a large number of texts may take some time. If you have already loaded the data locations and wish to add or modify metadata or a matching dictionary you can click the Data Locations **Disabled** check box to index these model elements without reloading the data locations.

After specifying data locations, you must **Save** and **Compile** the domain, then select the **Build** button to build the data indices.

3.3.3.1 Add Data from Table

This option allows you to specify data stored in an existing SQL table in the current namespace. It provides the following fields:

- Name: you can either specify a name or take the default name for the extracted result set table. Follows SQL table naming conventions. The default name is Table_1 (with the integer incrementing for each additional extracted result set table you define).

- Batch Mode: a check box indicating whether or not to load source text data in batch mode.

- Schema: from this drop-down list select an existing schema in the current namespace.

- Table Name: from this drop-down list select an existing table in the selected schema.

- ID Field: from this drop-down list select a field from the selected table to serve as the ID field (primary record identifier). An ID field must contain unique, non-null values.

  Selecting `-custom-` from the drop-down list allows you to input a field name; for example, a hidden RowId field or a field that does not (yet) exist. Field names are not case-sensitive. Selecting `-custom-` also displays the **Show Default Options** button. This button selects the first non-hidden field in the table from the drop-down list and also allows you to return to the drop-down list of fields.

- Group Field: an SQL select-item expression that retrieves a secondary record identifier from the selected table. This field defaults to the initial ID Field selection.

  Selecting `-custom-` from the drop-down list allows you to input a field name; for example, a hidden RowId field or a field that does not (yet) exist. Field names are not case-sensitive. Selecting `-custom-` also displays the **Show Default Options** button. This button selects the first non-hidden field in the table from the drop-down list and also allows you to return to the drop-down list of fields.

- Data Field: from this drop-down list select a field from the selected table to serve as the data field. The data field contains the text data loaded for iKnow indexing.

  Selecting `-custom-` from the drop-down list allows you to input a field name; for example, a hidden RowId field or a field that does not (yet) exist. Field names are not case-sensitive. Selecting `-custom-` also displays the **Show Default Options** button. This button selects the first non-hidden field in the table from the drop-down list and also allows you to return to the drop-down list of fields.

- Where Clause: you can optionally specify an SQL WHERE clause to limit which records are included in the result set table. Do not include the WHERE keyword.

If you have defined one or more Metadata Fields for this domain, the **Metadata mapping** option allows you to specify a metadata field for this table. From the drop-down list you can select a field from the selected table, select **– not mapped –**, or select **– custom –**. If you select **– custom –** the Architect displays an empty field in which you can specify the custom mapping.

If you have not defined any Metadata Fields for this domain, the **Metadata mapping** option provides a **Declare Metadata** button that directs you to the **Add Metadata** domain option.

3.3.3.2 Add Data from Query

Add data from query is similar to **Add data from table**, but allows you to specify a fully-formed SQL query for an existing table (or tables), from which you provides the following fields:

- Name: you can either specify a name or take the default name for the extracted result set table. Follows SQL table naming conventions. The default name is Query_1 (with the integer incrementing for each additional extracted result set table you define).

- Batch Mode: a check box indicating whether or not to load source text data in batch mode.

- SQL: the query text, a Caché SQL SELECT statement. Defining a query allows you to select fields from more than one table by using JOIN syntax. When specifying more than one table, assign column aliases to selected fields. Defining a query also allows you to specify an expression field that you can use as the Group field.

  The following field selection drop-down lists display the selected fields. They do not display table alias prefixes. If the field has a column alias, this alias is listed rather than the field name.

- ID Field: from this drop-down list select a field from the selected table to serve as the ID field. An ID field must contain unique, non-null values.

- Group Field: from this drop-down list select a select-item expression (such as an SQL function expression) from the query to serve as a secondary record identifier (group field). For example, `YEAR(EventDate)`.

- Data Field: from this drop-down list select a field from the selected table to serve as the data field. The data field contains the text data loaded for iKnow indexing.

If you have defined one or more Metadata Fields for this domain, the **Metadata mapping** option allows you to select either **– not mapped –** or **– custom –** for each defined metadata field. The default is **– not mapped –**. If you select **– custom –** the Architect displays an empty field in which you can specify the custom mapping.

If you have not defined any Metadata Fields for this domain, the **Metadata mapping** option provides a **Declare Metadata** button that directs you to the **Add Metadata** domain option.

The Model Elements window Element Type column displays a truncated form of the query you defined; the query is truncated after the first table name in the FROM clause. The full query is shown in the Details window.

3.3.3.3 Add Data from File

This option allows you to specify data stored in files. It provides the following fields:

- Name: you can either specify a name or take the default name for the extracted data file. The default name is File_1 (with the integer incrementing for each additional extracted data files you define).

- Path: the complete directory path to the directory containing the desired files. The Path syntax is filesystem dependent; on a Windows system it might look like the following: `C:\\temp\iKnowSources\`

- Extensions: the file extension, such as `txt` or `xml`. Do not include the dot prefix when specifying the file extension. Specify multiple extensions as a comma-separated list with no dots and no spaces; for example, `txt,xml`. If specified,

only files with the specified extensions are included in the resulting extracted data. If the Extensions field is left blank (the default) all files are included, regardless of their extensions.

- Filter Condition: a condition used to restrict which files are to included in the resulting extracted data.

- Recursive: a check box indicating whether to select files recursively. When checked, data can be extracted from the files in the specified directory and files in all of its subdirectories, and their sub-subdirectories, etc. When not checked, data can be extracted only from files in the specified directory. The default is non-recursive (check box not checked).

- Batch Mode: a check box indicating whether or not to load source text data in batch mode.

- Encoding: a drop-down list of the types of character set encoding to use to process the files.

3.3.3.4 Add RSS Data

This option allows you to specify data from an RSS stream feed. It provides the following fields:

- Name: you can either specify a name or take the default name for the extracted data. The default name is RSS_1 (with the integer incrementing for each additional RSS source you define).

- Batch Mode: a check box indicating whether or not to load source text data in batch mode.

- Server Name: the name of the host server on which the URL is found.

- URL: the navigation path within the server address to the actual RSS feed.

- Text Elements: a comma-separated list of text elements to load from the RSS feed. For example `title,description`. Leave blank for defaults.

3.3.3.5 Add Data from Global

This option allows you to specify data from a Caché global. It provides the following fields:

- Name: you can either specify a name or take the default name for the extracted data. The default name is Global_1 (with the integer incrementing for each additional global source you define).

- Batch Mode: a check box indicating whether or not to load source text data in batch mode.

- Global Reference: The global from which you wish to extract the source data.

- Begin Subscript: the first global subscript in a range of subscripts to include.

- End Subscript: the last global subscript in a range of subscripts to include.

- Filter Condition: a condition used to restrict which files are to included in the resulting extracted data.

3.3.4 Blacklists

Define blacklists: After creating a domain, you can optionally create one or more blacklists for that domain. A blacklist is a list of terms (words or phrases) that you do *not* want a query to return. Thus a blacklist allows you to perform iKnow operations that ignore specific terms in data sources loaded in the domain.

- Name: specify the name of a new blacklist, or take the default name. Blacklist names are not case-sensitive. Specifying a duplicate blacklist name results in a compile error. The default name is Blacklist_1 (with the integer incrementing for each additional blacklist you define).

- Entries: specify terms to include in the blacklist, one term per line. Terms should be in lower case. Duplicate terms are permitted. You can copy/paste terms from one blacklist to another. You can include blank lines to separate groups of terms. A line return at the end of your list of terms is optional; blank lines are not counted as entries.

If you add, modify, or delete a blacklist, you must **Save** and **Compile** the domain for this change to take effect.

Because defining blacklists has no effect on how data is loaded into a domain, changes to blacklists do not require re-building the domain.

Defining blacklists has no effect on how data is loaded into a domain. The blacklists defined here are compiled, then supplied to the Knowledge Portal, which allows you to specify none, one, or multiple blacklists when performing analysis of source text data loaded into the domain. A blacklist is applied to some (but not all) Knowledge Portal analytics.

3.3.5 Matching

The Matching option provides the **Add Dictionary** option to define a dictionary and specify its items and terms.

The Matching option provides four check box options, as follows:

- **Disabled**: You can select the Disabled check box to disable building of all dictionaries, or you can select the Disabled check box displayed with an individual dictionary to disable the building of that dictionary. Selecting **Disabled** check boxes allows you to build only those dictionaries that you have changed. The default is off.

- **DropBeforeBuild**: default on

- **AutoExecute**: default on

- **IgnoreDictionaryErrors**: default on

3.3.5.1 Add Dictionary

The **Add Dictionary** button displays the dictionary definition options: dictionary name (with a supplied default), an optional description, the dictionary language selected from a drop-down list of iKnow supported languages, and the disabled check box. The default name is Dictionary_1 (with the integer incrementing for each additional dictionary you define).

The **Add Item** button displays the item definition options: item name (with a supplied default), a uri name (with a supplied default), the item language selected from a drop-down list of iKnow supported languages, and the disabled check box. To define more items, select the dictionary name. Items are listed in order of creation, with the most recent at the top of the list. Within each item you can define one or more terms. The default name is Item_1, the default uri name is uri:1 (with the integer incrementing for each additional item you define for this dictionary).

The **Add Term** button displays the term definition options: a string specifying the term, the term language selected from a drop-down list of iKnow supported languages, and the disabled check box. To define more terms, select the item name. Terms are listed in order of creation, with the most recent at the top of the list.

3.4 Save, Compile, and Build

You must save, compile, and build a domain (in that order) using the buttons provided. You must save and compile a domain after adding, modifying, or deleting any Model Elements.

The **Save** button saves the current domain definition. Architect greys out (disables) the **Save** button if no domain definition is open. Architect does not issue an error if you save a domain definition without changing it.

The **Compile** button compiles the current domain definition. It compiles all of the classes and routines that comprise the domain definition. If you have not saved changes that you made to the domain definition, the compile operation prompts you to save the domain definition before compiling.

The **Build** button loads the specified sources into the current domain. If you have made changes to the Data Locations, Metadata Fields, or Matching dictionaries, you must build the domain. The Build Domain window displays progress messages such as the following:

```
13:50:48: Loading data...
13:51:49: Finished loading 3 sources
13:51:49: Creating dictionaries and profiles...
13:51:49: Finished creating 1 dictionaries, 1 items, 3 terms and 0 formats
13:51:49: Matching sources...
13:51:50: Finished matching sources
13:51:50: Successfully built domain 'mydomain'
```

The build operation can be time-consuming. If a **Disabled** check box is checked for a model element, the Build operation does not load the corresponding sources. Selecting **Disabled** check boxes allows you to build only those model elements that you have changed.

3.5 Knowledge Portal

The **Tools** tab provides the **Knowledge Portal** button. Once you have specified Data Locations and populated the domain with this data using the **Build** button, you can select **Knowledge Portal** to display iKnow analysis of the data. This displays the Knowledge Portal as a separate browser tab.

The Knowledge Portal is a Zen page query display interface with broad application. It shows a wealth of information about the source text data indexed in a domain. It initially displays a list of either the top (most-frequently-occurring) concepts, or the dominant (highest dominance) concepts. You can toggle between these two lists.

If you select an entity, the **Knowledge Portal** provides analysis of similar entities and related concepts, and analysis of the appearance of the specified entity in larger text units (sources, paths, and CRCs). This provides a contextual at-a-glance view of what's in your data.

The **Knowledge Portal** provides generic filters that support selecting subsets of the sources in a domain based on metadata criteria. This interface provides a sample of how iKnow Smart Indexing can be used to quickly overview and navigate a large set of documents.

3.5.1 Selecting a Domain

By default, the **Knowledge Portal** displays analysis of the domain that was current in iKnow Architect when you invoked the **Knowledge Portal**.

To select another domain:

1. Select the **Gear icon** at the upper right of the **Knowledge Portal**. This displays the **Settings** box.

2. The **Settings** box contains the **Switch domain** drop-down list. Select a domain from this list.

The number at the top right of the **Knowledge Portal** is the number of sources loaded in the selected domain that are available for data analysis. This number can be limited by applying filters.

3.5.2 Listing All Concepts

The Knowledge Portal initially provides concept analysis of the data sources loaded in the domain. There are two ways to list concepts, by frequency or by dominance. You can toggle between these two by selecting the **frequency** or **dominance** button:

- **Top Concepts**: selecting the **frequency** button lists all concepts in the sources in descending order of frequency. If multiple concepts have the same frequency, the concepts are listed in descending collation order. Each concept is listed with its frequency (total number of occurrences in all sources) and spread (number of sources containing that concept). To view frequency counts for a single source, use the Indexing Results tool.

- **Dominant Concepts**: selecting the **dominance** button lists all concepts in the sources in descending order of dominance score. If multiple concepts have the same dominance score, the concepts are listed in descending collation order. The dominance score is calculated by taking the dominance values for each source and using an averaging algorithm to determine the dominance of a concept across all loaded sources. Dominance values in a single source are integer values, with the most dominant concept given a dominance of 1000. To view dominance values for a single source, use the Indexing Results tool.

3.5.3 Analyzing a Specified Entity

There are two ways to display analysis of a specific entity:

- Select a concept from either the **Top Concepts** or **Dominant Concepts** listings.

- In the entry field in the top left corner you can type the first few characters (minimum of 2, not case-sensitive) of a word found in an entity, and the Knowledge Portal displays a drop-down list of all of the existing entities that contain a word beginning with those characters. Select an entity from this drop-down list, then press the **Explore!** button. You can use this option to display Relations or Concepts; both types of Entities are shown in the drop-down list.

Selecting an entity displays two kinds of analysis of that entity: associated entities and specified entity in context.

3.5.3.1 Associated Entities

Selecting an entity displays the following listings:

- **Similar Entities**: a list of concepts and relations that are similar to the specified entity, with the frequency (total number of occurrences in all sources) and spread (number of sources containing that concept) of each concept or relation. The first similar entity listed is always the specified entity itself. For a concept, this first listed entity is the same as the **Top Concepts** listing for that concept.

- **Related Concepts**: selecting the **related** button displays a list of concepts that are related to the specified concept, with the frequency (total number of occurrences in all sources) and spread (number of sources containing that concept) each concept. A related concept is a concept that appears in a CRC with the specified concept.

- **Proximity Profile**: selecting the **proximity** button displays the Proximity Profile table. This lists concepts associated by proximity to the specified concept, with a proximity score for each concept.

Selecting an entity from the **Similar Entities**, **Related Concepts**, or **Proximity Profile** listings changes all listings to analysis of that entity. It does not change the **Top Concepts** and **Dominant Concepts** listings.

3.5.3.2 Entity in Context

Selecting an entity also displays the following listings of that entity in context:

- **Sources**: a list of source texts containing the specified entity (shown highlighted in green), along with the internal source ID (an integer) and external source ID. Sources are listed in descending order by internal source ID. The source text displays all sentences in the source that contain the entity; intervening sentences that do not contain the entity are not displayed, but are indicated by ellipsis (. . .); note that leading ellipsis is not shown when the first displayed sentence is not the first sentence in the source, and trailing ellipsis is always shown after the final sentence, even when the last displayed sentence is actually the last sentence in the source.

  Red text indicates negation, with the entities within the scope of the negation attribute in red letters. Negation scope is not necessarily the same as the corresponding path, sentence, or CRC.

  Selecting the **Eye icon** or clicking anywhere in the listing for a source displays the full text of the source. Each occurrence of the specified entity is highlighted and each negation scope text is shown in red letters in the full text. (The % option must be set to 100% to display all occurrence of the specified entity in this full text box.)

Selecting the **Arrow icon** displays the Indexing Results tool.

- **Paths**: a list of paths containing the specified entity. Paths are listed in descending order by ID. Note that because path IDs are assigned on a per-source basis, the same path text may be listed multiple times with different path IDs.

   The elements of the path are highlighted by type:

   - Green: the specified entity (either a Concept or a Relation).

   - Blue: a Concept.

   - White: a Relation.

   - Light Blue: a Path-relevant Word.

   Negation scope text is displayed in red letters.

   Selecting a path element changes all listings to analysis of that entity. It does not change the **Top Concepts** and **Dominant Concepts** listings.

   Selecting the **Eye icon** displays the full text of the source with the specified entity highlighted in green.

   Selecting the **Arrow icon** displays the Indexing Results tool.

- **CRCs**: a list of Concept-Relation-Concept (CRC) sequences that contain the specified entity, with the frequency (total number of occurrences of that CRC in all sources) and spread (number of sources containing that CRC). Note that many CRCs contain only one concept: CR or RC. The entity type highlighting is the same as for **Paths**, except that Path-relevant Words are not part of CRCs and are therefore not displayed.

   Selecting a CRC element changes all listings to analysis of that entity. It does not change the **Top Concepts** and **Dominant Concepts** listings.

   Selecting the **Eye icon** displays the **Sources with selected CRCs** box, listing each source that contains an instance of the CRC. The CRC is highlighted in green in the context of its sentence, and flagged with the Source ID of the source. A source ID listing can contain multiple sentences containing the specified CRC; intervening sentences that do not contain the CRC are indicated by ellipsis. From the **Sources with selected CRCs** box you can select the **Eye icon** for a source containing the CRC to display the full text of the source with the specified entity (not the CRC) highlighted in green.

Note: If Japanese is the only language supported for the domain, the Knowledge Portal display differs as follows: the **Related Concepts** and **CRCs** listings are not shown. An **Entity Vectors** listing is substituted for the **Paths** listing.

3.5.3.3 Full Text Box

The **Eye icon** displays the full text of a selected source. This text box is identified by the external ID of the source. For example, `:SQL:1171:1171`.

The source text is tagged as follows:

- The specified entity is highlighted in green.

- Red text indicates negation, with the entities within the scope of the negation attribute in red letters.

This full text box provides the following option buttons:

- **metadata**: displays the metadata for the source. All sources are provided with a DateIndexed metadata field. This date stamp is represented as a UTC date and time in the Display format for your locale. It is truncated to whole seconds. To return to the source text, press the **metadata** button again.

- **highlight**: performs no action.

- **indexing**: displays the source text highlighted to indicate the types of entities, as follows:

- Green: the specified entity (either a Concept or a Relation).

- Blue: a Concept.

- White: a Relation.

- Light Blue: a Path-relevant Word.

- Unmarked: a Non-relevant word.

Negation scope text is displayed in red letters.

- **dictionaries**: performs no action.

- **%**: summarizes the source text. The default percentage is 100% (full text). Specifying a integer less than 100 and then pressing the **%** button summarizes the source text by reducing the text to (roughly) the specified size by eliminating sentences that are have a low relevancy score, when compared to the other sentences in the source. Summerization does not necessarily retain sentences that contain the specified entity.

3.5.4 Limiting the Sources to Analyze

You can limit the scope of your data analysis by using filters. A filter includes or excludes data sources that are loaded in the domain from analysis. By default, the **Knowledge Portal** analyzes all data sources loaded in the domain.

- The **Filter icon** (funnel) button at the top right of the Knowledge Portal applies a filter, which includes or excludes sources from analysis based on the criteria you specify. You can specify several types of filters, and can apply more than one filter. Multiple filters can be associated with AND, OR, NOT AND, or NOT OR logic.

  To add a filter, select the filter type from the drop-down list, specify the filter criteria, then select the **add** button, then the **Apply** button. When adding multiple filters, you select the AND/OR logic option associating the filters after the **add** button and before the **Apply** button.

  When one or more filters are in effect, the **Filter icon** displays in green.

  The number to the left of the **Filter icon** indicates the number of sources included after applying the filters. If no filters are applied, this number is the total number of sources in the domain.

- To remove a single filter, select the **Filter icon**, then select the black **X** next to the filter description, then select the **Apply** button. To remove all filters, select the **Filter icon**, then the **Clear** button, then the **Apply** button.

  The following filter types are supported:

  - **Metadata**: used to exclude sources by their metadata values. By default, all sources have DateIndexed metadata. To apply DateIndexed metadata, select this field, select an operator, and select a date value by clicking on the calendar icon, then selecting the desired day.

  - **Source IDs**: used to select sources for inclusion by source ID. You can specify a single source ID or a comma-separated list of source IDs.

  - **Source ID Range**: used to select sources for inclusion by source ID. You can range of source IDs by specifying the from and to range values. The range is inclusive of these values.

  - **External IDs**: used to select sources for inclusion by their external IDs. For example, `:SQL:1171:1171`. You can specify a single ID or a comma-separated list of IDs. External source IDs are listed in the **Sources** listing.

  - **SQL**: used to select sources for inclusion by specifying an SQL query.

3.6 Indexing Results

You can access the **Indexing Results** tool in two ways:

- From the Caché Management Portal **System Explorer iKnow** option. All iKnow domains exist within a specific namespace. Therefore, you must specify which namespace you wish to use from the list of available namespaces. A namespace must be iKnow-enabled before it can be used. Selecting an iKnow-enabled namespace displays the iKnow **Indexing Results** option.

- From the iKnow Knowledge Portal **Tools** tab **Indexing Results** button. Once you have specified Data Locations and populated the domain with this data using the **Build** button, you can select **Indexing Results** to display how iKnow has indexed the data. This displays the **Indexing Results** window as a separate browser tab.

At the top left of the **Indexing Results** window is a drop-down list that shows the sources loaded into the domain. (The domain is shown in the drop-down list at the top right.) Select a data source from the drop-down list, then press the **manual input** button.

This displays three listings: Indexed Sentences, Concepts, and CRCs

3.6.1 Indexed Sentences

The sentences in the source are listed in order, one sentence per line, with iKnow indexing indicated by highlighting. The sentence text is highlighted as follows:

- Yellow: a concept.

- Underlined: a relation.

- Italic: a non-relevant word.

- Red: a negation attribute phrase. The negation word is enclosed in a red box; multi-word negation terms (such as "was not") are shown with each word enclosed in a red box. The concepts and relations included in the negation phrase are shown with their appropriate highlighting (yellow highlighting or underlining), with the text of the phrase shown in red. Non-relevants within the phrase are not shown in red.

3.6.2 Concepts and CRCs

The Indexing Results displays two listings, one of all concepts in the source, one of all of the CRCs in the source

- **Concepts** in the source in descending order.

- **CRCs** in the source highlighted (as above) to indicate concepts and relations, in descending order. Note that the CRCs listings do not indicate negation attributes, and do not include non-relevant words.

The **sort by** buttons at the top of the window allow you to toggle the Concepts and CRCs listings to display either frequency counts or dominance values in descending order.

In the Concepts listing, the most dominant concept(s) are given a dominance of 1000. Less dominant concepts are given smaller integer values, with larger sources tending to have lower least-dominant values. For example, a source containing 25 concepts might have a dominance range between 1000 and 83; a source containing 300 concepts might have a dominance range between 1000 and 2.

Note: If Japanese is the only language supported for the domain, the Indexing Results display substitutes a single **Entities** listing for the **Concepts** and **CRCs** listings.

4

REST Interface

A REST API interface is provided for common iKnow tasks in the %iKnow.REST.v1 class. This REST API covers basics (querying domains & configurations), classic queries (including semantics), simple domain operations (adding/deleting sources, blacklist management) and support for the matching and dictionary APIs.

Here are some of the main characteristics of the API:

- Most API calls have a GET and POST variant. The GET variant takes default values for all non-essential parameters and should be easiest to invoke. The POST variant gives you access to all relevant arguments through a JSON string passed as the request object, which will complement what you put in the URL and override it in the event of an overlap. For example, some entity values that contain a slash cannot be used with a GET request, but can be supplied as a JSON property submitted in a POST request.

- Available arguments for POST requests are described succinctly in the Class Reference in %iKnow.REST.v1 and some shared arguments (filters & highlighting) in %iKnow.REST.Base.

- The entire URL is always in lowercase. POST arguments and returned JSON property names are in camelcase, with the first character in lowercase.

- Returned JSON is typically comprehensive, returning much more information than is supplied by the corresponding flat (SQL-compatible) query. For example, if you ask for all sources containing an entity, you'll get the list of sources, metadata, and optionally highlighted snippets back. You can exclude some of this functionality if desired for performance reasons, but by default most functionality is included automatically.

4.1 Swagger

The iKnow REST API is fully documented using the OpenAPI Specification (also known as Swagger). The description in YAML is available from the "/swagger" endpoint and can be loaded directly into swagger-ui for convenient GUI capabilities on top of this API.

To use it, either install swagger-ui or go to http://petstore.swagger.io and point to this endpoint.

- Download swagger-ui

- Unzip the swagger-ui download.

- In the Swagger box specify either:

  – for Caché with Minimal security: `http://localhost:57775/api/iknow/v1/samples/swagger` (substituting your Webserver port number for `57775` and the desired namespace for `samples`).

– for Caché with Normal security:
`http://localhost:57775/api/iknow/v1/samples/swagger?CacheUserName=myname&CachePassword=mypassword`
(substituting your Webserver port number for `57775`, the desired namespace for `samples`, your Caché user name for `myname`, and your Caché accounts password for `mypassword`).

• Select the Explore button.

4.1.1 Returning iKnow Data Using Swagger

The Swagger–UI display for each REST statement can be used to invoke the REST API directly, or to provide a Request URL value that can be used to invoke the REST API in a browser window (for GET requests).

To invoke a REST API directly, specify the required **Parameters**. Commonly, an iKnow REST API requires the domain ID parameter, specified as an integer number. For example, to GET the domain details of the domain with the ID of 2 generates the following Request URL:

`http://localhost:57775/api/iknow/v1/user/domain/2/details`

Note that the Swagger interface infers the namespace to use from the URL you used to access the /swagger endpoint. For example, to invoke the REST API in namespace SAMPLES, load the swagger endpoint using the URL:

`http://localhost:57775/api/iknow/v1/samples/swagger`

The iKnow data is returned to the browser in JSON format. For example, this Request URL returns the following JSON string for a domain named "crashes":

{"id":2,"name":"crashes","params":{"DefaultConfig":"crashesConfiguration","DefinitionClass":"crashes","ManagedBy":"crashes"},"meta":[{"id":1,"name":"DateIdx","ops":["=","!=","<","<=",">",">="],"dataType":"date","storage":"numeric","caseInsensitive":0,"hidden":0}]}

4.2 REST Operations

The iKnow REST operations are grouped into the following topics:

• Domain and Configurations information (listed in Swagger under "Miscellaneous")

• Sources within a specified domain.

• Entities within a specified domain.

• Sentences within a specified domain.

• Paths and CRCs within a specified domain.

• Dictionaries and Matching within a specified domain.

• Blacklists within a specified domain.

• Highlighting (listed in Swagger under "Miscellaneous")

While the core REST API supports access to any defined domain (by specifying the domain id at the start of almost every URL), you can also generate a domain-specific REST API, allowing you to expose a single domain through your webapp. To do this, use the **CreateDomainAPI()** method found in the %iKnow.REST.Base class, and then create a new web application referring to the newly created class as the "Dispatch Class". This may be useful if you only want to enable access to a specific domain and have disabled the general /api/iknow/ web application.

For any of these REST operations, if the specified domain does not exist, the operation returns
`{"errors":[{"error":"ERROR #8021: Domain with id 7 does not`

`exist","code":8021,"domain":"%ObjectErrors","id":"IKNoDomainWithId","params":["7"]}],"summary":"ERROR #8021: Domain with id 7 does not exist"}`, unless stated otherwise.

4.2.1 Domains and Configurations

The iKnow REST operations for Domains and Configurations (categorized as "miscellaneous") are the following:

- List Domains: GET and POST to retrieve information about all domains in the current namespace. See **GetDomains()**.

  If there are no defined domains, returns {"domains":[]}. Otherwise, returns a JSON array of domains in domain name order. For each domain, it lists the domain id, domain name, iKnow version, definitionClass name, and sourceCount.

  Listing domains is further described in the "iKnow Environment" chapter of this guide.

- List Configurations: GET and POST to retrieve information about all configurations in the current namespace. See **GetConfigurations()**.

  An iKnow-enabled namespace is assigned the configuration {"configurations":[{"id":1,"name":"DEFAULT","languages":["en"]}]}. You can explicitly define additional configurations. When you use iKnow Architect to create a domain, iKnow creates a corresponding configuration.

  This option returns a JSON array of configurations in configuration name order. For each configuration it lists the configuration id, the configuration name, a JSON array of supported languages, and (optionally) a userDictionary JSON object. The language defaults to English: ["en"].

  Listing configurations is further described in the "iKnow Environment" chapter of this guide.

- Domain Details: GET and POST to retrieve detailed information about a specified domain. See **GetDomainDetails()**.

  Data returned includes the domain's ID, name, parameters, and metadata.

  - parameters include DefaultConfig (domainname.Configuration), DefinitionClass, and ManagedBy (defaults to the DefinitionClass).

  - metadata includes metadata id, metadata name (defaults to DateIndexed), an array of supported operators, datatype (DateIndexed is datatype date), storage (defaults to normal), caseSensitive (a boolean value), and hidden (a boolean value).

4.2.2 Sources

The iKnow REST operations for sources are the following:

- Filtered Sources: GET and POST to retrieve all sources that pass a specified filter. By default, returns all sources. See **GetSources()**.

  If the domain exists but contains no sources that pass the filter, returns {"sources":[]}. If there are sources, returns the id, extId (external ID), metadata (the date and time when the source was indexed), and a snippet of the text for each source. This snippet consists of two sentences from the text. If the sentences are not consecutive, an ellipsis (three dots) is specified to indicate this gap. If there two or fewer sentences in the text, the snippet contains the full text.

  Filtering sources is further described in the "Filtering Sources" chapter of this guide.

- Sources by Entity: GET and POST to retrieve sources containing a specified entity. See **GetSourcesByEntity()**

  If the domain exists but contains no sources that pass the filter, returns {"sources":[]}. If there are sources, returns the id, extId (external ID), metadata (the date and time when the source was indexed), and a snippet of the text for each source. This snippet consists of (at most) two sentences from the text that contain the specified entity. These sentences are not necessarily in consecutive order.

  Filtering sources by entity match is further described in the "Filtering Sources" chapter of this guide.

- Sources by CRC: GET and POST to retrieve sources containing a specified CRC. The operations can be used to retrieve a CRC (concept:relation:concept), a CR (concept:relation), a RC (:relation:concept), or a CC (concept::concept). See **GetSourcesByCRC()**.

  If the domain exists but contains no sources that contain the CRC, returns {"sources":[]}. If there are sources that contain the CRC, returns the id, extId (external ID), metadata (the date and time when the source was indexed), and a snippet of the text for each source. This snippet consists of (at most) two sentences from the text that contain the specified CRC. These sentences are not necessarily in consecutive order.

  Note that a CR must be followed by a concept. Therefore the CR pilot:stated returns "pilot stated the airplane", pilot stated he" and "pilot stated \"I then". It does not return "pilot stated that" or "pilot stated to".

- Similar Sources: GET and POST to retrieve sources similar to a specified source. See **GetSimilarSources()**.

  If the domain exists but contains no similar sources, returns {"sources":[]}. If there are similar sources, returns a JSON array of entities in descending order by similarity score. If there are multiple similar entities that have the same similarity score, they are listed in ID order. For each similar source, returns the id, extId (external ID), score (similarity score), percentageMatched, percentageNew, numberOfEntitiesInRefSource, numberOfEntitiesInCommon, numberOfEntitiesInThisSource, metadata (the date and time when the source was indexed), and a snippet of the text for each source. This snippet consists of (at most) two sentences from the text that contain similar entities.

  Similar sources are further described in the "iKnow Queries" chapter of this guide.

- Source Details: GET and POST to retrieve details about a specified source. See **GetSourceDetails()**.

  For the specified source returns a virtual boolean, metadata (the date and time when the source was indexed), and the full text of the source.

- Add a Source: POST to add a source to a domain. See **AddSource()**.

  Adding sources are further described in the "Loading Text Data Programmatically" chapter of this guide.

- Delete a Source: DELETE to delete a source from a domain. See **DropSource()**.

  Deleting sources are further described in the "Loading Text Data Programmatically" chapter of this guide.

- Source Metadata: POST to update metadata for a specific source. See **SetMetadata()**.

4.2.3 Entities

The iKnow REST operations for entities are the following:

- Top Entities: GET and POST to retrieve the top entities in the specified domain. See **GetEntities()**.

  If there are no defined sources containing entities, returns {"entities":[]}. If there are entities, returns a JSON array of entities in descending order by frequency. If there are multiple entities that have the same frequency, they are listed in ID order. Each entity is listed as a JSON object with key/value pairs for id, value, frequency, and spread.

  Top entities are further described in the "iKnow Queries" chapter of this guide.

- Similar Entities: GET and POST to retrieve entities similar to a specified string. See **GetSimilarEntities()**.

  If there are no similar entities, returns {"entities":[]}. If there are similar entities, returns a JSON array of entities in descending order by frequency. If there are multiple similar entities that have the same frequency, they are listed in ID order. Each entity is listed as a JSON object with key/value pairs for id, value, frequency, and spread.

  Similar entities are further described in the "iKnow Queries" chapter of this guide.

- Related Entities: GET and POST to retrieve entities related to a specified entity. See **GetRelatedEntities()**.

If there are no related entities, returns {"entities":[]}. If there are related entities, returns a JSON array of entities in descending order by proximity. If there are multiple similar entities that have the same proximity, they are listed in ID order. Each entity is listed as a JSON object with key/value pairs for id, value, and proximity.

Related entities are further described in the "iKnow Queries" chapter of this guide.

- Entity Details: GET and POST to retrieve details about a specified entity. See **GetEntityDetails()**.

If the specified entity does not exist in the domain, or the specified domain does not exist, returns {"id":"","value":"aardvark"}. If the entity is found, returns a JSON object with key/value pairs for id, value, and the subarrays metricsAsConcept, metricsAsRelation, and metricsAsAny. Each of these subarrays lists frequency, spread, and tfidf.

If the entity only occurs as a concept, the metricsAsRelation values are 0 and the metricsAsAny values are the same as the metricsAsConcept values. If the entity only occurs as a relation, the metricsAsConcept values are 0 and the metricsAsAny values are the same as the metricsAsRelation values. If the entity occurs as both a concept and a relation, the metricsAsAny values for frequency and spread are the totals of their concept and relation values. The tfidf values is calculated from the corresponding concept and relation values.

4.2.4 Sentences

The iKnow REST operations for sentences are the following:

- Sentences by Entity: GET and POST to retrieve sentences containing a specified entity. See **GetSentencesByEntity()**.

If there are no sentences containing the specified entity, returns {"sentences":[]}. If there are sentences that contain the entity, returns a JSON array of sentences. For each sentence lists the sentence id, the source id, the text of the sentence, and the parts of the sentence. Parts are listed in sentence order, with each part an element of the parts array. For each part in the part array it lists partId, literal (with original letter case), role (concept, relation, pathRelevant, or nonRelevant), entityId, and entity (in lowercase letters). If the part is a nonRelevant, no entityId or entity are listed.

- Sentence Details: GET and POST to retrieve details about a specified sentence. See **GetSentenceDetails()**.

If the specified sentence exists, returns the sentence id, the parts of the sentence, the text, and the sourceId. Parts are listed in sentence order, with each part an element of the parts array. For each part in the part array it lists partId, literal (with original letter case), role (concept, relation, pathRelevant, or nonRelevant), entityId, and entity (in lowercase letters). If the part is a nonRelevant, no entityId or entity are listed.

4.2.5 Paths and CRCs

The iKnow REST operations for Paths and CRCs (concept-relation-concept strings) are the following:

- Top CRCs: GET and POST to retrieve the top CRCs in the specified domain. See **GetCRCs()**.

If there are no sources containing CRCs, returns {"crcs":[]}. Returns a JSON array of CRCs in descending order by frequency. If there are multiple CRCs that have the same frequency, they are listed in ID order. For each CRC it lists the id, master concept, relation, slave concept, frequency, and spread. For some CRCs, the master or the slave values are "" (the empty string. For example, `"master":""` would occur in a CRC that begins with the relation "according to..." or "during"; `"slave":""` would occur in a CRC with a relation such as "occurred" or "said".

- CRCs by Entity: GET and POST to retrieve CRCs containing a specified entity. See **GetCRCsByEntity()**.

If there are no sources with CRCs containing a specified entity, returns {"crcs":[]}. Otherwise, returns a JSON array of CRCs in descending order by frequency. If there are multiple CRCs that have the same frequency, they are listed in ID order. For each CRC it lists the id, master concept, relation, slave concept, frequency, and spread. For some CRCs, the master or slave key:value pair is omitted.

- Paths by Entity: GET and POST to retrieve Paths containing a specified entity. See **GetPathsByEntity()**.

If there are no sources with paths containing a specified entity, returns {"paths":[]}. Otherwise, returns a JSON array of paths, each consisting of a path id, an entities array, a sourceId and a sentenceId. The entities array contains for each entity an entity id, a partId, an entity value, a role specifier (concept, relation, or pathRelevant), and (optionally) an attributes array. An attributes array consists of type, (negation), level (path, word), and, if level is word, a words key:value pair. For example, `"attributes":[{"type":"negation","level":"path"}]}`.

4.2.6 Dictionaries and Matching

The iKnow REST operations for dictionaries are the following:

- Create Dictionary: POST to create a new dictionary. See **CreateDictionary()**.

  Creating a dictionary is further described in the "Smart Matching: Creating a Dictionary" chapter of this guide.

- Retrieve Dictionaries: GET and POST to retrieve the dictionaries defined for the specified domain. See **GetDictionaries()**.

  If there are no defined dictionaries, returns {"dictionaries":[]}.

- Retrieve Dictionary: GET and POST to retrieve the detailed contents of a specified dictionary. See **GetDictionaryDetails()**.

- Drop Dictionary: DELETE to drop (delete) a specified dictionary. See **DropDictionary()**.

- Add Items: POST to add one or more items to a dictionary. See **CreateDictionaryItems()**.

- Item Details: GET and POST to retrieve the detailed contents of a dictionary item. See **GetDictionaryItemDetails()**.

- Drop Item: DELETE to drop (delete) a specified dictionary item. See **DropDictionaryItem()**.

- Add Terms: POST to add one or more terms to a dictionary item. See **CreateDictionaryTerms()**.

- Drop Term: DELETE to drop (delete) a specified term from a dictionary item. See **DropDictionaryTerm()**.

The iKnow REST operations for matching operations using a dictionary are the following:

- Match Sources: GET and POST to match all sources against a dictionary. See **MatchAll()**.

  Filtering sources by dictionary match is further described in the "Filtering Sources" chapter of this guide.

- Match Source: GET and POST to match a specified source against a dictionary. See **GetMatchesBySource()**.

- Dictionary Matches: GET and POST to retrieve all dictionary matches for a specified source. May specify more than one dictionary. See **MatchSource()**.

- Item Matches: GET and POST to retrieve all matches for a specified dictionary item. See **GetMatchesByItem()**.

4.2.7 Blacklists

The iKnow REST operations for blacklists are the following:

- List Blacklists: GET and POST to retrieve a list of all blacklists for the domain. See **GetBlacklists()**.

  If there are no defined blacklists, returns {"blacklists":[]}. If there are one or more blacklists, returns a JSON array of blacklists, each blacklist listed as a JSON object with key/value pairs for id, name, and a JSON array of elements.

- Create Blacklist: GET and POST to create a new blacklist. See **CreateBlacklist()**.

  Creating a blacklist is further described in the "Blacklists" chapter of this guide.

- Retrieve Blacklist: GET and POST to retrieve the contents of a specified blacklist. See **GetBlacklistDetails()**.

Retrieves the blacklist specified by blacklist id as a JSON object with key/value pairs for id, name, and a JSON array of elements.

- Drop Blacklist: DELETE to drop (delete) a specified blacklist. See **DropBlacklist()**.

- Clear Blacklist: GET and POST to clear (delete) all of the contents of a specified blacklist. See **ClearBlacklist()**.

- Add Entity: GET and POST to add an entity to a specified blacklist. See **AddStringToBlacklist()**.

- Remove Entity: GET and POST to remove an entity from a specified blacklist. See **RemoveStringFromBlacklist()**.

5

Alternatives for Creating an iKnow Environment

Before using iKnow on source data, you must create an instance of the objects that define the iKnow environment. You then load source texts into this environment. The recommended way to do this is to use iKnow Architect. This chapter describes the iKnow environment objects in greater depth, and provides alternative ways to create and extend them.

The following comprise the iKnow environment:

* Domain: establishes the logical space for iKnow operations. Specifying a domain is mandatory.

* Configuration: establishes the language environment for source document content. Specifying a Configuration is optional. If not specified, iKnow provides a default. iKnow Architect allows you to define the language environment for a domain; this chapter describes how to create domain-independent Configurations, as well as additional features and options.

* UserDictionary: establishes a set of text substitutions to apply when loading source texts into iKnow. A UserDictionary is supplied to one or more Configurations. Specifying a UserDictionary is optional. If not specified, no UserDictionary is used. iKnow Architect does not provide UserDictionary features; this chapter describes how to add this functionality.

You can create multiple instances of Domains, Configurations, and UserDictionaries. These environment objects are independent of one another, and are independent of any specified set of source data.

5.1 iKnow Domains

All iKnow operations occur within a Domain. A domain is a iKnow defined unit within a Caché namespace. All source data to be used by iKnow is listed and loaded into a domain. A Caché namespace can contain multiple iKnow domains.

You can define, modify and delete an iKnow domain in three ways:

* Defining a domain using iKnow Architect. This is the easiest way to define a domain and to specify its languages, metadata, and data to be loaded.

* Defining a domain as a subclass of %iKnow.DomainDefinition. This option provides a powerful and fully-featured way to define a domain and specify its Configuration settings, metadata, and data to be loaded.

* Defining a domain programmatically using the %iKnow.Domain class methods and properties.

5.1.1 Defining a Domain Using DomainDefinition

When a user creates and compiles a class inheriting from %iKnow.DomainDefinition, the compiler will automatically create an iKnow domain corresponding to the settings specified in XML representation in the class's Domain XData block. The user can specify static elements, such as domain parameters, metadata field definitions, and an assigned Configuration, all of which are created automatically at compile time. In addition, the user can specify sources of text data to be loaded into the domain. Caché uses this source information to generate a dedicated **%Build()** method in a new class named [class-name].Domain. This **%Build()** method can then be used to load the specified data into the domain.

Use the following steps to define a domain by inheriting from %iKnow.DomainDefinition:

1. In Studio, select the desired namespace (**File**->**Change Namespace**), then create a new class definition (**File**->**New**, select the **Caché Class Definition** icon from the General tab). This invokes the New Class Wizard.

2. In the New Class Wizard specify a Package Name and Class Name of your choice. Press Next. In the Class Type box, select Extends and specify %iKnow.DomainDefinition as the name of the superclass. Press Finish.

3. In Studio, select **Class**->**Refactor**->**Override**. Select the XData tab. Select the Domain icon. Press OK. This creates an XData block.

4. Place the cursor within the XData block curly braces and type "<". Studio Assist immediately begins offering XML code options. Specify a domain name of your choice. You must specify at minimum the following:

```
{
 <domain name="AviationEvents">
 </domain>
}
```

Using XML syntax and Studio Assist options you can add other properties within the XData block (as described below). Minimally, you need a <data> </data> tags, and within them a data source. For example:

```
{
 <domain name="AviationEvents">
   <data>
    <files path="C:\MyDocs\" />
   </data>
 </domain>
}
```

5. In Studio, save the file (in this case as "MyDomain"), then select **Build**->**Compile**. This creates the domain.

The following is an example of this kind of domain definition:

Class Definition

```
Class Aviation.MyDomain Extends %iKnow.DomainDefinition
{
/// An XML representation of the domain this class defines.
XData Domain [ XMLNamespace = "http://www.intersystems.com/iknow" ]
 {
 <domain name="AviationEvents">
   <parameter name="Status" value="1" />
   <configuration name="MyConfig" detectLanguage="1" languages="en,es" />
   <parameter name="DefaultConfig" value="MyConfig" />
   <metadata>
     <field name="EventDate" dataType="DATE"/>
       <field name="Type" dataType="STRING" />
   </metadata>
  <data dropBeforeBuild="true">
    <files path="C:\MyDocs\" encoding="utf-8" recursive="1" extensions="txt"
       configuration="MyConfig" />
     <query sql="SELECT %ID,EventDate,Type,NarrativeFull
                    FROM Aviation.Event"
                    idField="ID" groupField="ID" dataFields="NarrativeFull"
     metadataFields="EventDate,Type" metadataColumns="EventDate,Type" />
   </data>
```

```
    </domain>
  }
}
```

At compile-time, this definition creates a domain "AviationEvents", with a Status parameter set to 1, and two metadata fields. It defines and assigns to the domain a Configuration "MyConfig" for processing English (en) and Spanish (es) texts.

This definition specifies the files to be loaded into this domain. It will load text (.txt) files from the C:\MyDocs\ directory, and it will load Caché SQL data from the Aviation.Event table. Refer to the %iKnow.Model.listFiles and %iKnow.Model.listQuery class properties for details.

The system generates a **%Build()** method in a dependent class named Aviation.MyDomain.Domain that contains the logic to load data from the C:\MyDocs directory and the Aviation.Event table.

To load the specified text data sources into this domain:

ObjectScript

```
SET stat=##class(Aviation.MyDomain).%Build()
```

After using **%Build()**, you can check for errors:

ObjectScript

```
DO $SYSTEM.iKnow.ListErrors("AviationEvents",0)
```

This lists three types of errors: errors, failed sources, and warnings.

To display the domains defined in the current namespace and the number of sources loaded for each domain:

ObjectScript

```
DO $SYSTEM.iKnow.ListDomains()
```

To display the metadata fields defined for this domain:

ObjectScript

```
DO $SYSTEM.iKnow.ListMetadata("AviationEvents")
```

iKnow assigns every domain one metadata field: DateIndexed. You can define additional metadata fields.

You can specify a <matching> element in the domain definition. The <matching> element describes dictionary information and specifies whether or not to automatically match loaded sources to these dictionaries. Caché performs basic validation on these objects during class compilation, but because they are loaded as part of the **%Build()** method, some name conflicts might only arise at runtime. Refer to the %iKnow.Model.matching class properties for details.

You can specify a <metrics> element in the domain definition. The <metrics> element adds custom metrics to the domain. No call to **%iKnow.Metrics.MetricDefinition.Register()** is required, because this is automatically performed by the Domain Definition code at compile time. Refer to the %iKnow.Model.metrics class properties for details.

5.1.2 Defining a Domain Programmatically

To define a new domain using class methods, invoke the **%iKnow.Domain.%New()** persistent method, supplying the domain name as the method parameter. A domain name can be any valid string; domain names are not case-sensitive. The name you assign to this domain must be unique for the current namespace. This method returns a domain object reference (oref) which is unique for all namespaces of the Caché instance. You must then save this instance using the **%Save()** method to make it persistent. The domain Id property (an integer value) is not defined until you save the instance as a persistent object, as shown in the following example:

ObjectScript

```
CreateDomain
    SET domOref=##class(%iKnow.Domain).%New("FirstExampleDomain")
    WRITE "Id before save: ",domOref.Id,!
    DO domOref.%Save()
    WRITE "Id after save: ",domOref.Id,!
CleanUp
    DO ##class(%iKnow.Domain).%DeleteId(domOref.Id)
    WRITE "All done"
```

Use **NameIndexExists()** to determine if the domain already exists. If the domain exists, use **NameIndexOpen()** to open it. If the domain doesn't exist, use **%New()** to create it and then use **%Save()**.

The following example checks whether a domain exists. If the domain doesn't exist, the program creates it. If the domain does exist, the program opens it. For the purpose of demonstration, this program then randomly either deletes or doesn't delete the domain.

ObjectScript

```
DomainCreateOrOpen
  SET domn="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(domn))
      { WRITE "The ",domn," domain already exists",!
        SET domo=##class(%iKnow.Domain).NameIndexOpen(domn)
        SET domId=domo.Id
      }
  ELSE {
      SET domo=##class(%iKnow.Domain).%New(domn)
      DO domo.%Save()
      SET domId=domo.Id
      WRITE "Created the ",domn," domain",!
      WRITE "with domain ID ",domId,! }
ContainsData
      SET x=domo.IsEmpty()
      IF x=1 {WRITE "Domain ",domn," contains no data",!}
      ELSE {WRITE "Domain ",domn," contains data",!}
CleanupForNextTime
      SET rnd=$RANDOM(2)
      IF rnd {
          SET stat=##class(%iKnow.Domain).%DeleteId(domId)
          IF stat {WRITE "Deleted the ",domn," domain" }
          ELSE { WRITE "Domain delete error:",stat }
          }
      ELSE {WRITE "No delete this time" }
```

The %iKnow.Domain class methods that create or open a domain are provided with an output %Status parameter. This parameter is set when the current system does not have license access to iKnow, and thus cannot create or open an iKnow domain.

5.1.3 Setting Domain Parameters

Domain parameters govern the behavior of a wide variety of iKnow operations. The specific parameters are described where applicable throughout this manual. For a list of available domain parameters, refer to the Appendix "Domain Parameters".

Note: In the examples that follow, domain parameters are referenced by their macro equivalent (for example, $$$IKP-FULLMATCHONLY), not their parameter name (For example, FullMatchOnly). The recommended programming practice is to use these %IKPublic macros rather than the parameter names.

All domain parameters take a default value. Commonly, iKnow will give optimal results without specifically setting any domain parameters. iKnow determines the value for each parameter as follows:

1. If you have specified a parameter value for the current domain, that value is used. Note that some parameters can only be set before loading data into a domain, while others can be set at any time. You can use the **IsEmpty()** method to determine if any data has been loaded into the current domain.

2. If you have specified a systemwide parameter value, that value is used as a default for all domains, except for a domain where a domain-specific value has been set.

3. If you have not specified a value for a parameter at either the domain level or the system level, iKnow uses its default value for that parameter.

5.1.3.1 Setting Parameters for the Current Domain

Once you have created a domain, you can set domain parameters for this specific domain using the **SetParameter()** instance method. **SetParameter()** returns a status indicating whether the parameter specified is valid and was set. **GetParameter()** returns the parameter value and the level at which the parameter was set (DEFAULT, DOMAIN, or SYSTEM). Note that **GetParameter()** does not check the validity of a parameter name; it returns DEFAULT for any parameter name it cannot identify as being set at the domain or system level.

The following example gets the default for the SortField domain parameter, sets this parameter for the current domain, then gets the value you set and the level at which it was set (DOMAIN):

ObjectScript

```
#include %IKPublic
DomainCreate
  SET domn="paramdomain"
    SET domo=##class(%iKnow.Domain).%New(domn)
    WRITE "Created the ",domn," domain",!
    DO domo.%Save()
DomainParameters
  SET sfval=domo.GetParameter($$$IKPSORTFIELD,.sf)
    WRITE "SortField before SET=",sfval," ",sf,!
  IF sfval=0 {WRITE "changing SortByFrequency to SortBySpread",!
          SET stat=domo.SetParameter($$$IKPSORTFIELD,1)
          IF stat=0 {WRITE "SetParameter failed"  QUIT} }
  WRITE "SortField after SET=",domo.GetParameter($$$IKPSORTFIELD,.str)," ",str,!!
CleanupForNextTime
  SET stat=##class(%iKnow.Domain).%DeleteId(domo.Id)
  IF stat {WRITE "Deleted the ",domn," domain" }
  ELSE { WRITE "Domain delete error:",stat }
```

5.1.3.2 Setting Parameters Systemwide

You can set domain parameters for all domains systemwide using the **SetSystemParameter()** method. A parameter set using this method immediately becomes the default parameter value for all existing and subsequently created domains in all namespaces. This systemwide default is overriden for an individual domain using the **SetParameter()** instance method.

Note: The SortField and Jobs domain parameters are exceptions. Setting these parameters at the system level has no effect on the domain settings.

You can determine if a domain parameter has been established as the system default using the **GetSystemParameter()** method. The initial value for a systemwide parameter is always the null string (no default).

If you wish to remove a systemwide default setting for a domain parameter, use the **UnsetSystemParameter()** method. Once a systemwide parameter setting has been established, you must unset it before you can set it to a new value. **UnsetSystemParameter()** returns a status of 1 (success) even when there was no parameter default value to unset.

The following example establishes a FullMatchOnly system-wide parameter value. If no system-wide default has been established, the program sets this systemwide parameter. If a systemwide default has been established, the program unsets this systemwide parameter, then sets it.

ObjectScript

```
#include %IKPublic
SystemwideParameterSet
  /* Initial set */
  SET stat=##class(%iKnow.Domain).SetSystemParameter($$$IKPFULLMATCHONLY,1)
  IF stat=1 {
```

```
        WRITE "FullMatchOnly set systemwide to: "
        WRITE ##class(%iKnow.Domain).GetSystemParameter($$$IKPFULLMATCHONLY),!
        QUIT }
    ELSE {
    /* Unset and Reset */
        SET stat=##class(%iKnow.Domain).UnsetSystemParameter($$$IKPFULLMATCHONLY)
        IF stat=1 {
           SET stat=##class(%iKnow.Domain).SetSystemParameter($$$IKPFULLMATCHONLY,1)
           IF stat=1 {
              WRITE "FullMatchOnly was unset systemwide",!,"then set to: "
              WRITE ##class(%iKnow.Domain).GetSystemParameter($$$IKPFULLMATCHONLY),!!
              GOTO CleanUpForNextTime }
           ELSE {WRITE "System Parameter set error",stat,!}
        }
        ELSE {WRITE "System Parameter set error",stat,!}
    }
CleanUpForNextTime
    SET stat=##class(%iKnow.Domain).UnsetSystemParameter($$$IKPFULLMATCHONLY)
    IF stat '=1 {WRITE "   Unset error status:",stat}
```

The following example shows that setting a systemwide parameter value immediately sets the parameter value for all domains. After setting a systemwide parameter value, you can override this value for individual domains:

ObjectScript

```
#include %IKPublic
SystemwideParameterUnset
   SET stat=##class(%iKnow.Domain).UnsetSystemParameter($$$IKPFULLMATCHONLY)
   WRITE "Systemwide setting
FullMatchOnly=",##class(%iKnow.Domain).GetSystemParameter($$$IKPFULLMATCHONLY),!!
Domain1Create
   SET domn1="mysysdomain1"
       SET domo1=##class(%iKnow.Domain).%New(domn1)
       DO domo1.%Save()
       SET dom1Id=domo1.Id
       WRITE "Created the ",domn1," domain ",dom1Id,!
       WRITE "FullMatchOnly=",domo1.GetParameter($$$IKPFULLMATCHONLY,.str)," ",str,!!
SystemwideParameterSet
   SET stat=##class(%iKnow.Domain).SetSystemParameter($$$IKPFULLMATCHONLY,1)
       IF stat=0 {WRITE "SetSystemParameter failed"  QUIT}
   WRITE "Set systemwide FullMatchOnly=",##class(%iKnow.Domain).GetSystemParameter($$$IKPFULLMATCHONLY),!!
Domain2Create
   SET domn2="mysysdomain2"
       SET domo2=##class(%iKnow.Domain).%New(domn2)
       DO domo2.%Save()
       SET dom2Id=domo2.Id
       WRITE "Created the ",domn2," domain ",dom2Id,!
       WRITE "Domain setting FullMatchOnly=",domo2.GetParameter($$$IKPFULLMATCHONLY,.str)," ",str,!!
DomainParameters
   WRITE "New domain ",dom2Id," FullMatchOnly=",domo2.GetParameter($$$IKPFULLMATCHONLY,.str)," ",str,!
   WRITE "Existing domain ",dom1Id," FullMatchOnly=",domo1.GetParameter($$$IKPFULLMATCHONLY,.str),"
",str,!!
OverrideForOneDomain
   SET stat=domo1.SetParameter($$$IKPFULLMATCHONLY,0)
       IF stat=0 {WRITE "SetParameter failed"  QUIT}
   WRITE "Domain override FullMatchOnly=",domo1.GetParameter($$$IKPFULLMATCHONLY,.str)," ",str,!
CleanupForNextTime
   SET stat=##class(%iKnow.Domain).%DeleteId(dom1Id)
   SET stat=##class(%iKnow.Domain).%DeleteId(dom2Id)
   SET stat=##class(%iKnow.Domain).UnsetSystemParameter($$$IKPFULLMATCHONLY)
```

5.1.4 Assigning to a Domain

Once you have created a domain and (optionally) specified its domain parameters, you can assign various components to that domain:

- Source Data: After creating a domain, you commonly will load a number (usually a large number) of text sources into a domain; this generates iKnow indexed data within that domain. Loading text sources is a required precondition for most iKnow operations. A variety of text sources are supported, including files, SQL fields, and text strings. After iKnow has indexed a data source, the original data source can be removed without affecting iKnow processing. Changing a data source has no effect on iKnow processing, unless you re-load that data source to update the iKnow indexed data in the domain.

- **Filters**: After creating a domain, you can optionally create one or more filters for that domain. A filter specifies criteria used to exclude some of the loaded sources from a query. Thus a filter allows you to perform iKnow operations on a subset of the data loaded in the domain.

- **Metadata**: After creating a domain, you can optionally specify one or more metadata fields that you can use as criteria for filtering sources. A metadata field is data associated with a source that is not iKnow indexed data. For example, the date and time that a text source was loaded is a metadata field for that source. Metadata fields must be defined *before* loading text sources into a domain.

- **Blacklists**: After creating a domain, you can optionally create one or more blacklists for that domain. A blacklist is a list of entities (such as words or phrases) that you do *not* want a query to return. Thus a blacklist allows you to perform iKnow operations that ignore specific data entities in data sources loaded in the domain.

- **Dictionaries**: After creating a domain, you can optionally create one or more dictionaries for that domain. A dictionary contains entities that are used to match the iKnow indexed data.

These components are defined using various iKnow classes and methods. You can also use the Caché iKnow Architect to define metadata fields, load sources, and define blacklists.

Metadata fields must be defined before loading sources. Filters, blacklists, and dictionaries can be defined or modified at any time.

5.1.5 Deleting All Data from a Domain

Deleting or changing an original source text has no effect on the source data listed and loaded from that text into a iKnow domain. You must explicitly add or delete a source to the set of indexed sources.

The **%DeleteId()** persistent method deletes a domain and all source data that has been listed and loaded in that domain. You can use the **DropData()** method to delete all source data that has been loaded into a domain without deleting the domain itself. Either method deletes all indexed source data, allowing you to start over with a new set of data sources.

Note: When deleting a domain that contains a significant number of sources, use **DropData()** to delete the data before using **%DeleteId()** to delete the domain. If you use **%DeleteId()** to delete a domain while it still has data in it, Caché will delete the data, but it will journal each data deletion, even if journaling has been disabled. Deleting the data, and then deleting the domain prevents the generation of these large journal files.

You can use the **IsEmpty()** method to determine if any data has been loaded into a domain.

The following example demonstrates deleting the data from a domain. If the named domain doesn't exist, the program creates the domain. If the named domain does exist, the program tests for the presence of data. If there is data in the domain, the program opens the domain and deletes the data.

ObjectScript

```
DomainCreateOrOpen
  SET dname="mytestdomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { WRITE "The ",dname," domain already exists",!
       SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
       IF domoref.IsEmpty() {GOTO RestOfProgram}
       ELSE {GOTO DeleteData }
     }
  ELSE
     { WRITE "The ",dname," domain does not exist",!
       SET domoref=##class(%iKnow.Domain).%New(dname)
       DO domoref.%Save()
       WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
       GOTO RestOfProgram }
DeleteData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!
            GOTO RestOfProgram }
  ELSE    { WRITE "DropData error",!
```

```
            QUIT}
RestOfProgram
  WRITE "The ",dname," domain contains no data"
```

5.1.6 Listing All Domains

You can use the GetAllDomains query to list all current domains in all namespaces. This is shown in the following example:

ObjectScript

```
SET stmt=##class(%SQL.Statement).%New()
SET status=stmt.%PrepareClassQuery("%iKnow.Domain","GetAllDomains")
   IF status'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(status) QUIT}
SET rset= stmt.%Execute()
WRITE !,"Domains in all namespaces",!
DO rset.%Display()
```

Each domain is listed on a separate line, using the following format: `domainId:domainName:namespace:version`.

The *Version* property is an integer that shows what version of iKnow data structure was used when the domain was created. The iKnow system version number changes when a release contains a change to the iKnow data structures. Therefore, a new version of Caché or the introduction of new iKnow features may not change the iKnow system version number. If the Version property value for a domain is not the current iKnow system version, you may wish to upgrade the domain to take advantage of the latest features of iKnow. See Upgrading iKnow Data in the "iKnow Implementation" chapter.

By default, **GetAllDomains** lists all the current domains for all namespaces. You can specify a boolean argument in **%Execute()** to limit the listing of domains to the current namespace, as shown in the following example:

ObjectScript

```
SET stmt=##class(%SQL.Statement).%New()
SET status=stmt.%PrepareClassQuery("%iKnow.Domain","GetAllDomains")
   IF status'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(status) QUIT}
SET rset= stmt.%Execute(1)
WRITE !,"Domains in all namespaces",!
DO rset.%Display()
```

A boolean value of 1 limits listing to domains in the current namespace. A boolean value of 0 (the default) lists all domains in all namespaces. (Note: listed Version property values may not be correct for domains other than the current domain.)

You can also list all domains in the current namespace using:

ObjectScript

```
DO ##class(%SYSTEM.iKnow).ListDomains()
```

This method lists the domain Ids, domain names, number of sources, and the domain version number.

5.1.7 Renaming a Domain

You can use the **Rename()** class method to change the name of an existing domain within the current namespace, as shown in the following example:

ObjectScript

```
SET stat=##class(%iKnow.Domain).Rename(oldname,newname)
IF stat=1 {WRITE "renamed ",oldname," to ",newname,!}
ELSE {WRITE "no rename",oldname," is unchanged",! }
```

Renaming a domain changes the name used to open the domain, assigning the existing Domain Id to the new name. **Rename()** does not change the name of a current instance of the domain. For a rename to occur, the old domain name must exist and the new domain name must not exist.

5.1.8 Copying a Domain

You can copy an existing domain to a new domain in the current namespace by using the **CopyDomain()** method of the %iKnow.Utils.CopyUtils class. The **CopyDomain()** method copies a domain definition to a new domain, assigning a unique domain name and domain Id; the existing domain is unchanged. If the new domain does not exist, this method creates a new domain. By default, this method copies the domain parameter settings and assigned domain components from the existing domain to the copy, if these components are present.

By default, the **CopyDomain()** method copies the source data from the existing domain to the copy. However, if source data copying is requested and no source data is present in the existing domain, the **CopyDomain()** operation fails.

The following example copies a the domain named "mydomain" and its parameter settings and source data to a new domain named "mydupdomain". Because "mydomain" contains no source data, the 3rd argument (which specifies whether to copy source data) is set to 0:

ObjectScript

```
DomainMustExistToBeCopied
  SET olddom="mydomain"_$PIECE($H,",",2)
  SET domo=##class(%iKnow.Domain).%New(olddom)
    DO domo.%Save()
  IF (##class(%iKnow.Domain).NameIndexExists(olddom))
    {WRITE "Old domain exists, proceed with copy",!!}
  ELSE {WRITE "Old domain does not exist" QUIT}
CopyDomain
  SET newdom="mydupdomain"
  IF (##class(%iKnow.Domain).NameIndexExists(newdom))
    {WRITE "Domain copy overwriting domain ",newdom,!}
  ELSE {WRITE "Domain copy creating domain ",newdom,!}
  SET stat=##class(%iKnow.Utils.CopyUtils).CopyDomain(olddom,newdom,0)
  IF stat=1 {WRITE !!,"Copied ",olddom," to ",newdom," copying all assignments",!!}
  ELSE {WRITE "Domain copy failed with status ",stat,!}
CleanUp
  SET stat=##class(%iKnow.Domain).%DeleteId(domo.Id)
  WRITE "Deleted the old domain",!
```

The **CopyDomain()** method allows you to quickly copy all of the domain settings, source data, and assigned components of an existing domain to a new domain. It provides boolean options for all-or-nothing copying of assigned components. Other methods in the %iKnow.Utils.CopyUtils class provide greater control in specifying which assigned components to copy from one existing domain to another.

5.2 iKnow Configurations

An iKnow configuration specifies behavior for handling source documents. It is only used during the source data loading operation. A configuration is specific to its namespace; you can create multiple configurations within a namespace. iKnow assigns each configuration in a namespace a configuration Id, a unique integer. Configuration Id values are not reused. You can apply the same configuration to different domains and source text loads. Defining or using an iKnow configuration is optional; if you don't specify a configuration, iKnow uses the property defaults.

You can define an iKnow configuration in two ways:

• Using the %iKnow.Configuration class methods and properties, as described in this chapter.

• Using the iKnow Architect to specify supported languages as part of domain definition.

5.2.1 Defining a Configuration

You can define a configuration using the **%New()** persistent method of the %iKnow.Configuration class.

You can determine if an iKnow configuration with that name already exists by invoking the **Exists()** method. If the configuration exists, you can open it using the **Open()** method, as shown in the following example:

ObjectScript

```
IF ##class(%iKnow.Configuration).Exists("EnFr") {
      SET cfg=##class(%iKnow.Configuration).Open("EnFr") }
  ELSE { SET cfg=##class(%iKnow.Configuration).%New("EnFr",1,$LB("en","fr"))
       DO cfg.%Save() }
```

5.2.2 Setting Configuration Properties

A configuration defines the following properties:

- Name: A configuration name can be any valid string; configuration names are not case-sensitive. The name you assign to this configuration must be unique for the current namespace.

- DetectLanguage: A boolean value that specifies whether to use automatic language identification if more that one language is specified in the Languages property. Because this option may have a significant effect on performance it should not be set unless needed. The default is 0 (do not use automatic language identification).

- Languages: What language(s) the source documents contain, and therefore which languages to test for and which language models to apply. The available options are Dutch (nl), English (en), French (fr), German (de), Japanese (ja), Portuguese (pt), Russian (ru), Spanish (es), Swedish (sv), and Ukrainian (uk). The default is English (en). Languages are always specified using their ISO 639-1 two-letter abbreviation. This property value is specified as a Caché list of strings (using **$LISTBUILD**).

- UserDictionary: Either the name of a defined UserDictionary object or the file path location of defined UserDictionary file. A UserDictionary contains user-defined substitution pairs that iKnow applies to the source text entities during the load operation. This property is optional; the default is the null string.

- Summarize: a boolean value that specifies whether to store summary information when loading source texts. If set to 1, source information is generated that iKnow requires to generate summaries of the loaded source texts. If set to 0, no summaries can be generated for the sources processed with this Configuration object. Setting this option to 1 is generally recommended. The default is 1.

All configuration properties (except the Name) are assigned default values. You can get or set a configuration property by using property dispatch:

ObjectScript

```
  IF cfgOref.DetectLanguage=0 {
    SET cfgOref.DetectLanguage=1
    DO cfgOref.%Save() }
```

Note that you must first **%Save()** the newly created configuration before you can change its properties using property dispatch, and then you must **%Save()** the configuration after changing the property values.

The following example creates a configuration that supports English and French with automatic language identification. It then changes the configuration to support English and Spanish:

ObjectScript

```
OpenOrCreateConfiguration
  SET myconfig="Bilingual"
  IF ##class(%iKnow.Configuration).Exists(myconfig) {
      SET cfg=##class(%iKnow.Configuration).Open(myconfig)
      WRITE "Opened existing configuration ",myconfig,! }
  ELSE { SET cfg=##class(%iKnow.Configuration).%New(myconfig,1,$LB("en","fr"))
       DO cfg.%Save()
      WRITE "Created new configuration ",myconfig,! }
GetLanguages
     WRITE "that supports ",$LISTTOSTRING(cfg.Languages),!
```

```
SetConfigParameters
     SET cfg.Languages=$LISTBUILD("en","sp")
     DO cfg.%Save()
     WRITE "changed ",myconfig," to support ",$LISTTOSTRING(cfg.Languages),!
CleanUpForNextTime
  SET rnd=$RANDOM(2)
  IF rnd {
       SET stat=##class(%iKnow.Configuration).%DeleteId(cfg.Id)
       IF stat {WRITE "Deleted the ",myconfig," configuration" }
       }
  ELSE {WRITE "No delete this time",! }
```

For a description of using multiple languages and automatic language identification, refer to the "Language Identification" chapter of this manual.

5.2.3 Using a Configuration

You can apply a defined configuration in any of the following ways:

- Defining the DefaultConfig domain parameter.

- Specifying the configuration as the first argument of the **Init()** instance method to initialize the Lister instance and override the configuration default.

- Invoking **%iKnow.Source.Lister.SetConfig()**.

- Specifying the configuration as an argument of the **loader.ProcessBuffer()** or **loader.ProcessVirtualBuffer()** method.

5.2.4 Listing All Configurations

You can use the GetAllConfigurations query to list all defined configurations in the current namespace. This is shown in the following example:

ObjectScript

```
SET stmt=##class(%SQL.Statement).%New()
SET status=stmt.%PrepareClassQuery("%iKnow.Configuration","GetAllConfigurations")
    IF status'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(status) QUIT}
SET rset= stmt.%Execute()
WRITE "The current namespace is: ",$NAMESPACE,!
WRITE "It contains the following configurations: ",!
DO rset.%Display()
```

Each configuration is listed on a separate line, listing the configuration Id followed by the configuration parameter values. Listed values are separated by colons. If the configuration is defined with a list of supported languages, **GetAllConfigurations** displays these language abbreviations separated by commas.

You can also list all configurations in the current namespace using:

ObjectScript

```
DO ##class(%SYSTEM.iKnow).ListConfigurations()
```

5.2.5 Using a Configuration to Normalize a String

Using a defined iKnow configuration, you can perform iKnow text normalization on a string using the **Normalize()** method. This method both normalizes the string characters and (optionally) applies a UserDictionary, as shown in the following example:

ObjectScript

```
DefineUserDictionary
  SET time=$PIECE($H,",",2)
  SET udname="Abbrev"_time
  SET udict=##class(%iKnow.UserDictionary).%New(udname)
  DO udict.%Save()
  DO udict.AddEntry("Dr.","Doctor")
  DO udict.AddEntry("Mr.","Mister")
  DO udict.AddEntry("\&\","and")
DisplayUserDictionary
  DO udict.GetEntries(.dictlist)
  SET i=1
  WHILE $DATA(dictlist(i)) {
    WRITE $LISTTOSTRING(dictlist(i),",",1),!
    SET i=i+1 }
  WRITE "End of UserDictionary",!!
DefineConfiguration
  SET cfg=##class(%iKnow.Configuration).%New("EnUDict"_time,0,$LB("en"),udname)
  DO cfg.%Save()
NormalizeAString
  SET mystring="...The Strange Case  of Dr. Jekyll     & Mr. Hyde"
  SET normstring=cfg.Normalize(mystring)
  WRITE normstring
CleanUp
  DO ##class(%iKnow.UserDictionary).%DeleteId(udict.Id)
  DO ##class(%iKnow.Configuration).%DeleteId(cfg.Id)
```

You can perform iKnow text normalization on a string independent of a configuration using the **NormalizeWithParams()** method.

These methods perform these operations, in the following order:

1. Apply a UserDictionary, if one is specified

2. Perform iKnow language model preprocessing

3. Convert all text to lowercase letters

4. Replace multiple whitespace characters with a single space

5.3 iKnow UserDictionary

A UserDictionary specifies a set of user-defined paired terms applied to the source texts. iKnow substitutes each occurrence of the first term of the pair with the second term as part of source text listing. This operation changes the source text used by iKnow; all subsequent iKnow operations see only the substituted term. For example, if UserDictionary replaces the abbreviation "Dr." with "Doctor", every occurrence of "Dr." is replaced by the word "Doctor" in the data indexed by iKnow. The original source file is not changed, but all representations of the source text within iKnow contain this substitution. Unlike all other components of iKnow, UserDictionary changes the source content before listing and loading.

You can use the UserDictionary to substitute one term for another, to expand acronyms and abbreviations (or the reverse), or to avoid or cause a sentence break.

Substitution pairs are applied *before* iKnow text normalization, which converts the iKnow internal text representation to lowercase letters. For this reason, substitution pairs are case-sensitive. Thus, to replace all instances of "physician" with "doctor" you will need the substitution pairs `"physician","doctor"`, `"Physician","Doctor"`, and perhaps `"PHYSICIAN","DOCTOR"`.

Defining a UserDictionary is optional. A UserDictionary exists independent of any specific configuration or domain. A defined UserDictionary can be assigned as a Configuration property. Only one UserDictionary can be assigned to a Configuration. The same UserDictionary can be assigned to multiple Configurations.

A defined UserDictionary can also be specified to the **NormalizeWithParams()** method, independent of any Configuration.

Note: You cannot modify an existing configuration; a **%New()** does not delete/replace an existing configuration. Therefore, to add a UserDictionary to an existing configuration you must explicitly delete then re-create the named configuration. Alternatively, you can create a new configuration with a new configuration name.

The UserDictionary is applied to sources when the sources are listed; already indexed sources are not affected by changes to UserDictionary.

5.3.1 UserDictionary Format

UserDictionary pairs often perform the simple substitution of a term for an equivalent term. For example, replacing every occurrence of "physician" with "doctor". Using the backslash character provides additional formatting options:

Format	Meaning
\	Only perform substitution if a blank space occurs here.
\noend	Do not issue a sentence break.
\end	Issue a sentence break.

These are shown in the following sample UserDictionary pairs:

```
\UK,United Kingdom
\+\,plus
Fr.,\noend
\STOP,\end
```

5.3.2 Defining a UserDictionary as an Object Instance

You must first create a UserDictionary object, then populate that instance.

ObjectScript

```
SET udict=##class(%iKnow.UserDictionary).%New("MyUserDict")
DO udict.%Save()
DO udict.AddEntry("Dr.","Doctor")
DO udict.AddEntry("physician","doctor")
DO udict.AddEntry("Physician","Doctor")
```

To populate a UserDictionary object, you use the **AddEntry()** method to specify substitution pairs. Each substitution pair requires a separate **AddEntry()** with the following format: AddEntry(*oldstring*,*newstring*). Note that substitution is string substitution, and that pairs are case-sensitive. You can, optionally, specify the position at which to add the UserDictionary entry (the *position* default is to add the entry at the end of the UserDictionary). Because iKnow applies substitution pairs in UserDictionary order, you can use *position* to perform additive substitutions. For example, first replace "PA" with "physician's assistant", then replace "physician" with "doctor".

To assign a UserDictionary object, you supply the UserDictionary name as the 4th argument in the Configuration **%New()** method:

ObjectScript

```
SET cfg=##class(%iKnow.Configuration).%New("MyConfig",0,$LISTBUILD("en"),"MyUserDict",1)
DO cfg.%Save()
```

5.3.3 Defining a UserDictionary as a File

You must first create a UserDictionary file, populate it, then assign this UserDictionary file to a Configuration.

A UserDictionary file must be a text file in UTF-8 format encoding.

To populate a UserDictionary file, you specify substitution pairs in a text file. Each substitution pair is a separate line with the following format: *oldstring*,*newstring*. Note that substitution is string substitution, and that pairs are case-sensitive. The following is a sample UserDictionary file:

```
Mr.,Mister
Dr.,Doctor
Fr.,Fr
\UK,United Kingdom
```

To assign a UserDictionary file, you supply the full pathname as the 4th argument in the Configuration **%New()** method:

ObjectScript

```
SET cfg=##class(%iKnow.Configuration).%New(myconfig,0,$LISTBUILD("en"),"C:\temp\udict.txt",1)
DO cfg.%Save()
```

6

Loading Text Data Programmatically

Before iKnow can analyze text data, the data sources must be loaded into a domain. This can be done in three ways:

- Using iKnow Architect to specify the data locations source texts for a domain. The **Build** button loads the specified sources into the domain.

- Creating an subclass allows you to specify the data locations for source texts for a domain. It generates a **%Build()** method in a dependent class that contains the logic to load this data.

- Specifying a Loader and Lister programmatically to load the specified sources into a domain, as described in this chapter.

To make text data available for iKnow analysis, the domain must invoke an instance of a Loader and a Lister. The Loader supervises iKnow processing of text sources, using the Lister and a Processor. The Lister identifies the text sources to be used by the Loader. iKnow provides a variety of Listers for different types of source text data. Each Lister, by default, automatically invokes the corresponding Processor with default parameters. There is one Loader used for data sources of all types.

Note that the Loader and Lister objects can be created in any order, but both must have been created before you invoke the Lister **AddListToBatch()** instance method and then the Loader **ProcessBatch()** instance method (or other equivalent Lister and Loader methods).

6.1 Loader

The Loader (%iKnow.Source.Loader) is the main class coordinating the loading process. You must create a new loader object for the domain. To create a loader object:

ObjectScript

```
SET domo=##class(%iKnow.Domain).NameIndexOpen("mydomain")
    SET domId=domo.Id
SET myloader=##class(%iKnow.Source.Loader).%New(domId)
```

After creating a loader and a lister, you issue instance methods to list and process the sources. For example, when performing a batch load you issue the Lister **AddListToBatch()** instance method to list the text sources. You then issue the Loader **ProcessBatch()** instance method to process the listed sources. This Loader method calls the Lister to scan the locations marked by **AddListToBatch()**, then calls the Processor to read those documents and push them to the iKnow engine and finally, it invokes the ^%iKnow.BuildGlobals routine to process the staging globals loaded by the iKnow engine.

6.1.1 Loader Error Logging

If a load operation completes, but encounters errors in loading one or more sources, these errors are recorded in an error log. Errors of varying severity can be retrieved using the **GetErrors()**, **GetWarnings()**, and **GetFailed()** methods. For example, a failed load error (**GetFailed()**) occurs if you attempt to load a source file that has no contents. A warning load error (**GetWarnings()**) occurs if there is an error in the source metadata.

You can use the **ClearLogs()** method to clear the error log of error messages at any or all of these severity levels.

6.1.2 Loader Reset()

If a load operation didn't complete in an expected fashion and you want to start from scratch, you should invoke the **Reset()** method for the loader instance, as follows:

ObjectScript

```
DO myloader.Reset()
```

6.2 Lister

The Lister identifies text files, records, or other sources of unstructured data you wish iKnow to index. That is, all text that will eventually end up as a Source in the domain. The unit of content in iKnow is a Source, which can represent any unit of text you wish to analyze, such as a text file, a record in a SQL table, an RSS posting, or other text source.

Usually a Source is a text containing multiple sentences. However, a source can contain content of any type. For example, a file containing the number 123 is treated as a Source containing one sentence. A file with no contents is not listed as a Source.

All listers are found in class %iKnow.Source.Lister and have their own specific type of sources they can scan. For example, the subclass %iKnow.Source.File.Lister scans a file system and the subclass %iKnow.Source.RSS.Lister scans RSS web feeds, such as blog postings, in XML file format. iKnow provides seven listers for different types of sources. You can also create your own custom lister.

Most text sources require a Lister. However, text that is directly specified as a string does not require a Lister.

Through the **AddListToBatch()** method you can instruct the Lister to look into a specific directory, SQL table, or RSS feed for Sources. The lister parameters depend on the actual Lister class.

6.2.1 Initializing a Lister

You can create a Lister instance for a domain using the **%New()** method for that type of lister, supplying the domain Id. The following example creates two listers within the specified domain:

ObjectScript

```
SET domo=##class(%iKnow.Domain).NameIndexOpen("mydomain")
    SET domId=domo.Id
SET flister=##class(%iKnow.Source.File.Lister).%New(domId)
WRITE flister,!
SET rlister=##class(%iKnow.Source.RSS.Lister).%New(domId)
WRITE rlister
```

Each lister automatically invokes the corresponding processor, as follows:

- The File.Lister invokes the File.Processor.

- The Global.Lister invokes the Global.Processor.

- The Domain.Lister invokes the Domain.Processor.

- All other Listers invokes the Temp.Processor. The `%iKnow.Source.Temp.Processor` has that name because it processes temporary globals that are automatically created and deleted by iKnow during the loading process.

Each processor has default processor parameters, which are appropriate for most iKnow sources. Therefore, in most cases, you do not need to specify a processor or processor parameters. If you do not specify a processor, iKnow uses the default processor, as shown by the **DefaultProcessor()** method.

6.2.2 Overriding Lister Instance Defaults

In most cases, the lister instance defaults are appropriate for the processing of your iKnow sources.

If you wish to overriding lister instance defaults for Configuration, Processor, or Converter objects, you can, optionally, use the **Init()** instance method to initialize the Lister instance. If you omit **Init()** the defaults are used.

The complete Lister initialization is as follows:

```
Init(config,processor,processorparams,converter,converterparams)
```

To specify the default for any of these items, specify the empty string ("") as the Init() parameter value.

You can also initialize these objects separately using the **SetConfig()**, **SetProcessor()**, and **SetConverter()** methods.

- Configuration (Config): If you do not specify a configuration, iKnow uses the default configuration. A configuration specifies what language(s) the text documents contain, and whether or not automatic language identification should be used. A configuration object is not domain-specific; you can use the same configuration for multiple domains. While not required, explicitly specifying a configuration is recommended.

- Processor: Using **lister.Init()** you can specify a processor and processor parameters. A processor reads the texts into iKnow. Specifying a processor is optional. If you do not specify a processor, iKnow uses the default processor and its parameter defaults. If you specify a processor, you can specify the processor parameter values, as shown in the following example:

  ### ObjectScript

  ```
  SET flister=##class(%iKnow.Source.File.Lister).%New(domId)
  SET processor="%iKnow.Source.File.Processor"
  SET pparams=$LB("Latin1")
  DO flister.Init("",processor,pparams,"","")
  ```

  If explicitly specified, the processor subclass should be either of the same type as the Lister subclass (for example, %iKnow.Source.File.Lister takes %iKnow.Source.File.Processor) or %iKnow.Source.Temp.Processor if the Lister subclass has no corresponding Processor subclass. You can also create your own custom processor.

  Processor parameters are specified as a Caché list. For %iKnow.Source.File.Processor the first list element is the name of the character set used (for example "Latin1"). The %iKnow.Source.Temp.Processor does not take any processor parameters.

- Converter: Using **lister.Init()** you can specify a user-defined converter and converter parameters. A Converter converts formatted source documents to plain text, removing HTML or XML tags, PDF formatting, or other non-text contents. Usually separate converters are used for each source document formatting type. Specifying a converter is optional. The default is to use no converter. If no converter is used, iKnow indexes formatting contents as well as text contents.

6.2.3 Lister Assigns IDs to Sources

The lister assigns two unique IDs to each source:

- Source ID (internal ID): a unique integer assigned by iKnow that is used for iKnow internal processing.

- External ID: a unique identifying string or number. The External ID is used as the link for any user-specified application that wishes to use iKnow. The External ID has the following structure:

```
ListerReference:FullReference
```

The Lister Reference is either the full class name of the Lister class used to load this source, or a short alias defined by the Lister class itself, prefixed with a colon. The Full Reference is a string for which the format is defined by the Lister class. It contains a Group Name and a Local Reference. It is up to the Lister to provide the implementation to derive the Group Name and Local Reference from this Full Reference, and to rebuild the Full Reference from the Group Name and Local Reference.

For example, the text file external ID `:FILE:c:\mytextfiles\mydoc.txt` consists of:

- ListerReference: the Lister class alias `:FILE`

- FullReference: `c:\mytextfiles\mydoc.txt`, which consists of the Group Name `c:\mytextfiles\` and the Local Reference `mydoc.txt`.

For data in an SQL table, the ListerReference is `:SQL`. The Group Name is the groupfield, a field in the record that contains a unique value, and the Local Reference is the row ID.

For data in a string or global variable, the ListerReference is `:TEMP`.

The external ID format described here is the default; external ID format is configurable using the SimpleExtIds domain parameter.

You can access a source using either ID. The %iKnow.Queries.SourceAPI class contains methods for accessing these IDs. The **GetByDomain()** method returns both IDs for each source. Given the source ID, the **GetExternalId()** method returns the external ID. Given the external ID, the **GetSourceId()** method returns the source ID.

You can determine the lister class alias using the **GetAlias()** method of the %iKnow.Source.File.Lister class. If no alias exists, the External ID contains the full Lister class name.

6.2.4 Lister Defaults Example

The following is a minimal Lister and Loader example, taking all defaults. It establishes a domain, then creates Lister and Loader instance objects for that domain. It does not invoke **lister.Init()**, but takes the defaults for configuration, processor, and converter. It then lists and loads a directory of user-defined .txt and .log files:

ObjectScript

```
  SET domo=##class(%iKnow.Domain).NameIndexOpen("mydomain")
      SET domId=domo.Id
SetListerAndLoader
  SET mylister=##class(%iKnow.Source.File.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
UseListerAndLoader
  SET install=$SYSTEM.Util.InstallDirectory()
  SET dirpath=install_"mgr\Temp\iknow\mytextfiles"
  SET stat=mylister.AddListToBatch(dirpath,$LB("txt","log"),0,"")
      WRITE "The lister status is ",$System.Status.DisplayError(stat),!
  SET stat=myloader.ProcessBatch()
      WRITE "The loader status is ",$System.Status.DisplayError(stat),!
```

Most examples in this book delete old data before using the Lister and Loader; this old data deletion is for demonstration purposes to allow these examples to be run repeatedly. Most examples in this book do not specify the processor and processor parameters, taking the defaults. Many examples in this book specify values for configuration rather than taking the defaults.

6.2.5 Lister Parameters

When you invoke a method to specify sources, you specify Lister parameters. You specify the same Lister parameters for the **AddListToBatch()** Lister instance method (for large batch loads of sources) and the **ProcessList()** Loader instance method (for adding a small number of sources to an existing batch of sources).

There are four Lister parameters that cumulatively define which sources are to be listed for iKnow indexing:

- Path: the location where the sources are located, specified as a string. This parameter is mandatory.

- Extensions: one or more file extension suffixes that identify which sources are to be listed. Specified as a Caché list data structure, each element of which is a string (refer to $LISTBUILD for details on Caché list data structures). By default the Lister selects all files in the Path directory that contain data, regardless of their file extension suffix. This includes files with no file extension suffix or with a file extension suffix indicating a non-text (such as .jpg). Empty files are not selected. Directories are not selected. When an extension suffix parameter is specified, the Lister selects only those files in the Path directory with that file extension suffix (or with no file extension suffix) that contain data.

- Recursive: a boolean value that specifies whether to search subdirectories of the path for sources. If selected, multiple levels of subdirectories are searched for sources. 1 = include subdirectories. 0 = do not include subdirectories. The default is 0.

- Filter: a string specifying a filter used to limit which sources are to be listed for iKnow indexing. For example, a user-designed filter could limit the Lister to only those files that have a specified substring in their file names. The default is to use no filter. (Note that this use of the word "filter" is completely separate from the filters in the %iKnow.Filters class that are used to include or exclude already-indexed sources supplied to an iKnow query.)

6.2.6 Batch or List?

iKnow provides two ways to load sources of all types, batch loading (**ProcessBatch()**) or list loading (**ProcessList()**). Both perform the same processing, they differ in their speed of execution. Which one you use depends primarily on how many sources you are loading. As a general rule, when loading ten or fewer sources, use **ProcessList()**; when loading one hundred or more sources, use **ProcessBatch()**. Which to use on intermediate numbers of sources depends on the nature of the specific sources.

6.3 Listing and Loading Examples

The examples in this section show the different ways to load sources:

- **lister.AddListToBatch()** and **loader.ProcessBatch()** to batch load a large number of sources.

- **loader.SetLister()** and **loader.ProcessList()** to load a small number of sources, or to add sources to an existing batch load.

- **loader.BufferSource()** and **loader.ProcessBuffer()** to load a string as a source. You can, of course, specify a local or global variable that contains the string.

You can also load sources as virtual sources using **loader.ProcessVirtualList()** or **loader.ProcessVirtualBuffer()**, as described in Loading a Virtual Source.

6.3.1 Loading Files

The following executable example performs a batch load of the source files in the Windows directory *dirpath* that have the extensions .txt or .log.

ObjectScript

```
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        GOTO SetEnvironment }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { GOTO SetEnvironment }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
SetEnvironment
  SET domId=domoref.Id
  IF ##class(%iKnow.Configuration).Exists("myconfig") {
        SET cfg=##class(%iKnow.Configuration).Open("myconfig") }
  ELSE { SET cfg=##class(%iKnow.Configuration).%New("myconfig",0,$LISTBUILD("en"),"",1)
        DO cfg.%Save() }
CreateListerAndLoader
  SET flister=##class(%iKnow.Source.File.Lister).%New(domId)
      DO flister.Init("myconfig","","","","")
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
UseListerAndLoader
  SET install=$SYSTEM.Util.InstallDirectory()
  SET dirpath=install_"mgr\Temp\iknow\mytextfiles"
  SET stat=flister.AddListToBatch(dirpath,$LB("txt","log"),0,"")
      WRITE "The lister status is ",$System.Status.DisplayError(stat),!
  SET stat=myloader.ProcessBatch()
      WRITE "The loader status is ",$System.Status.DisplayError(stat),!
QueryLoadedSources
  WRITE ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)," sources"
```

This example performs a batch load, appropriate for loading a large number of files. To load a small number of files use the **SetLister()** and **ProcessList()** methods.

6.3.2 Loading SQL Records

The following executable example performs a batch load of the records of the Cinema.Review table. It loads as a source text the ReviewText field value for each record. If there is an error in the SQL query, the Loader returns an error status.

iKnow programs that load SQL data must use the `%iKnow.Source.SQL.Lister`. This lister always invokes the `%iKnow.Source.Temp.Processor`, which takes no parameters. There is, therefore, no reason to specify the processor, unless you have created your own custom processor.

ObjectScript

```
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO SetEnvironment }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO SetEnvironment }
```

```
    ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
              QUIT}
SetEnvironment
  SET domId=domoref.Id
  IF ##class(%iKnow.Configuration).Exists("myconfig") {
        SET cfg=##class(%iKnow.Configuration).Open("myconfig") }
  ELSE { SET cfg=##class(%iKnow.Configuration).%New("myconfig",0,$LISTBUILD("en"),"",1)
        DO cfg.%Save() }
CreateListerAndLoader
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
      DO flister.Init("myconfig")
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT Top 25 ID AS UniqueVal,Type,NarrativeFull,EventDate FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
QueryLoadedSources
  WRITE ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)," sources loaded"
```

This example performs a batch load, appropriate for loading a large number of SQL records. To load a small number of SQL records use the **SetLister()** and **ProcessList()** methods.

You can also use the %SYSTEM.iKnow utility method **IndexTable()**.

6.3.3 Loading Elements of a Subscripted Global

The following executable example loads the elements of a subscripted global. It uses the `%iKnow.Source.Global.Lister` and specifies the following Lister parameters to the **ProcessList()** method: global name, first subscript (inclusive), and last subscript (inclusive). This example uses the ^Aviation.AircraftD global, found in the Samples namespace. Because this is a sparse array, only a few of the subscripts between 1 and 50,000 contain data:

ObjectScript

```
DomainCreateOrOpen
  ZNSPACE "Samples"
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        GOTO SetEnvironment }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { GOTO SetEnvironment }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
              QUIT}
SetEnvironment
  SET domId=domoref.Id
  IF ##class(%iKnow.Configuration).Exists("myconfig") {
        SET cfg=##class(%iKnow.Configuration).Open("myconfig") }
  ELSE { SET cfg=##class(%iKnow.Configuration).%New("myconfig",0,$LISTBUILD("en"),"",1)
        DO cfg.%Save() }
ListerAndLoader
  SET mylister=##class(%iKnow.Source.Global.Lister).%New(domId)
        DO mylister.Init("myconfig","","","","")
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
  SET stat=myloader.SetLister(mylister)
  IF stat '= 1 { WRITE "SetLister error ",$System.Status.DisplayError(stat)
                QUIT}
  SET gbl="^Aviation.AircraftD"
  SET stat=myloader.ProcessList(gbl,1,50000)
  IF stat '= 1 { WRITE "ProcessList error ",$System.Status.DisplayError(stat)
                QUIT }
SourceSentenceQueries
  SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
  WRITE "The domain contains ",numSrcD," sources",!
  SET numSentD=##class(%iKnow.Queries.SentenceAPI).GetCountByDomain(domId)
  WRITE "These sources contain ",numSentD," sentences"
```

The **ProcessList()** method can specify only one subscript level at a time. In order to iterate through multiple subscript levels, you must write code to invoke this method at the desired subscript level. For example, to load the second level subscripts 1 and 2, you would write code such as the following:

ObjectScript

```
FOR i=1:1:90000 {
   SET gbl="^Aviation.NarrativeS("_i_")"
   SET stat=myloader.ProcessList(gbl,1,2) }
```

This loads globals such as ^Aviation.NarrativeS(85879,1) and ^Aviation.NarrativeS(85879,2).

6.3.4 Loading a String

The following executable example loads a single global (or a string literal) as a source file. Note that no Lister is required when loading a string. You can specify the Configuration to apply in the **ProcessBuffer()** method.

ObjectScript

```
ConfigurationCreateOrOpen
  IF ##class(%iKnow.Configuration).Exists("EnFr") {
      SET cfg=##class(%iKnow.Configuration).Open("EnFr") }
  ELSE { SET cfg=##class(%iKnow.Configuration).%New("EnFr",1,$LB("en","fr"))
        DO cfg.%Save() }
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { WRITE "The ",dname," domain already exists",!
       SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
       GOTO DeleteOldData }
  ELSE
     { WRITE "The ",dname," domain does not exist",!
       SET domoref=##class(%iKnow.Domain).%New(dname)
       DO domoref.%Save()
       SET domId=domoref.Id
       WRITE "Created the ",dname," domain with domain ID ",domId,!
       GOTO CreateLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            SET domId=domoref.Id
            GOTO CreateLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
CreateLoader
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
UseLoader
  SET ^a="I drove at 70mph then sped up to 100mph when the light changed."
  DO myloader.BufferSource("ref",^a)
  DO myloader.ProcessBuffer("EnFr")
QuerySources
  WRITE "number of sources:",##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
```

The first argument of the **BufferSource()** method specifies a unique external source Id. The following example creates a separate source for each global subscript:

ObjectScript

```
  SET i=1
  WHILE $DATA(^a(i)) {
     DO myloader.BufferSource("ref"_i,^a(i))
     DO myloader.ProcessBuffer()
     SET i=i+1 }
  WRITE "end of data"
```

You can also use the %SYSTEM.iKnow utility method **IndexString()**.

6.4 Updating the Domain Contents

After you have performed an initial load of sources to a domain, you can change this list of sources by adding sources or by deleting sources. Updating a domain refers to responding to changes in the set of source texts. This should not be confused with upgrading a domain, which refers to responding to changes in the iKnow software, commonly after installing a significant new version of Caché.

6.4.1 Adding Sources

After you have performed an initial load of sources to a domain (using the **AddListToBatch()** and **ProcessBatch()** methods) you may want to add more files to the list of sources. This is done using the **SetLister()** and **ProcessList()** methods. The **ProcessList()** method takes the same parameters as the **AddListToBatch()** method.

- To add a one source at a time: SET stat=myloader.ProcessList("C:\mytextfiles\newfile.txt")

- To add a directory of sources: SET
  stat=myloader.ProcessList("C:\mytextfiles\logfiles",$LB("log"),0,"")

Adding more sources to a batch load is shown in the following example:

ObjectScript

```
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE { SET domoref=##class(%iKnow.Domain).%New(dname)
         DO domoref.%Save()
         GOTO SetEnvironment }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { GOTO SetEnvironment }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
SetEnvironment
  SET domId=domoref.Id
  IF ##class(%iKnow.Configuration).Exists("myconfig") {
        SET cfg=##class(%iKnow.Configuration).Open("myconfig") }
  ELSE { SET cfg=##class(%iKnow.Configuration).%New("myconfig",0,$LISTBUILD("en"),"",1)
        DO cfg.%Save() }
ListerAndLoader
  SET flister=##class(%iKnow.Source.File.Lister).%New(domId)
      DO flister.Init("myconfig","","","","")
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
  SET stat=myloader.SetLister(flister)
SourceBatchLoad
  SET install=$SYSTEM.Util.InstallDirectory()
  SET dirpath=install_"mgr\Temp\iknow\mytextfiles"
  SET stat=flister.AddListToBatch(dirpath,$LB("txt"),0,"")
  SET stat=myloader.ProcessBatch()
  IF stat '= 1 { WRITE "Loader error ",$System.Status.DisplayError(stat)
                    QUIT }
QueryLoadedSources
  WRITE "Source count is ",##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId),!
ExpandListofSources
  SET elister=##class(%iKnow.Source.File.Lister).%New(domId)
      DO elister.Init("myconfig")
  SET stat=myloader.SetLister(elister)
  SET addpath=install_"dev\Cache"
  SET stat=myloader.ProcessList(addpath,$LB("txt"),1,"")
  IF stat '= 1 { WRITE "The ProcessList loader status is ",$System.Status.DisplayError(stat)
               QUIT }
QueryTotalSources
  WRITE "Expanded source count is ",##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
```

You can also use the %SYSTEM.iKnow utility methods **IndexFile()** and **IndexDirectory()**.

6.4.2 Deleting Sources

You can remove a source that has been loaded to a domain using the **DeleteSource()** method. This method cannot be used to delete a virtual source; a separate **DeleteVirtualSource()** method is provided for this purpose. Both methods are found in the %SYSTEM.iKnow class.

6.5 Loading a Virtual Source

A virtual source is a source that is not static. You might, for example, use a virtual source for a file that is being frequently modified. The srcId of a virtual source is a negative integer. The external Id of a virtual source begins with the ListerReference (the Lister class alias), commonly :TEMP.

Adding a virtual source does not update iKnow statistics. For this reason, using a virtual source may be desirable when you wish to temporarily add sources for a specific purpose without incurring the overhead of revising the domain statistics. You should use a virtual source when adding a source that is being continuously modified, such a source in the process of being written. Because the virtual source Id is a negative number, it is easy to distinguish virtual sources from regular sources. Different methods are used to delete virtual sources and regular sources.

You can load virtual sources using **loader.SetLister()** and **loader.ProcessVirtualList()** or **loader.BufferSource()** and **loader.ProcessVirtualBuffer()**. The following program loads a virtual source using **ProcessVirtualBuffer()**.

ObjectScript

```
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { WRITE "The ",dname," domain already exists",!
       SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
       GOTO DeleteOldData }
  ELSE
     { WRITE "The ",dname," domain does not exist",!
       SET domoref=##class(%iKnow.Domain).%New(dname)
       DO domoref.%Save()
       SET domId=domoref.Id
       WRITE "Created the ",dname," domain with domain ID ",domId,!
       GOTO SetEnvironment }
DeleteOldData /* This DOES NOT delete virtual sources */
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            SET domId=domoref.Id
            GOTO SetEnvironment }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
SetEnvironment
  SET config="VSConfig"
  IF ##class(%iKnow.Configuration).Exists(config) {
        SET cfg=##class(%iKnow.Configuration).Open(config) }
  ELSE { SET cfg=##class(%iKnow.Configuration).%New(config,1)
         DO cfg.%Save() }
CreateLoader
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
VirtualSource
  SET node="",(total,status)=0
  FOR { SET node=$ORDER(^VendorData(node),1,data) QUIT:node=""
   SET company=$LIST(data,1) QUIT:company=""
   SET address=$LTS($LIST(data,2))
   SET total=total+1
   SET status=myloader.BufferSource("SourceTest"_total,company)
   SET status=myloader.BufferSource("SourceTest"_total,address)
  }
  SET status=myloader.ProcessVirtualBuffer(config)

  SET vsrclist=myloader.GetSourceIds()
  FOR i=1:1:$LL(vsrclist) {
     SET srcid=-$LIST(vsrclist,i)
     WRITE "External Id=",##class(%iKnow.Queries.SourceAPI).GetExternalId(domId,srcid)
     WRITE "  Source Id=",srcid,!
```

```
        WRITE "  Sentence Count=",##class(%iKnow.Queries.SentenceAPI).GetCountBySource(domId,$lb(srcid)),!
    }
```

Note that the **%iKnow.Queries.SourceAPI.GetCountByDomain()** method does not count virtual sources. You can determine if a virtual source has been loaded by invoking **%iKnow.Queries.SourceAPI.GetExternalId(domId,-1)**. Here -1 is the srcId of the first virtual source loaded.

By default, many iKnow queries process only ordinary sources and ignore virtual sources. To use these queries to process a virtual souce you must specify a vSrcId parameter value for the query method.

6.5.1 Deleting a Virtual Source

The %iKnow.Source.Loader class provides two methods for deleting virtual sources.

- **DeleteVirtualSource()** deletes a single virtual source indexed for a domain. You specify the domain Id (a positive integer) and the virtual source Id (a negative integer). This deletes all iKnow entities generated for this source text.

- **DeleteAllVirtualSources()** deletes all of the virtual sources indexed for a specified domain. This deletes all iKnow entities generated for these source texts.

6.6 Copying and Re-indexing Loaded Source Data

After you have successfully loaded sources into a domain, you may wish to copy some or all of these sources to another domain. When iKnow copies these loaded sources it also re-indexes them. The copied sources therefore have different source Ids and entity Ids; the external Ids are not changed.

Some reasons you might want to copy/re-index from one domain to another:

- To create a copy of a domain. You may wish to make a backup copy, or to create a copy to serve as a snapshot of the domain at a particular time. For example, when indexing RSS feeds you may wish to create a snapshot because these feeds change over time; at a future date you might no longer have access to the original source data.

- To create a domain containing a subset of the original set of sources. The new domain can be smaller, more efficient, and easier to work with. You can specify this copied subset of sources by a list of source Ids to copy, or by a filter that limits which sources to copy. For example, you could create a domain consisting of only the newest sources, which you could then query without having to filter by date for each query.

- To create a domain containing the merged sets of sources from two domains, or to add sources from one domain into a domain that already contains sources.

- To re-index the sources in a domain after extreme modification of the set of sources. For example, if you very frequently add or delete multiple sources in a domain, the indexing may no longer be optimal. (Normal adding and deleting of sources does not degrade index performance.) By copying the domain, you re-index the current sources that you are copying, making the indexing in the new domain optimal.

- To apply iKnow language model revisions. Release versions of iKnow commonly contain improvements to its language models. These may include introduction of support for new languages and improvements to already-supported languages. Copying the set of sources in a domain re-indexes these sources, and therefore applies the most current iKnow language models to the copied sources.

You use the %iKnow.Source.Domain.Lister class to copy/re-index from one domain to another. The new domain must already be defined before you can create a Lister instance for this class using the **%New()** method. Both domains must be in the same namespace.

The following example populates the firstdomain domain, then copies the contents of firstdomain to an empty domain named newdomain, automatically re-indexing the newdomain contents:

ObjectScript

```
EstablishAndPopulateFirstDomain
   SET domOref=##class(%iKnow.Domain).%New("firstdomain")
   DO domOref.%Save()
   SET domId=domOref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
  SET myquery="SELECT Top 25 ID AS UniqueVal,Type,NarrativeFull,EventDate FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
TestQueryFirstDomain
  WRITE ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)," sources in the from domain",!
CreateSecondDomain
   SET domOref=##class(%iKnow.Domain).%New("newdomain")
   DO domOref.%Save()
   SET domNewId=domOref.Id
CopyAndReindexFromFirstDomainToSecondDomain
  SET newlister=##class(%iKnow.Source.Domain.Lister).%New(domNewId)
  SET newloader=##class(%iKnow.Source.Loader).%New(domNewId)
  SET stat=newlister.AddListToBatch(domId)
  SET stat=newloader.ProcessBatch()
TestQuerySecondDomain
  WRITE ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domNewId)," sources in the to domain"
CleanUpForNextTime
  SET stat=##class(%iKnow.Domain).%DeleteId(domId)
  IF stat '= 1 {WRITE "Domain delete error:",stat }
  SET stat=##class(%iKnow.Domain).%DeleteId(domNewId)
  IF stat '= 1 {WRITE "Domain delete error:",stat }
```

The **AddListToBatch()** method can take a second lister parameter to specify which sources are to be copied. It can either specify a list of sources (a comma-separated list of source Id integers) or specify a filter. The following example is identical to the previous example, except that it limits which sources are to be copied by specifying a comma-separated list of source Ids.

ObjectScript

```
EstablishAndPopulateFirstDomain
   SET domOref=##class(%iKnow.Domain).%New("firstdomain")
   DO domOref.%Save()
   SET domId=domOref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
  SET myquery="SELECT Top 25 ID AS UniqueVal,Type,NarrativeFull,EventDate FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
TestQueryFirstDomain
  WRITE ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)," sources in the from domain",!
CreateSecondDomain
   SET domOref=##class(%iKnow.Domain).%New("newdomain")
   DO domOref.%Save()
   SET domNewId=domOref.Id
SubsetOfSourcesToCopy
  SET subset="1,3,5,7,9,11,13,15,17,19"
CopyAndReindexFromFirstDomainToSecondDomain
  SET newlister=##class(%iKnow.Source.Domain.Lister).%New(domNewId)
  SET newloader=##class(%iKnow.Source.Loader).%New(domNewId)
  SET stat=newlister.AddListToBatch(domId,subset)
  SET stat=newloader.ProcessBatch()
TestQuerySecondDomain
  WRITE ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domNewId)," sources in the to domain"
CleanUpForNextTime
  SET stat=##class(%iKnow.Domain).%DeleteId(domId)
```

```
IF stat '= 1 {WRITE "Domain delete error:",stat }
SET stat=##class(%iKnow.Domain).%DeleteId(domNewId)
IF stat '= 1 {WRITE "Domain delete error:",stat }
```

6.6.1 UserDictionary and Copied Sources

A UserDictionary is applied when a source is listed. Therefore, any UserDictionary modifications made to the initial loaded sources will appear in the copied sources. However, because the copy operation is also a list operation, you can also apply a new UserDictionary to modify the sources as they are copied.

For example, the UserDictionary used when the sources were originally listed substitutes "Doctor" for the abbreviation "Dr."; this substitution will be present in the copied sources. Later you modified the UserDictionary to also substitute "doctor" for "physician". This change to your UserDictionary had no effect on the already-loaded sources. When you copy the sources, you apply this revised UserDictionary. The "Dr." to "Doctor" substitution is performed 0 times, because that substitution is already present in the initial loaded sources; the "physician" to "doctor" substitution is performed on the copied sources.

7

Performance Considerations when Loading Texts

Because iKnow typically handles large amounts of text data, the following performance considerations should be heeded when loading source text:

- Before starting a batch load of a significant number of sources, stop database journaling. Once the batch load completes, make sure to restart journaling. Refer to the "Journaling" chapter of the *Caché Data Integrity Guide* for information on stopping and restarting journaling.

- Before starting a batch load of a significant number of sources (or a small number of very large sources), set the global buffer pool to a size large enough to handle this operation. iKnow indexing creates a large number of temporary globals. If the global buffer pool is not large enough to handle these temporary globals in memory, they are written to disk. These disk I/O operations can significantly affect iKnow performance. Refer to "Memory and Startup Settings" in the "Configuring Caché" chapter of the *Caché System Administration Guide*.

- iKnow indexing requires substantially more disk space than the space occupied by the source texts. The approximate space requirements for temporary and permanent globals are described in "Globals and Space Requirements" section of the "Implementation" chapter of this manual.

- Do not configure more language support than is required for your sources. Your iKnow Configuration should specify only those languages that are actually found in your sources. If all of your sources are in one language, do not specify automatic language identification. Unless n-grams are required for the language, do not set the EnableNgrams domain parameter.

8

iKnow Queries

The iKnow semantic analysis engine supplies a large number of query APIs which are used to return text entities and statistics about these text entities. For example, the **%iKnow.Queries.CrcAPI.GetTop()** method returns the most frequently occurring CRCs in a specified domain. The **%iKnow.Queries.CrcAPI.GetCountBySource()** returns the total number of unique CRCs that appear in the specified sources.

8.1 Types of Queries

There are three types of queries provided. They are distinguished by their name suffixes:

- API: ObjectScript queries
- QAPI: Caché SQL queries
- WSAPI: SOAP-accessible Web Services queries

For each of these types, iKnow provides queries for:

- Entities: return all entities in a source or multiple sources; the most frequently occurring entities; entities similar to a supplied string, etc.
- CCs: return concept-concept pairs.
- CRCs: return concept-relation-concept (master-relation-slave) sequences.
- Paths: return chain of concept-relation-concept sequences within a sentence. A path contains a minimum of two CRCs (CRCRC).
- Sentences: return sentences that contain a specified CRC, entity, etc.
- Sources: return sources that contain a specified CRC, entity, etc.

8.2 Queries Described in this Chapter

This chapter describes and provides examples of many commonly-used iKnow queries:

- Counting sources in a domain
- Counting sentences in a source

- Counting sources that contain a specified entity

- Listing "top" (most prominent) entities in a domain

- Listing CRCs that contain a specified entity (domain-wide)

- Counting sources that contain a specified CRC

- List sentences that fulfill a CRC mask (match entity value and position)

- Listing entities that are similar to a specified string (domain-wide)

- Listing entities that are related to a specified entity (domain-wide)

- Counting paths in a source

- Listing sources that are similar to a specified source (domain-wide)

- Summarizing a source (listing summary sentences)

Note that the query examples in this chapter use the default Configuration. Queries that you write may require a specified Configuration to establish the language environment.

8.3 Query Method Parameters

The following parameters are common to many query methods:

- domainid: The domain ID is an integer that identifies the domain.

- result: If a query returns an array of values rather than just a single result value, the result set is passed by reference (using the dot prefix operator, for example, .result). You can then use ZWRITE to display the whole result set in raw Caché list format, or use a loop structure and list-to-string conversion to return one row at a time, as shown in the examples in this chapter.

- page and pagesize (optional): To prevent methods from retrieving and returning thousands of records, the Query API uses a paging mechanism to allow the user to limit the number of results returned. It divides the results into equal-length pages, with the length of each page specified as the pagesize. For example, if you want the first ten results, you specify page 1 and a pagesize of 10. If you want the next page of results, you specify page 2 and pagesize 10. The default values are page=1 and pagesize=10.

- setop (optional): If a query applies more than one selection criteria, the Setop logical operator specifies whether the query should return the union or the intersection of the result sets. 1 ($$$UNION) returns results that match any of the supplied selection criteria. 2 ($$$INTERSECT) returns results that match all of the supplied selection criteria. The default is $$$UNION.

- entitylist: In queries that return matches to an entity (for example, **GetByEntities()**, **GetRelated()**, **GetSimilar()**) a Caché list of entities. You can specify entitylist entities in any mix of uppercase and lowercase letters; iKnow matches them against indexed entities normalized to lowercase.

- vSrcId: The source Id of a virtual source, specified as a negative integer. If specified, only entities in that virtual source are processed by the query. If omitted or specified as 0, only ordinary sources are considered by the query and virtual sources are ignored. The default is 0.

8.4 Counting Sources and Sentences

To count the number of sources loaded, you can use the **GetCountByDomain()** method of the %iKnow.Queries.SourceAPI class.

To count the sentences in all of the sources loaded, you can use the **GetCountByDomain()** method of the %iKnow.Queries.SentenceAPI class. To count the sentences in a single source, you can use the **GetCountBySource()** method.

The following example uses data loaded from .txt files (such as source1.txt, source2.txt, etc.) in the mytextfiles directory to demonstrate these sentence count methods. The default Configuration is used:

ObjectScript

```
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET mylister=##class(%iKnow.Source.File.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
  SET stat=myloader.SetLister(mylister)
  SET install=$SYSTEM.Util.InstallDirectory()
  SET dirpath=install_"mgr\Temp\iknow\mytextfiles"
  SET stat=myloader.ProcessList(dirpath,$LB("txt"),0,"")
  IF stat '= 1 { WRITE "Loader error ",$System.Status.DisplayError(stat)
                 QUIT }
SourceSentenceQueries
  SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
  WRITE "The domain contains ",numSrcD," sources",!
  SET numSentD=##class(%iKnow.Queries.SentenceAPI).GetCountByDomain(domId)
  WRITE "These sources contain ",numSentD," sentences",!!
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,20)
  SET i=1
  WHILE $DATA(result(i)) {
     SET extId = $LISTGET(result(i),2)
     SET fullref = $PIECE(extId,":",3,4)
     SET fname = $PIECE(fullref,"\",$LENGTH(extId,"\"))
     SET numSentS = ##class(%iKnow.Queries.SentenceAPI).GetCountBySource(domId,result(i))
     WRITE fname," has ",numSentS," sentences",!
     SET i=i+1 }
```

The following example uses data loaded from a field of the Aviation.Event SQL table to demonstrate these sentence count methods. In this example only a sample of 10 data records (TOP 10) are loaded:

ObjectScript

```
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
```

```
   IF stat { WRITE "Deleted the data from the ",dname," domain",!!
             GOTO ListerAndLoader }
   ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
             QUIT}
ListerAndLoader
   SET domId=domoref.Id
   SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
   SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT Top 10 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseLister
   SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
       IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
   SET stat=myloader.ProcessBatch()
       IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceSentenceQueries
   SET numSrcD=##class(%iKnow.Queries.SourceQAPI).GetCountByDomain(domId)
   WRITE "The domain contains ",numSrcD," sources",!
   SET numSentD=##class(%iKnow.Queries.SentenceQAPI).GetCountByDomain(domId)
   WRITE "These sources contain ",numSentD," sentences",!!
   DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,20)
   SET i=1
   WHILE $DATA(result(i)) {
       SET extId = $LISTGET(result(i),2)
       SET fullref = $PIECE(extId,":",3,4)
       SET fname = $PIECE(fullref,"\",$LENGTH(extId,"\"))
       SET numSentS = ##class(%iKnow.Queries.SentenceAPI).GetCountBySource(domId,result(i))
       WRITE fname," has ",numSentS," sentences",!
       SET i=i+1 }
```

For details on what iKnow considers a sentence, refer to the Logical Text Units Identified by iKnow section of the "Conceptual Overview" chapter.

8.5 Counting Entities

To count the number of sources that contain one or more occurrences of a specified entity, you can use the **GetCountByEntities()** method of the %iKnow.Queries.SourceAPI class. In this method you can specify a list on one or more entities to search for in the loaded sources.

Note that here, and throughout iKnow, the concept of "entity" differs significantly from the familiar notion of a search term. For example, the entity "dog" *does not occur* in the sentence "The quick brown fox jumped over the lazy dog." The entity "lazy dog" does occur in this sentence. An entity can be a concept or a relation; you could, for example, count the number of sources that contain the entity "is" or the entity "jumped over". However, in these examples and in most real-world cases, iKnow matches concepts or concepts associated by a relation.

The following example demonstrates these query count methods:

ObjectScript

```
#include %IKPublic
   ZNSPACE "Samples"
DomainCreateOrOpen
   SET dname="mydomain"
   IF (##class(%iKnow.Domain).NameIndexExists(dname))
       { WRITE "The ",dname," domain already exists",!
         SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
         GOTO DeleteOldData }
   ELSE
       { WRITE "The ",dname," domain does not exist",!
         SET domoref=##class(%iKnow.Domain).%New(dname)
         DO domoref.%Save()
         WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
         GOTO ListerAndLoader }
DeleteOldData
   SET stat=domoref.DropData()
   IF stat { WRITE "Deleted the data from the ",dname," domain",!!
             GOTO ListerAndLoader }
```

```
   ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
               QUIT}
ListerAndLoader
   SET domId=domoref.Id
   SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
   SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
    SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
    SET idfld="UniqueVal"
    SET grpfld="Type"
    SET dataflds=$LB("NarrativeFull")
UseLister
   SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
       IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
   SET stat=myloader.ProcessBatch()
       IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceCountQuery
   SET numSrcD=##class(%iKnow.Queries.SourceQAPI).GetCountByDomain(domId)
   WRITE "The domain contains ",numSrcD," sources",!
SingleEntityCounts
   SET ent=$LB("NTSB","National Transportation Safety Board",
     "NTSB investigator-in-charge","NTSB oversight","NTSB's Materials Laboratory",
     "FAA","Federal Aviation Administration","FAA inspector")
   SET entcnt=$LISTLENGTH(ent)
   SET ptr=0
   FOR x=1:1:entcnt {
    SET stat=$LISTNEXT(ent,ptr,val)
    WRITE ##class(%iKnow.Queries.SourceAPI).GetCountByEntities(domId,val)," contain ",val,!
    }
    WRITE "end of listing"
```

8.6 Listing Top Entities

iKnow has three query methods you can use to return the "top" entities in the source documents of a domain:

- **GetTop()** list the most-frequently-occurring entities in descending order by frequency count (by default). It can also be used to list most-frequently-occurring entities by spread. It provides the frequency and spread for each entity. This is the most basic listing of top entities.

- **GetTopTFIDF()** lists the top entities using a frequency-based metric similar to the TFIDF score. It calculates this score by combining an entity's Term Frequency (TF) with its Inverse Document Frequency (IDF). The Term Frequency counts how often the entity appears in a single source. The Inverse Document Frequency is based on the inverse of the spread (also known as "document frequency") of an entity. It uses this IDF frequency to diminish the Term Frequency. Thus an entity that appears multiple times in a small percentage of the sources is given a high TFIDF score; an entity that appears multiple times in a large percentage of the sources is given a low TFIDF score.

- **GetTopBM25()** lists the top entities using a frequency-based metric similar to the Okapi BM25 algorithm, which combines an entity's Term Frequency with its Inverse Document Frequency (IDF), taking into account document length.

All three of these methods return top Concepts by default, but can be used to return top Relations. All three of these methods can apply a filter to limit the scope of sources used.

The **GetTop()** method ignores entities of less than three characters. The **GetTopTFIDF()** and **GetTopBM25()** methods can return 1-character and 2-character entities.

8.6.1 GetTop(): Most-Frequently-Occurring Entities

An iKnow query can return the most frequently occurring entities in the source documents in descending order of frequency or spread. Each entity is returned as a separate record in Caché list format.

The entity record format is as follows:

- The entity ID, a unique integer assigned by iKnow.

- The entity value, specified as a string.

- Frequency: an integer count of how many times the entity occurs in the source documents.

- Spread: an integer count of how many source documents contain the entity.

The following query returns the most frequent (top) entities in the sources loaded by this program. By default these are Concept entities. It sets the page (1) and pagesize (50) parameters to specify how many entities to return. It returns (at most) the top 50 entities. It uses the domain default *sorttype*, which is in descending order by frequency:

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
    SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
    SET idfld="UniqueVal"
    SET grpfld="Type"
    SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceCountQuery
  SET numSrcD=##class(%iKnow.Queries.SourceQAPI).GetCountByDomain(domId)
  WRITE "The domain contains ",numSrcD," sources",!
TopEntitiesQuery
  DO ##class(%iKnow.Queries.EntityAPI).GetTop(.result,domId,1,50)
  SET i=1
  WHILE $DATA(result(i)) {
      SET outstr = $LISTTOSTRING(result(i),",",1)
        SET entity = $PIECE(outstr,",",2)
        SET freq = $PIECE(outstr,",",3)
        SET spread = $PIECE(outstr,",",4)
      WRITE "[",entity,"] appears ",freq," times in ",spread," sources",!
      SET i=i+1 }
  WRITE "Printed the top ",i-1," entities"
```

The following **GetTop()** method returns the top entities by spread:

ObjectScript

```
  DO ##class(%iKnow.Queries.EntityAPI).GetTop(.result,domId,1,50,,,$$$SORTBYSPREAD)
```

8.6.2 GetTopTFIDF() and GetTopBM25()

These two methods return a list of top entities in descending order by a calculated score. By default these are Concept entities. Because they are using different algorithms to assign a score to an entity, the list of "top" entities may differ sig-

nificantly. For example, the following table shows the relative order of four entities in the Aviation.Event database when analyzed using different methods:

	"airplane"	"helicopter"	"flight instructor"	"student pilot"
GetTop()	1st	12th	17th	43rd
GetTopTFIDF()	(not in listing)	1st	4th	22nd
GetTopBM25()	(not in listing)	3rd	2nd	1st

The top 5 entities in the Aviation.Event database returned by **GetTop()** are: "airplane", "pilot", "engine", "flight", and "accident". All of these entities occur at least once in more than half of the sources. While these are frequently-occurring entities, they are of little value in determining the contents of specific sources. An entity that occurs in more than half of the sources is given a negative IDF value. For this reason, none of these entities appear in the **GetTopTFIDF()** and **GetTopBM25()** listings.

The following example list the top 50 entities using **GetTopTFIDF()**:

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { WRITE "The ",dname," domain already exists",!
       SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
       GOTO DeleteOldData }
  ELSE
     { WRITE "The ",dname," domain does not exist",!
       SET domoref=##class(%iKnow.Domain).%New(dname)
       DO domoref.%Save()
       WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
       GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
  SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
     IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
     IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceCountQuery
  SET numSrcD=##class(%iKnow.Queries.SourceQAPI).GetCountByDomain(domId)
  WRITE "The domain contains ",numSrcD," sources",!
TopEntitiesQuery
  DO ##class(%iKnow.Queries.EntityAPI).GetTopTFIDF(.result,domId,1,50)
  SET i=1
  WHILE $DATA(result(i)) {
      SET outstr = $LISTTOSTRING(result(i),",",1)
        SET entity = $PIECE(outstr,",",2)
        SET score = $PIECE(outstr,",",3)
      WRITE "[",entity,"] has a TFIDF score of ",score,!
      SET i=i+1 }
  WRITE "Printed the top ",i-1," entities"
```

The following example list the top 50 entities using **GetTopBM25()**:

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
  SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceCountQuery
  SET numSrcD=##class(%iKnow.Queries.SourceQAPI).GetCountByDomain(domId)
  WRITE "The domain contains ",numSrcD," sources",!
TopEntitiesQuery
  DO ##class(%iKnow.Queries.EntityAPI).GetTopBM25(.result,domId,1,50)
  SET i=1
  WHILE $DATA(result(i)) {
      SET outstr = $LISTTOSTRING(result(i),",",1)
        SET entity = $PIECE(outstr,",",2)
        SET score = $PIECE(outstr,",",3)
      WRITE "[",entity,"] has a BM25 score of ",score,!
      SET i=i+1 }
  WRITE "Printed the top ",i-1," entities"
```

8.7 CRC Queries

An iKnow query that returns a CRC (Concept-Relation-Concept sequence) returns it in the following format:

• The CRC ID, a unique integer assigned by iKnow.

• The Master Concept, specified as a string.

• The Relation, specified as a string.

• The Slave Concept, specified as a string.

• Frequency: an integer count of how many times the CRC occurs in the source documents.

• Spread: an integer count of how many source documents contain the CRC.

8.7.1 Listing CRCs that Contain Entities

One common use of CRCs is to specify an entity (usually a Concept) and return the CRCs that contain that entity. This provides the various contexts in which an entity appears in a source (or sources). Because iKnow normalizes all text to lowercase letters, you must specify these matching entities in lowercase.

The following query returns all of the CRCs that contain the specified Concepts ("left wing", "right wing", "wings", "leading edge", and "trailing edge") as either the master concept or the slave concept of a CRC. Note that the **GetByEntities()** method *page* argument has been set to 25 to return more CRCs; it defaults to 10.

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { WRITE "The ",dname," domain already exists",!
       SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
       GOTO DeleteOldData }
  ELSE
     { WRITE "The ",dname," domain does not exist",!
       SET domoref=##class(%iKnow.Domain).%New(dname)
       DO domoref.%Save()
       WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
       GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
  SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
     IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
     IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
CRCQuery
  SET myconcepts=$LB("left wing","right wing","wings","leading edge","trailing edge")
  DO ##class(%iKnow.Queries.CrcAPI).GetByEntities(.result,domId,myconcepts,1,25)
  SET i=1
  WHILE $DATA(result(i)) {
     SET mycrcs=$LISTTOSTRING(result(i),",",1)
     WRITE "[",$PIECE(mycrcs,",",2,4),"]"
     WRITE "  appears ",$PIECE(mycrcs,",",5)," times in "
     WRITE $PIECE(mycrcs,",",6)," sources",!
     SET i=i+1 }
  WRITE !,"End of listing"
```

8.7.2 Counting Sources that Contain a CRC

The following program example returns the count of sources that contain the specified CRCs. To specify CRCs to the **GetCountByCrcs()** method, you must specify each CRC as a %List (using $LB), and then group these CRCs together as a %List. This is shown in the following example:

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
```

```
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
   ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
   SET stat=domoref.DropData()
   IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
   ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
   SET domId=domoref.Id
   SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
   SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseLister
   SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
   SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
CRCCount
   SET numSrcD=##class(%iKnow.Queries.SourceQAPI).GetCountByDomain(domId)
   SET mycrcs=$LB($LB("leading edge","of","wing"),$LB("leading edge","of","right wing"),
            $LB("leading edge","of","left wing"),$LB("leading edges","of","wings"),
            $LB("leading edges","of","both wings"))
   SET numSrc=##class(%iKnow.Queries.SourceAPI).GetCountByCrcs(domId,mycrcs)
   WRITE "From ",numSrcD," indexed sources there are ",!
   WRITE numSrc," sources containing one or more of the following CRCs:",!
   FOR i=1:1:$LISTLENGTH(mycrcs) {
       WRITE $LISTTOSTRING($LIST(mycrcs,i)," "),!
   }
```

The **GetCountByCrcs()** method returns the count of sources that contain any of the specified CRCs.

8.7.3 Listing Sources or Sentences that Fulfill a CRC Mask

You can use a CRC mask to specify an entity value for a specific CRC position. Each CRC has three positions: master, relation, and slave. With a CRC mask you can specify either an entity value or a wildcard for each position. A CRC mask enables you to list sources or sentences that contain CRCs that match one or more positional values. Because it specifies both position and entity value, the **GetByCrcMask()** partial CRC match is a more restrictive match than **GetByEntities()**, but a less restrictive match than **GetByCrcs()**.

The following example uses a CRC mask that matches the entity "student pilot" in master position, while using wildcards to permit any value in the CRC relation and slave positions. The **GetByCrcMask()** method matches this mask against every sentence in each source, and returns the sentence Id and the sentence text of those sentences that contain a CRC with "student pilot" in the master position.

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { WRITE "The ",dname," domain already exists",!
       SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
       GOTO DeleteOldData }
  ELSE
     { WRITE "The ",dname," domain does not exist",!
       SET domoref=##class(%iKnow.Domain).%New(dname)
       DO domoref.%Save()
       WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
       GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
```

```
   IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
   ELSE     { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
CRCMaskSentencesBySource
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,100)
  SET i=1
  WHILE $DATA(result(i)) {
      SET srcId = $LISTGET(result(i),1)
      SET extId = $LISTGET(result(i),2)
      SET srcname = $PIECE($PIECE(extId,":",3,4),"\",$LENGTH(extId,"\"))
      SET numSentS = ##class(%iKnow.Queries.SentenceAPI).GetCountBySource(domId,result(i))
      WRITE numSentS," sentences in ",srcname,!
      SET stat=##class(%iKnow.Queries.SentenceAPI).GetByCrcMask(.sentresult,domId,"student pilot",
            $$$WILDCARD,$$$WILDCARD,srcId)
      SET i=i+1
      FOR j=1:1:20 {
          IF $DATA(sentresult(j)) {
          SET sent = $LISTTOSTRING(sentresult(j),",",1)
          SET sentId = $PIECE(sent,",",3)
          WRITE "The SentenceId is ",sentId," in source ",srcname,":",!
          WRITE "  ",##class(%iKnow.Queries.SentenceAPI).GetValue(domId,sentId),!
          }
          ELSE { WRITE "Listed ",j-1," sentence that match the CRC mask",!!
          QUIT }
      }
  }
```

8.8 Listing Similar Entities

You can list the unique entities that are similar to a specified string. An entity is similar if one of the following applies:

- The string is identical to the entity.

- The string is one of the words of the entity.

- The string is the first letters of one of the words of the entity.

Similarity returns each unique entity (Master Concept or Slave Concept) with integer counts of its frequency and spread, in descending sort order of these integer counts. Similarity does not match Relations. As is true throughout iKnow, matching ignores letter case; all entities are returned in lowercase letters. Similarity does not use stemming logic; "cat" returns both "cats" and "category".

The following example lists the entities that are similar to the string "student pilot":

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
```

```
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
    SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
    SET idfld="UniqueVal"
    SET grpfld="Type"
    SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceCountQuery
  WRITE ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)," total sources",!!
SimilarEntityQuery
  WRITE "Entities similar to 'Student Pilot':",!
  DO ##class(%iKnow.Queries.EntityAPI).GetSimilar(.simresult,domId,"student pilot",1,50)
  SET j=1
  WHILE $DATA(simresult(j)) {
      SET outstr = $LISTTOSTRING(simresult(j),",",1)
        SET entity = $PIECE(outstr,",",2)
        SET freq = $PIECE(outstr,",",3)
        SET spread = $PIECE(outstr,",",4)
      WRITE "(",entity,")  appears ",freq," times in ",spread," sources",!
      SET j=j+1 }
```

The default domain parameter setting governing entity similarity is EnableNgrams, a boolean value.

8.8.1 Parts and N-grams

The **GetSimilar()** and **GetSimilarCounts()** methods have a mode parameter that specifies where to search for similarity. There are two available values:

- $$$USEPARTS causes iKnow to match the beginning of each part (word) for similarity. For texts in English and most other languages this is generally the preferred setting. $$$USEPARTS is the default.

- $$$USENGRAMS causes iKnow to match words and linguistic units within words (n-grams) for similarity. This mode is used when the source text language compounds words. For example, $$$USENGRAMS would commonly be used with German, a language which regularly forms compound words. $$$USENGRAMS would *not* be used with English, a language which does not compound words. $$$USENGRAMS can only be used in a domain that has the EnableNgrams domain parameter set.

8.9 Listing Related Entities

An entity is related to another entity if both occur in a CRC. By default, the related entity can be either a master concept or a slave concept. (Refer to "Limiting by Position" (below) to override this default.)

The following example shows how iKnow returns related entities. It first determines how many CRCs contain the entity "student pilot" and lists these CRCs. (In this small example, you can simply read all the CRCs to see what is related to "student pilot"; in a much larger collection of sources this would not be practical.) The program example then lists all of the entities that are related to "student pilot" as either slave or master (you can confirm these relations by matching these entities against the CRCs listed earlier):

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { WRITE "The ",dname," domain already exists",!
       SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
       GOTO DeleteOldData }
  ELSE
     { WRITE "The ",dname," domain does not exist",!
       SET domoref=##class(%iKnow.Domain).%New(dname)
       DO domoref.%Save()
       WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
       GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
     IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
     IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceCountQuery
  WRITE ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)," total sources",!!
ContainCRCQuery
  SET crccount = ##class(%iKnow.Queries.CrcAPI).GetCountByEntities(domId,$LB("student pilot"))
       WRITE crccount," CRCs contain 'student pilot'",!
  DO ##class(%iKnow.Queries.CrcAPI).GetByEntities(.result,domId,$LB("student pilot"),1,crccount)
  SET i=1
  WHILE $DATA(result(i)) {
    WRITE $LISTTOSTRING(result(i),",",1),!
    SET i=i+1 }
  SET relcount = ##class(%iKnow.Queries.EntityAPI).GetRelatedCount(domId,$LB("student pilot"))
  WRITE !,relcount," entities are related to 'student pilot':",!
RelatedEntityQuery
  DO ##class(%iKnow.Queries.EntityAPI).GetRelated(.rresult,domId,$LB("student pilot"),1,relcount)
  SET j=1
  WHILE $DATA(rresult(j)) {
     WRITE $LISTTOSTRING(rresult(j),",",1),!
    SET j=j+1 }
```

8.9.1 Limiting by Position

The position of an entity can be Master Concept, Relation, or Slave Concept. By default, the **GetRelated()** method returns all related concepts regardless of position and does not return relations. You can change this default by specifying a macro constant for the 8th parameter (positiontomatch). The available constants are as follows:

Constant	Value	Meaning
$$$USEPOSM	1	Master Concepts
$$$USEPOSR	2	Relations
$$$USEPOSMR	3	Master Concepts and Relations
$$$USEPOSS	4	Slave Concepts
$$$USEPOSMS (the default)	5	Master Concepts and Slave Concepts
$$$USEPOSRS	6	Relations and Slave Concepts
$$$USEPOSALL	7	Master Concepts, Relations, and Slave Concepts

The following example separates the related master concepts and the related slave concepts. (Note that $$$USEPOSM means that the supplied string is the master concept in the CRC, and the related entities are the slave concepts.)

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { WRITE "The ",dname," domain already exists",!
       SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
       GOTO DeleteOldData }
  ELSE
     { WRITE "The ",dname," domain does not exist",!
       SET domoref=##class(%iKnow.Domain).%New(dname)
       DO domoref.%Save()
       WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
       GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
  SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
     IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
     IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceCountQuery
  WRITE ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)," total sources",!!
ContainCRCQuery
  SET crccount = ##class(%iKnow.Queries.CrcAPI).GetCountByEntities(domId,$LB("student pilot"))
     WRITE crccount," CRCs contain 'student pilot'",!
  DO ##class(%iKnow.Queries.CrcAPI).GetByEntities(.result,domId,$LB("student pilot"),1,crccount)
  SET i=1
  WHILE $DATA(result(i)) {
    WRITE $LISTTOSTRING(result(i),",",1),!
    SET i=i+1 }
  SET relcount = ##class(%iKnow.Queries.EntityAPI).GetRelatedCount(domId,$LB("student pilot"))
  WRITE !,relcount," entities are related to 'student pilot':",!

ListRelatedMastersQuery
  DO ##class(%iKnow.Queries.EntityAPI).GetRelated(.mresult,domId,$LB("student
pilot"),1,relcount,"","",$$$USEPOSM)
  WRITE !,"The following have 'student pilot' as a master:",!
  SET j=1
```

```
      WHILE $DATA(mresult(j)) {
          WRITE $LISTTOSTRING(mresult(j),",",1),!
          SET j=j+1 }
ListRelatedSlavesQuery
   DO ##class(%iKnow.Queries.EntityAPI).GetRelated(.sresult,domId,$LB("student
pilot"),1,relcount,"","",$$$USEPOSS)
   WRITE !,"The following have 'student pilot' as a slave:",!
   SET k=1
   WHILE $DATA(sresult(k)) {
       WRITE $LISTTOSTRING(sresult(k),",",1),!
       SET k=k+1 }
```

8.10 Counting Paths

The following example shows the count of paths and the count of sentences for 50 sources. Commonly there are more paths than sentences in a source. However, it is possible that there may be more sentences than paths in some sources.

ObjectScript

```
#include %IKPublic
   ZNSPACE "Samples"
DomainCreateOrOpen
   SET dname="mydomain"
   IF (##class(%iKnow.Domain).NameIndexExists(dname))
       { WRITE "The ",dname," domain already exists",!
         SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
         GOTO DeleteOldData }
   ELSE
       { WRITE "The ",dname," domain does not exist",!
         SET domoref=##class(%iKnow.Domain).%New(dname)
         DO domoref.%Save()
         WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
         GOTO ListerAndLoader }
DeleteOldData
   SET stat=domoref.DropData()
   IF stat { WRITE "Deleted the data from the ",dname," domain",!!
             GOTO ListerAndLoader }
   ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
             QUIT}
ListerAndLoader
   SET domId=domoref.Id
   SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
   SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT TOP 50 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseLister
   SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
       IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
   SET stat=myloader.ProcessBatch()
       IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceCountQuery
   WRITE ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)," total sources",!!
PathCountBySource
   DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,50)
   SET i=1
   WHILE $DATA(result(i)) {
       SET srcId = $LISTGET(result(i),1)
       SET extId = $LISTGET(result(i),2)
       SET fullref = $PIECE(extId,":",3,4)
       SET fname = $PIECE(fullref,"\",$LENGTH(extId,"\"))
       SET numPathS = ##class(%iKnow.Queries.PathAPI).GetCountBySource(domId,$LB(srcId))
       SET numSentS = ##class(%iKnow.Queries.SentenceAPI).GetCountBySource(domId,result(i))
       WRITE numPathS," paths and ",numSentS," sentences in ",fname,!
       SET i=i+1 }
```

8.11 Listing Similar Sources

The iKnow semantic analysis engine can list which sources are similar to a specified source. Similarity between sources is determined by the number of entities that appear in both sources (the overlap), and the percentage of the source contents that contain overlap.

The **GetSimilar()** method can calculate similarity of sources to a specified source. Because of the potentially large number of similar sources, this method is commonly used with a filter to limit the set of sources considered. **GetSimilar()** can use your choice of two algorithms, each of which takes an algorithm parameter:

- Basic similarity of items ($$$SIMSRCSIMPLE, the default). Available algorithm parameters are "ent" (entity similarity, the default), "crc" (Concept-Relation-Concept sequence), or "cc" (Concept + Concept pair).

- Using semantic dominance calculations ($$$SIMSRCDOMENTS). The algorithm parameter is a boolean flag that specifies limiting similarity to sources that contain a dominant entity that is also a dominant entity in the specified source.

For each similar source, iKnow returns a list of elements with the following format:

```
srcId,extId,percentageMatched,percentageNew,nbOfEntsInRefSrc,nbOfEntsInCommon,nbOfEntsInSimSrc,score
```

srcId	The source ID, an integer assigned by iKnow.
extId	The external ID for the source, a string value.
percentageMatched	The percentage of the contents of the source that is the same as the match source.
percentageNew	The percentage of the contents of the source that is new. New contents are those that do not match with the match source.
nbOfEntsInRefSrc	The number of unique entities in the source being referenced (matched against this source).
nbOfEntsInCommon	The number of unique entities that are found in both sources.
nbOfEntsInSimSrc	The number of unique entities in this source.
score	The similarity score, expressed as a fractional number. An identical source would have a similarity score of 1.

The following example demonstrates the listing of similar sources. It first limits the set of test sources to those that may describe an engine failure incident, by using **GetByEntities()** to select for a list of appropriate entities. It then uses **GetSimilar()** to find sources similar to these test sources, which may indicate a pattern of similar incidents. **GetSimilar()** takes the default similarity algorithm ($$$SIMSRCSIMPLE) and its default algorithm parameter ("ent"). The program displays only those similar sources with a high similarity score (>.33). The similarity display omits the source external IDs:

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { WRITE "The ",dname," domain already exists",!
       SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
       GOTO DeleteOldData }
  ELSE
     { WRITE "The ",dname," domain does not exist",!
       SET domoref=##class(%iKnow.Domain).%New(dname)
```

```
            DO domoref.%Save()
            WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
            GOTO ListerAndLoader }
DeleteOldData
   SET stat=domoref.DropData()
   IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
   ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
   SET domId=domoref.Id
   SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
   SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseLister
   SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
   SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceCountQuery
   SET totsrc = ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
   WRITE totsrc," total sources",!
SimiarSourcesQuery
   SET engineents = $LB("engine","engine failure","engine power","loss of
power","carburetor","crankshaft","piston")
   DO ##class(%iKnow.Queries.SourceAPI).GetByEntities(.result,domId,engineents,1,totsrc)
   SET i=1
   WHILE $DATA(result(i)) {
      SET src = $LISTTOSTRING(result(i),",",1)
      SET srcId = $PIECE(src,",",1)
      WRITE "Source ",srcId," contains an engine incident",!
     DO ##class(%iKnow.Queries.SourceAPI).GetSimilar(.sim,domId,srcId,1,50,"",$$$SIMSRCSIMPLE,$LB("ent"))

      SET j=1
      WHILE $DATA(sim(j)) {
         SET simlist=$LISTTOSTRING(sim(j))
         IF $PIECE(simlist,",",8) > .33 {
            WRITE "   similar to source ",$PIECE(simlist,",",1),": "
            WRITE $PIECE(simlist,",",3,8),! }
         SET j=j+1 }
   SET i=i+1 }
```

8.12 Summarizing a Source

The iKnow semantic analysis engine can summarize a source text by returning the most relevant sentences. It returns a user-specified number of sentences in the original sentence order, selecting those sentences that have the highest similarity to the overall content of the source text. iKnow determines relevance by calculating an internal relevancy score for each sentence. Sentences that contain concepts that appear many times in the source text are more likely to be included in the summary than those that contain concepts that only appear once in the source text. iKnow considers the overall frequency of each concept, the similarity of each concept to the most frequent concepts in the source, and other factors.

Summarizing a source is only available if the Summarize property was set to 1 in the Configuration when loading the source. The default Configuration specifies Summarize=1.

The accuracy of a summary therefore depends on two factors:

- The source text must be large enough to permit meaningful frequency analysis, but not too large. iKnow summarization works best on texts the length of a chapter or article. A book-length text should be summarized chapter-by-chapter.

- The number of sentences in the summary should be a large enough subset of the original for the returned sentences to form a readable summary text. The minimum summary percentage is between 25% and 33%, depending on the contents of the text.

iKnow provides three summary methods:

- **GetSummary()** which returns each sentence of the summary text as a separate result. The sentence Id is returned as the first element of each returned sentence.

- **GetSummaryDirect()** which returns the summary text as a single string. By default, the sentences within this string are separated by an ellipsis: a space followed by three periods, followed by a space. For example, "This is sentence one. ... This is sentence two." You may specify a different sentence separator, if desired. Because this method concatenates multiple sentences into a single string, it may attempt to create a string longer than the Caché maximum string length. When the maximum string length is reached, Caché sets this method's isTruncated boolean output parameter to 1, and truncates the remaining text.

- **GetSummaryForText()** which supports compiling a summary of a user-supplied string directly, rather than by supplying a specific source ID.

For details on what iKnow considers a sentence, refer to the Logical Text Units Identified by iKnow section of the "Conceptual Overview" chapter.

The following example goes through the source texts in a domain until it finds one that contains more than 100 sentences. It then uses **GetSummary()** to summarize that source to half of its original sentences:

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { WRITE "The ",dname," domain already exists",!
       SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
       GOTO DeleteOldData }
  ELSE
     { WRITE "The ",dname," domain does not exist",!
       SET domoref=##class(%iKnow.Domain).%New(dname)
       DO domoref.%Save()
       WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
       GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
  SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
     IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
     IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceSentenceTotals
  SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
  WRITE "The domain contains ",numSrcD," sources",!
  SET numSentD=##class(%iKnow.Queries.SentenceAPI).GetCountByDomain(domId)
  WRITE "These sources contain ",numSentD," sentences",!!
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId)
SentenceCounts
  FOR i=1:1:numSrcD {
     SET srcId = $LISTGET(result(i),1)
     SET extId = $LISTGET(result(i),2)
     SET fullref = $PIECE(extId,":",3,4)
     SET fname = $PIECE(fullref,"\",$LENGTH(extId,"\"))
     SET numSentS = ##class(%iKnow.Queries.SentenceAPI).GetCountBySource(domId,result(i))
       IF numSentS > 100 {WRITE fname," has ",numSentS," sentences",!
                          GOTO SummarizeASource }
     }
     QUIT
SummarizeASource
  SET sumlen=$NUMBER(numSentS/2,0)
```

```
   WRITE "total sentences=",numSentS," summary=",sumlen," sentences",!!
   DO ##class(%iKnow.Queries.SourceAPI).GetSummary(.sumresult,domId,srcId,sumlen)
   FOR j=1:1:sumlen { WRITE "[S",j,"]: ",$LISTGET(sumresult(j),2),! }
   WRITE !,"END OF ",fname," SUMMARY",!!
   QUIT
```

Note that $NUMBER is used to assure that the specified summary sentence count is an integer. $LISTGET is used to remove the sentence Id and return just the sentence text.

The following example uses **GetSummaryDirect()** to return the same summary as a single concatenated string. It then uses $EXTRACT to divide the string into 38-character lines for display purposes:

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { WRITE "The ",dname," domain already exists",!
       SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
       GOTO DeleteOldData }
  ELSE
     { WRITE "The ",dname," domain does not exist",!
       SET domoref=##class(%iKnow.Domain).%New(dname)
       DO domoref.%Save()
       WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
       GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
  SET datafields=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,datafields)
     IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
     IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceSentenceTotals
  SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
  WRITE "The domain contains ",numSrcD," sources",!
  SET numSentD=##class(%iKnow.Queries.SentenceAPI).GetCountByDomain(domId)
  WRITE "These sources contain ",numSentD," sentences",!!
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId)
SentenceCounts
  FOR i=1:1:numSrcD {
     SET srcId = $LISTGET(result(i),1)
     SET extId = $LISTGET(result(i),2)
     SET fullref = $PIECE(extId,":",3,4)
     SET fname = $PIECE(fullref,"\",$LENGTH(extId,"\"))
     SET numSentS = ##class(%iKnow.Queries.SentenceAPI).GetCountBySource(domId,result(i))
       IF numSentS > 100 {WRITE fname," has ",numSentS," sentences",!
                          GOTO SummarizeASource }
  }
     QUIT
SummarizeASource
  SET sumlen=$NUMBER(numSentS/2,0)
  WRITE "total sentences=",numSentS," summary=",sumlen," sentences",!!
  SET summary = ##class(%iKnow.Queries.SourceAPI).GetSummaryDirect(domId,srcId,sumlen)
FormatSummaryDisplay
  SET x=1
  SET totlines=$LENGTH(summary)/38
  FOR i=1:1:totlines {
     WRITE $EXTRACT(summary,x,x+38),!
     SET x=x+39 }
  WRITE !,"END OF ",fname," SUMMARY"
```

8.12.1 Custom Summaries

iKnow permits you to generate custom summaries for sources by specifying a summaryConfig parameter string. Custom summaries are provided for those who desire to tune the content of iKnow generated summaries to their specific needs. Custom summaries allow you to absolutely include, preferentially include, or absolutely exclude sentences into the summary. You can, for example, include or exclude standard components of sources that always appear at the same location, such as a title, byline, copyright, abstract, or summary. You can also absolutely or preferentially include or exclude sentences that contain a specified word.

The source summarization operation first gives each sentence a numeric summary weight, and then creates the summary by selecting the appropriate number of sentences with the highest weights. You can influence this ranking by specifying a summaryConfig parameter to the summary method.

The summaryConfig parameter value is a string consisting of one or more specifications. Each specification consists of three elements separated by vertical bars. For example, `"s|2|false"`. You can concatenate multiple specifications using a vertical bar. For example, `"s|1|true|s|2|false"`. The summaryConfig parameter default is the empty string.

You can configure the summary to select sentences according to the following:

- Always (or never) include a sentence.

  - By sentence number: `"s|1|true"` mean that sentence (`s`) number `1` is always included in the summary (`true`). For example, if the sources always contain a title, it can be beneficial to always include the first sentence (the title) in the summary. If the second line of each source is an author byline, and you never want this included in the summary, you can specify this as `"s|2|false"`. You can also specify that the last sentence should never be included in the summary: `"s|-1|false"`; this might be appropriate if all of the sources end with a transcription reference or journal citation. Sentences in a source are numbered forward from 1, or numbered backwards from the end of the source as -1, -2, and so forth.

  - By word: `"w|requirement|true"` mean that any sentence containing the word (`w`) `requirement` is always included in the summary (`true`). You can also exclude sentences containing a specific word. For example, `"w|foreign|false"` excludes all sentences that contain the word `foreign` from the summary. A "word" can be one or more whole words: it can consist of a string of multiple words separated by spaces; it cannot consist of a partial word string. Note that words are normalized, and thus must be specified in all lowercase letters.

- Give a sentence more summary weight. This increases the chances that the sentence will appear in the summary. Available weight values are the integers 0 through 9.

  - By sentence number: `"s|1|3"` mean that sentence (`s`) number `1` has its summary weight increased by a factor of `3`. For example, the title (first sentence) of sources should be included if it is somewhat descriptive of the contents, but not included if it is not directly descriptive (for example, a literary quotation). Sentences in a source are numbered forward from 1, or numbered backwards from the end of the source as -1, -2, and so forth.

  - By word: `"w|requirement|2"` mean that any sentence containing the word (`w`) `requirement` has its summary weight increased by a factor of `2`. A "word" can be one or more whole words: it can consist of a string of multiple words separated by spaces; it cannot consist of a partial word string. Note that words are normalized, and thus must be specified in all lowercase letters.

You can specify multiple summary customizations by concatenation. For example: `"s|1|true|s|2|false|w|surgery|3|w|hypnosis|false"` (always include the first sentence, never include the second sentence, increase the summary weight of all sentences containing the word "surgery", exclude all sentences containing the word "hypnosis".

Thus the user can give more or less importance to specific words and/or sentences. The weight of sentences affected by more than one of the specifications in the summaryConfig will be resolved by the Custom Summaries algorithm. This algorithm also applies when there is a conflict between specifications that apply to the same sentence:

- If there is a conflict between a sentence (`s`) specification and a word (`w`) specification, the sentence specification wins.

- If there is a conflict between an include (`true`) and an exclude (`false`) involving two `s` specifications, or two `w` specifications, the include specification wins.

- If there is a conflict between the specified summary length and the number of sentences that must be included or number of sentences that must be excluded, the summary length is ignored.

The options for custom summaries can be set by means of the summaryConfig parameter in the **%iKnow.Queries.SourceAPI.GetSummary()** and **%iKnow.Queries.SourceAPI.GetSummaryForText()** methods.

8.13 Querying a Subset of the Sources

iKnow provides filters that allow you to include or exclude sources from a query. You can include or exclude sources based on:

- Random sampling of sources.
- Source contents: specified entities or entities that match a dictionary.
- Source characteristics (metadata): including the source Id, sentence count, and the source's indexed date (the date the source was loaded into iKnow).
- User-defined metadata characteristics of the sources.

iKnow supports the combining of multiple filters through logical AND and logical OR operators. For further details, refer to the Filtering chapter of this manual.

9

Semantic Attributes

An iKnow attribute flag is associated with one or more terms (word or short phrase) that affects the interpretation of a path or sentence. A part of a path or sentence is flagged as being affected by that attribute, and can thus be separated from similar parts of paths or sentences that do not have the attribute.

iKnow supports two attributes:

- Negation: A negation attribute identifies a part of a sentence as having negation. For example, the words "no" and "not" negate the meaning of a section of their path or sentence. iKnow language models identify many common negation terms. You can specify additional negation terms using the iKnow UserDictionary.

- Sentiment: A sentiment attribute flags a sentence as having either a positive or negative sentiment. For example, the words "avoid", "harm", "reject" typically convey a negative sentiment in many (but not all) contexts; the words "approve", "accept", "beneficial" convey a positive sentiment. Because sentiment terms are highly dependent on the kind of texts being analyzed, iKnow does not automatically identify sentiment terms. You can specify a list of positive and negative sentiment terms using the iKnow UserDictionary. These terms are then identified in the texts by the iKnow engine, which determines which parts of the sentence or path is affected by it.

9.1 Negation

Negation is the process that turns an affirmative sentence (or part of a sentence) into its opposite, a denial. For example, the sentence "I am a doctor." can be negated as "I am not a doctor." In analyzing text it is often important to separate affirmative statements about a topic from negative statements about that topic.

iKnow provides a means to determine if a sentence or path is negated. During source indexing iKnow associates the attribute "negation" with a sentence and indicates which part of the text is negated.

While in its simplest form negation is simply associating "no" or "not" with an affirmative statement or phrase, in actual language usage negation is a more complex and language-specific operation. There are two basic types of negation:

- Formal, or grammatical, negation is always indicated by a specific morphological element in the text. For example, "no", "not", "don't" and other specific negating terms. These negating elements can be part of a concept "He has no knowledge of" or part of a relation "He doesn't know anything about". Formal negation is always binary: a sentence (or part of a sentence) either contains a negating element (and is thus negation), or it is affirmative.

- Semantic negation is a complex, context-dependent form of negation that is not indicated by any specific morphological element in the text. Semantic negation depends upon the specific meaning of a word or word group in a specific context, or results from a specific combination of meaning and tense (for example, conjunctive and subjunctive tenses in Romance languages). For example, "Fred would have been ready if he had stayed awake" and "Fred would have been

ready if the need had arisen" say opposite things about Fred's readiness. Semantic negation is not a binary principle; it is almost never absolute, but is subject to contextual and cultural insights.

The iKnow language models contain a variety of language-specific negation words and structures. Using these language models the iKnow analysis engine is able to automatically identify and flag for future use most instances of formal negation as part of the source loading operation. However, iKnow cannot identify instances of semantic negation.

The largest unit of negation in iKnow is a path; iKnow cannot identify negations in text units larger than a path. Many, but not all, sentences comprise a single path.

9.1.1 Properties of Formal Negation

Formal negation can be defined by three properties:

- Negation markers: formal negation is always marked by one or more negation markers. These negation markers can be part of a concept or a relation. Some examples of negation markers in English are no, not, doesn't, isn't, hasn't, neither, nor, never, nothing, none, nobody, nowhere. iKnow always identifies a negation marker as part of a concept or part of a relation.

- Negation span: negation is always expressed in the broader context of a statement or a sentence. The effect of formal negation is that the statement or sentence (or some part of it) is negated. Therefore, it is important to determine the span of the negation, the part of the sentence that is made negative by the negation marker(s). The maximum span of a formal negation is a full sentence.

- Negation stopper: in many cases the span of the negation is not a full sentence. The span of the negation is terminated by a negation stopper, such as the words "but" and "or". iKnow identifies negation stoppers and uses this information to limit negation span.

iKnow uses these properties to identify negated units of text. Negation markers are tagged at the entity (concept or relation) level by assigning a negation attribute. Negation span is tagged at the path level with negation-begin and negation-end tags.

Japanese supports the negation attribute at the entity level, but because of the fundamentally different definition of paths in Japanese, path expansion is not supported. Therefore, negation for Japanese does not necessarily expand to all affected entities at the path level.

9.1.2 Using Negation Attributes

Negation analysis information can be used with the following methods:

- %iKnow.Queries.SourceAPI: **GetAttributes()**

- %iKnow.Queries.EntityAPI: **GetOccurrenceAttributes()** and **IsAttributed()**

- %iKnow.Queries.PathAPI: **GetAttributes()**

- %iKnow.Queries.SentenceAPI: **GetAttributes()** and **GetHighlighted()**

You can specify the negation attribute ID using the $$$IKATTNEGATION macro, defined in the %IKPublic #include file.

9.1.3 Negation Attribute Structure

Negation is implemented in iKnow as an attribute. That is, sources, sentences, or paths that contain negation have the negation attribute. This attribute is a %List structure with the following elements:

- Element 2 is the word "negation"

- Element 3 is the entity position that contains the first negation marker. A negation marker can be part of a relation or a concept. For example, in "The White Rabbit usually hasn't any time." the negation marker is in entity 3, the relation

"usually hasn't". In "The White Rabbit usually has no time." the negation marker is in entity 4, the concept "no time". Note that for this position count non-relevant words (such as "the" and "a") are counted as separate entities.

- Element 4 is the scope of the negation as a count of entities. Negation scope is counted from the first entity containing a negation marker to the last entity containing a negation marker. For example, "The man is neither fat nor thin" has a negation scope of 3 entities: "is neither/fat/nor".

- Element 5 shows the position of the negation marker within the entity as a bit map. A "1" indicates a word that is a negation marker; a "0" indicates a word that is not a negation marker. A negation marker consisting of two adjacent words, such as "is not", is indicated as "11". Entity mapping stops when the negation marker has been indicated. For example, the relation "is often not" is "001", while the relation "often is not" is "011", and the relation "is not often" is "11".

9.1.4 Negation Bit Map

Element 5 is the negation bit map. It indicates where the negation markers are in the negation scope. When the negation scope is 1, this is a simple bit map. When the negation scope is greater than one, this is a series of bit maps separated by spaces, one bit map for each entity within the negation scope.

Within the negation scope, if an entity contains a negation marker the negation marker and each word preceding it is indicated by either a 1 (negation marker word) or a 0 (word preceding the negation marker). If an entity within the negation scope does not contain a negation marker, the whole entity is represented by a single 0. Note that non-relevant words, such as "a" and "the", are considered to be separate entities. Some examples of negation bit mapping are shown in the following table:

Negation Bit Map	Sentence Text with / entity dividers and underlined negation markers
01 0 1	Bartleby / is neither / busy / nor idle.
01 0 1	Bartleby / is neither / sixty-five years old / nor retired.
1 0 1	Bartleby / is / no idler / and certainly is / no loafer.
11 0 0 01	Bartleby / is not / my / favorite fictional character / but neither is / he / my / least favorite.
1 0 0 01	Bartleby / isn't / my / favorite fictional character / but neither is / he / my / least favorite.
001 0 0011	Bartleby / is either not / trying very hard / or he is not / succeeding.
11 0 0 0011	Bartleby / is not / a / wholly realistic character / and yet is not / wholly unbelievable.
1 0001	Bartleby / never works, / but he is never / wholly idle.
1 001	Bartleby / does / nothing / and yet never is / he / idle.

The largest entity bit map is 8 bits. In rare cases a negation marker can be more than eight words from the beginning of its entity. If the negation marker is a two-word marker at positions 8 and 9, the second "1" is omitted ("00000001"); if the negation marker is at position 9 or greater, no bit map is returned. In the following examples the negation marker is in the second entity, a relation containing many words (due to the semantic ambiguity of the word "in"): "They start when you get in and are not finished when you leave." maps as "00000011"; "They start when you get in and they are not finished when you leave." maps as "00000001" (second word of the negation marker not mapped); "They often start when you get in and they are not finished when you leave." returns no bit map.

You can determine if a negation bit map has been omitted by comparing the Element 4 scope of negation entity count with the Element 5 number of blank-separated bit maps. If these two counts do not match one or more negation entity bit maps are missing.

9.1.5 Negation and Dictionary Matching

iKnow recognizing negated entities when matching against a dictionary. It calculates the number of entities that are part of a negation and stores this number as part of the match-level information (as returned by methods such as **GetMatchesBySource()** or as the NegatedEntityCount property of %iKnow.Objects.DictionaryMatch). This allows you to create code that interprets matching results by considering negation content, for example by comparing negated entities to the total number of entities matched.

For further details, refer to the Smart Matching: Using a Dictionary chapter of this manual.

9.1.6 Negation Examples

The following example uses **%iKnow.Queries.SourceAPI.GetAttributes()** to search each source in a domain for paths and sentences that have the negation attribute. It displays the PathId or SentenceId, the start position and the span of each negation. To limit **%iKnow.Queries.SourceAPI.GetAttributes()** to paths, specify $$$IKATTLVLPATH rather than $$$IKATTLVLANY:

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeCause FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeCause")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
     IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
     IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
GetSourcesAndAttributes
   SET numSrcD=##class(%iKnow.Queries.SourceQAPI).GetCountByDomain(domId)
   DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.srcs,domId,1,numSrcD)
   SET i=1
   WHILE $DATA(srcs(i)) {
      SET srcId = $LISTGET(srcs(i),1)
      SET i=i+1
      DO ##class(%iKnow.Queries.SourceAPI).GetAttributes(.att,domId,srcId,1,10,"",$$$IKATTLVLANY)
      SET j=1
      WHILE $DATA(att(j)) {
          IF $LISTGET(att(j),1)=1 {
            SET type=$LISTGET(att(j),2)
            SET level=$LISTGET(att(j),3)
```

```
              SET targId=$LISTGET(att(j),4)
              SET start=$LISTGET(att(j),5)
              SET span=$LISTGET(att(j),6)
                  IF level=1 {WRITE "source ",srcId," ",type," path ",targId," start at ",start," span
",span,!}
                    ELSEIF level=2 {WRITE "source ",srcId," ",type," sentence ",targId," start at ",start,"
 span ",span,!!}
                      ELSE {WRITE "unexpected attribute level",! }
              }
      SET j=j+1
      }
    }
```

The following example uses **%iKnow.Queries.SentenceAPI.GetAttributes()** to find those sentences in each source in a domain that have the negation attribute. It displays which sentence id of those sentences that have this attribute, and the entity position that contains the negation marker. It then displays the text of these sentences.

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeCause FROM Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
  SET dataflds=$LB("NarrativeCause")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
GetSourcesAndSentences
  SET numSrcD=##class(%iKnow.Queries.SourceQAPI).GetCountByDomain(domId)
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.srcs,domId,1,numSrcD)
  SET i=1
  WHILE $DATA(srcs(i)) {
      SET srcId = $LISTGET(srcs(i),1)
      SET i=i+1
      SET st = ##class(%iKnow.Queries.SentenceAPI).GetBySource(.sent,domId,srcId)
      SET j=1
      WHILE $DATA(sent(j)) {
         SET sentId=$LISTGET(sent(j),1)
         SET text=$LISTGET(sent(j),2)
         SET j=j+1
CheckSentencesForNegation
         SET atstat=##class(%iKnow.Queries.SentenceAPI).GetAttributes(.att,domId,sentId)
         SET k=1
            WHILE $DATA(att(k)) {
             WRITE "sentence ",sentId," has attribute=",$LISTGET(att(k),2)
             WRITE ", marker at entity position=",$LISTGET(att(k),3),!
             /* Format for display */
             WRITE sentId,": "
             SET x=1
             SET totlines=$LENGTH(text)/60
               FOR L=1:1:totlines {
               WRITE $EXTRACT(text,x,x+60),!
               SET x=x+61 }
             WRITE "END OF SENTENCE ",sentId,!!
```

```
        SET k=k+1 }
    }
}
```

9.1.7 Adding Negation Terms

You can specify negation for other specific words or phrases using the iKnow UserDictionary. Using the **AddNegationTerm()** method, you can add a list of negation terms to a UserDictionary. When source texts are loaded into a domain, all appearances of these terms are flagged with the negation marker.

9.1.8 Negation Special Cases

The following are a few peculiarities of negation in English:

- No.: The word "No." (with capital letter and period, quoted or not quoted) in English is treated as an abbreviation. It is not treated as negation and is not treated as the end of a sentence. Lowercase "no." is treated as negation and as a sentence ending.

- Nor: The word "Nor" at the beginning of a sentence is not marked as negation. Within the body of a sentence the word "nor" is marked as negation.

- No-one: The hyphenated word "no-one" is treated as a negation marker. Other hyphenated forms (for example, "no-where") are not.

- False negatives: Because formal negation depends on words, not context, occasional cases of false negatives may inevitably arise. For example, the sentences "There was no answer" and "The answer was no" are both flagged as negation.

Because negation operates on sentence units, it is important to know what iKnow does (and does not) consider a sentence. For details on how iKnow identifies a sentence, refer to the Logical Text Units Identified by iKnow section of the "Conceptual Overview" chapter.

9.2 Sentiment

A sentiment attribute flags a sentence as having either a positive or negative sentiment. Sentiment terms are highly dependent on the kind of texts being analyzed. For example, in a customer perception survey context the following terms might be flagged with a sentiment attribute:

- The words "avoid", "terrible", "difficult", "hated" convey a negative sentiment.
- The words "attractive", "simple", "self-evident", "useful", "improved" convey a positive sentiment.

Because sentiment terms are often specific to the nature of the source texts, iKnow does not automatically identify sentiment terms. You can flag individual words as having a positive sentiment or a negative sentiment attribute. By default, no words have a sentiment attribute. You can specify a sentiment attribute for specific words using the iKnow UserDictionary. Using the **AddPositiveSentimentTerm()** and **AddNegativeSentimentTerm()** methods, you can add a list of sentiment terms to a UserDictionary. When source texts are loaded into a domain, each appearance of these terms and the part of the sentence affected by it is flagged with the specified positive or negative sentiment marker.

For example, if "hated" is specified as having a negative sentiment attribute, and "amazing" is specified as having a positive sentiment attribute, when iKnow applies them to the sentence:

I hated the rain outside, but the running shoes were amazing.

Negative sentiment would affect "rain" and positive sentiment would affect "running shoes".

Sentiment attributes are supported for the following languages: English, German, Portuguese, Russian, and Ukrainian. Sentiment attributes are not currently supported for Japanese.

9.2.1 Using Sentiment Attributes

Sentiment Analysis information can be used with the following methods:

- %iKnow.Queries.SourceAPI: **GetAttributes()**

- %iKnow.Queries.EntityAPI: **GetOccurrenceAttributes()** and **IsAttributed()**

- %iKnow.Queries.PathAPI: **GetAttributes()**

- %iKnow.Queries.SentenceAPI: **GetAttributes()** and **GetHighlighted()**

You can specify a sentiment attribute ID using either the $$$IKATTSENPOSITIVE or $$$IKATTSENNEGATIVE macro, defined in the %IKPublic #include file.

10
Stemming

iKnow supports stemming, an optional feature when performing iKnow indexing of sources. Stemming allows iKnow to recognize the stem form of a word that has several grammatical forms. For example, the stem of a noun is typically its nominative singular form. The stem of a verb is typically its infinitive form. Stemming occurs at the entity level. This means that iKnow establishes the stem form of an entity, rather than the stem forms of the entity's individual words. It is an additional level of normalization from the original source text.

Note: Stemming support significantly increases iKnow indexing space requirements and impacts iKnow indexing performance. It is recommended that you only activate stemming for a domain when there is a real need for this functionality. Stemming is recommended for use with Russian and Ukrainian text sources.

Stemming is not performed on path-relevant entities.

Stemming is not supported for Japanese at this time.

Stemming may result in a stemmed entity that is no longer grammatically valid. For example, the stem for of the entity "two bananas" would be "two banana". Because the stem form of an entity may not actually exist in the indexed sources, iKnow pairs the stem form with a "representation form" which is the closest entity (as measured by the Levenshtein distance) to the stem form that actually exists as an indexed entity in the sources. The representative form can, of course, be identical to the stem form, if the stem form exists as an indexed entity in the sources.

iKnow uses a plug-in architecture for generating stems, enabling the use of third-party stemming tools, if available. iKnow uses Hunspell to generate stems. This means that when stemming is required, iKnow searches for the Hunspell affix (.aff) file for the specified language in the INSTALLDIR/dev/hunspell directory. If no Hunspell dictionary is provided for the specified language, iKnow searches for the %Text subclass for the specified language. Caché provides %Text datatype classes for five of the iKnow supported languages: %Text.English (en), %Text.French (fr), %Text.German (de), %Text.Portuguese (pt), and %Text.Spanish (es). If neither a Hunspell dictionary file nor a language-specific %Text class is available, iKnow uses the %Text.Text class. Refer to the %Text package class documentation in the *InterSystems Class Reference* for further details.

10.1 Configuring Stemming

Stemming can only be activated or deactivated for an empty domain. Once a domain is populated with source texts, you cannot change the stemming option for that domain, except by first removing all domain contents.

To activate stemming for an empty iKnow domain, set the $$$IKPSTEMMING domain parameter to 1. Stemming is inactive by default. To deactivate stemming for an empty iKnow domain, set $$$IKPSTEMMING to 0.

The domain must be empty when setting this parameter. When you add source texts to a domain where stemming is activated, iKnow creates a stem form for every entity and populates appropriate data structures so that stems can be queried.

Create an instance of the stemmer and specify the default language (or languages, using %iKnow.Stemming.MultiLanguage-Config). Specify languages using their ISO 639-1 two-character abbreviations.

10.1.1 Hunspell

Caché provide Hunspell files for Russian (ru) and Ukrainian (uk) in the dev/hunspell directory. You should place Hunspell dictionary files for other languages in the Caché dev/hunspell directory.

iKnow modifies the behavior of Hunspell stemming in one important respect. Hunspell stemming removes prefixes from words. iKnow restores prefixes removed by Hunspell before indexing the resulting stem forms.

If Hunspell returns more than one possible stem for a word, iKnow will attempt to disambiguate these options using the entity context to determine if the entity is a concept or a relation.

10.2 Stem Retrieval Methods

The following %iKnow.Queries.EntityAPI methods return stem values:

• **GetStem()** returns the stem string corresponding to a specified entity string.

• **GetStemId()** returns the stem ID for a specified stem string, if the stem string exists within the domain.

• **GetStemValue()** returns the stem string for a specified stem ID.

• **GetStemRepresentationForm()** returns the representation form string for a specified stem ID.

You can also return the frequency and spread for a specified stem ID.

10.3 Using Stems

Many iKnow query methods allow you to query on stems rather than on entities. By setting the method argument pUseStems=1, these query methods return values based on stem form values rather than exact entity values. The default, pUseStems=0 causes these query methods to return values based on exact entity values.

For example, the **GetTop()** method, by default, returns the most frequently occurring entities in a domain. If you specify pUseStems=1, this method returns the most frequently occurring stem forms in a domain, potentially merging together as a single stem form multiple entities that differ only in grammatical form.

If you set pUseStems=1 in a domain that does not support stemming, the method returns no results.

For further details, refer to the iKnow Queries chapter of this manual.

11

Blacklists

A Blacklist is a list of entities that you do *not* want a query to return. For example, if your source texts include greetings and salutations, you might want to put "many thanks", "best regards", and other stock phrases with no real information content in your Blacklist. A Blacklist might also be used to suppress top concepts that are too general and widespread to be of interest when analyzing query results.

Note: When displaying results of a query that applied a Blacklist, it is important to note that use of a Blacklist silently changes the query results by suppressing some information. Make sure the Blacklists used are relevant for the data contents, for the user looking at the query results, and for the context in which query results are displayed.

11.1 Creating a Blacklist

You can define a Blacklist that is assigned to a specific domain, or define a Blacklist that is domain-independent (cross-domain) and can be used by any domain in the current namespace. You can define a Blacklist in two ways:

- Using the Caché iKnow Architect. This interface can be used define a Blacklist within a domain, add and delete Blacklist entities, delete a Blacklist, or list all Blacklists defined for a domain. This interface supports populating a Blacklist by specifying entities as strings.

- Using the %iKnow.Utils.MaintenanceAPI class methods to define, populate, and maintain Blacklists. This class allows you to create both domain-specific and cross-domain Blacklists. This class provides methods for populating a Blacklist either by specifying entities as strings or by specifying entities by entity Id. Use of some of the Blacklist %iKnow.Utils.MaintenanceAPI class methods is shown in this chapter.

You can use the **CopyBlackLists()** method of the %iKnow.Utils.CopyUtils class to copy all defined Blacklists in a domain to another domain.

The Knowledge Portal and the Basic Portal user interfaces support the use of Blacklists.

The following example creates a domain-specific Blacklist and populates it with elements. It then lists all Blacklists for the domain, and all of the elements in this domain. Finally, it deletes the blacklist.

ObjectScript

```
DomainCreateOrOpen
  SET domn="mydomainwithbl"
  IF (##class(%iKnow.Domain).NameIndexExists(domn))
    { SET domo=##class(%iKnow.Domain).NameIndexOpen(domn)
      SET domId=domo.Id }
  ELSE { SET domo=##class(%iKnow.Domain).%New(domn)
        DO domo.%Save()
        SET domId=domo.Id }
```

```
CreateBlackList
  SET blname="AviationBlacklist"
  SET blId=##class(%iKnow.Utils.MaintenanceAPI).CreateBlackList(domId,blname,
      "Aviation non-mechanical terms Blacklist")
PopulateBlackList
  SET black=$LB("aircraft","airplane","flight","accident","event","incident","pilot",
          "student pilot","flight instructor","runway","accident site","ground","visibility","faa")

  SET ptr=0
  FOR x=0:1:100 {
   SET moredata=$LISTNEXT(black,ptr,val)
    IF moredata=1 {
      SET stat=##class(%iKnow.Utils.MaintenanceAPI).AddStringToBlackList(domId,blId,val)
    }
    ELSE { WRITE x," entities in Blacklist",!!
         GOTO ListBlacklist }
  }
ListBlacklist
  SET stat=##class(%iKnow.Utils.MaintenanceAPI).GetBlackLists(.bl,domId,0)
  SET i=1
  WHILE $DATA(bl(i)) {
      WRITE $LISTTOSTRING(bl(i),",",1),!
      SET i=i+1 }
  WRITE "Printed the ",i-1," Blacklists",!
  SET stat=##class(%iKnow.Utils.MaintenanceAPI).GetBlackListElements(.ble,domId,blId)
   /* IF stat=1 {WRITE "success",!}
  ELSE {WRITE "GetBlackListElements failed" QUIT } */
  SET j=1
  WHILE $DATA(ble(j)) {
      WRITE $LISTTOSTRING(ble(j),",",1),!
      SET j=j+1 }
  WRITE "Printed the ",j-1," Blacklist elements",!
CleanUp
  SET stat=##class(%iKnow.Utils.MaintenanceAPI).DropBlackList(domId,blId)
  IF stat=1 {WRITE "Blacklist deleted",!}
  ELSE {WRITE "DropBlackList failed" QUIT }
```

The **CreateBlackList()** method allows you to specify both a name and a description for your Blacklist. A Blacklist name can be any valid string of any length; Blacklist names are case-sensitive. The name you assign to a Blacklist must be unique: for a domain-specific Blacklist it must be unique within the domain; for a cross-domain Blacklist it must be unique within the namespace. Specifying a duplicate Blacklist name generates ERROR #8091. The Blacklist description is optional; it can be a string of any length.

11.1.1 Blacklists and Domains

Each Blacklist you create can either be specific to a domain, or can be cross-domain (domain-independent) and usable by any domain in the current namespace:

- A domain-specific Blacklist is assigned to a domain by specifying a domainId in the **CreateBlackList()** method. This method returns a blacklistId as a sequential positive integer. Query methods that use this Blacklist reference it by this blacklistId. A domain-specific Blacklist can support stemming.

- A cross-domain Blacklist is not assigned to a domain. Instead, you specify a domainId of 0 in the **CreateBlackList()** method. This method returns a blacklistId as a sequential positive integer. Query methods that use this Blacklist reference it by a negative blacklistId; for example, the Blacklist identified by blacklistId 8 is referenced by the blacklistId value -8.

  Note: Cross-domain Blacklists are only available to domains with version 4 (or higher). Domains created prior to Caché 2014.1 must be upgraded to version 4 to use cross-domain Blacklists.

To populate a domain-specific Blacklist, you can use either **AddEntityToBlackList()** or **AddStringToBlackList()**. To populate a cross-domain Blacklist, you can only use **AddStringToBlackList()**.

GetBlackListElements() returns the empty string for the entUniId value for a cross-domain Blacklist.

The following example creates and populates two Blacklists, a domain-specific blacklist (AviationTermsBlacklist) and a cross-domain blacklist (JobTitleBlacklist). The **GetBlackLists()** method returns both blacklists, because the pIncludeCross-

Domain boolean is set to 1. Note that **GetBlackLists()** returns the blacklist Id for the cross-domain blacklist as a negative integer.

ObjectScript

```
DomainCreateOrOpen
  SET domn="mydomainwithbl"
  IF (##class(%iKnow.Domain).NameIndexExists(domn))
      { SET domo=##class(%iKnow.Domain).NameIndexOpen(domn)
        SET domId=domo.Id }
  ELSE { SET domo=##class(%iKnow.Domain).%New(domn)
         DO domo.%Save()
         SET domId=domo.Id }
CreateBlackList1
  SET blname="AviationTermsBlacklist"
  SET blId=##class(%iKnow.Utils.MaintenanceAPI).CreateBlackList(domId,blname,
      "Common aviation terms Blacklist")
PopulateBlackList1
  SET black=$LB("aircraft","airplane","flight","accident","event","incident","airport","runway")
  SET ptr=0
  FOR x=0:1:100 {
   SET moredata=$LISTNEXT(black,ptr,val)
     IF moredata=1 {
       SET stat=##class(%iKnow.Utils.MaintenanceAPI).AddStringToBlackList(domId,blId,val)
     }
  }
  WRITE "Blacklist ",blname," populated",!
CreateBlackList2
  SET bl2name="JobTitleBlacklist"
  SET bl2Id=##class(%iKnow.Utils.MaintenanceAPI).CreateBlackList(0,bl2name,
      "Aviation personnel Blacklist")
PopulateBlackList2
  SET jobblack=$LB("pilot","copilot","student pilot","flight instructor","passenger")
  SET ptr=0
  FOR x=0:1:100 {
   SET moredata=$LISTNEXT(jobblack,ptr,val)
     IF moredata=1 {
       SET stat=##class(%iKnow.Utils.MaintenanceAPI).AddStringToBlackList(0,bl2Id,val)
     }
  }
  WRITE "Blacklist ",bl2name," populated",!!
ListBlacklists
  SET pIncludeCrossDomain=1
  SET stat=##class(%iKnow.Utils.MaintenanceAPI).GetBlackLists(.bl,domId,pIncludeCrossDomain)
  SET i=1
  WHILE $DATA(bl(i)) {
     IF $LIST(bl(i),1)<0 {
         WRITE "cross-domain:",!,$LISTTOSTRING(bl(i),",",1),! }
     ELSE { WRITE "domain-specific:",!,$LISTTOSTRING(bl(i),",",1),! }
  SET i=i+1 }
  WRITE "Printed the ",i-1," Blacklists",!!
CleanUp
  SET stat=##class(%iKnow.Utils.MaintenanceAPI).DropBlackList(domId,blId)
  IF stat=1 {WRITE "domain Blacklist deleted",!}
  ELSE {WRITE "first DropBlackList failed" }
  SET stat=##class(%iKnow.Utils.MaintenanceAPI).DropBlackList(0,bl2Id)
  IF stat=1 {WRITE "cross-domain Blacklist deleted",!}
  ELSE {WRITE "second DropBlackList failed" }
```

11.2 Queries that Support Blacklists

The following query methods provide a parameter to specify Blacklists. You can specify multiple Blacklists to any of these methods by specifying the Blacklist Ids as elements of a %List structure, using the $LISTBUILD function. You specify a domain-specific blacklist as a positive integer blacklistId value; you specify a cross-domain blacklist as a negative integer blacklistId value.

Entity Queries:

- **%iKnow.Queries.EntityAPI.GetByFilter()**

- **%iKnow.Queries.EntityAPI.GetBySource()**

- **%iKnow.Queries.EntityAPI.GetCountByDomain()**

- **%iKnow.Queries.EntityAPI.GetCountBySource()**

- **%iKnow.Queries.EntityAPI.GetNewBySource()**

- **%iKnow.Queries.EntityAPI.GetRelated()**

- **%iKnow.Queries.EntityAPI.GetRelatedById()**

- **%iKnow.Queries.EntityAPI.GetSimilar()**

- **%iKnow.Queries.EntityAPI.GetSimilarCounts()**

- **%iKnow.Queries.EntityAPI.GetTop()**

Sentence Queries:

- **%iKnow.Queries.SentenceAPI.GetNewBySource()**

Source Queries:

- **%iKnow.Queries.SourceAPI.GetSimilar**(): iKnow ignores blacklisted entities both when selecting similar entities and when calculating similarity scores.

11.2.1 Blacklist Query Example

The following example suppresses non-mechanical aviation terms that are too general to be of interest. It uses **CreateBlackList()** to create a Blacklist, uses **AddStringToBlackList()** to add entities to the Blacklist, then supplies the Blacklist to the **GetTop()** method:

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { WRITE "The ",dname," domain already exists",!
       SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
       GOTO DeleteOldData }
  ELSE
     { WRITE "The ",dname," domain does not exist",!
       SET domoref=##class(%iKnow.Domain).%New(dname)
       DO domoref.%Save()
       WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
       GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
CreateBlackList1
  SET blname="AviationTermsBlacklist"
  SET blId=##class(%iKnow.Utils.MaintenanceAPI).CreateBlackList(domId,blname,
       "Common aviation terms Blacklist")
PopulateBlackList
  SET black=$LB("aircraft","airplane","flight","accident","event","incident","pilot","airport",
            "student pilot","flight instructor","runway","accident site","ground","visibility","faa")

  SET ptr=0
  FOR x=0:1:100 {
   SET moredata=$LISTNEXT(black,ptr,val)
     IF moredata=1 {
       SET stat=##class(%iKnow.Utils.MaintenanceAPI).AddStringToBlackList(domId,blId,val)
     }
  }
  WRITE "Blacklist ",blname," populated",!
```

```
QueryBuild
    SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
    SET idfld="UniqueVal"
    SET grpfld="Type"
    SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceCountQuery
  SET numSrcD=##class(%iKnow.Queries.SourceQAPI).GetCountByDomain(domId)
  WRITE "The domain contains ",numSrcD," sources",!
TopEntitiesQuery
  DO ##class(%iKnow.Queries.EntityAPI).GetTop(.result,domId,1,20,"",0,0,0,0,$LB(blId))
  WRITE "NOTE: the ",blname," Blacklist",!,
        "has been applied to this list of top entities",!
  SET i=1
  WHILE $DATA(result(i)) {
      SET outstr = $LISTTOSTRING(result(i),",",1)
        SET entity = $PIECE(outstr,",",2)
        SET freq = $PIECE(outstr,",",3)
        SET spread = $PIECE(outstr,",",4)
      WRITE "[",entity,"] appears ",freq," times in ",spread," sources",!
      SET i=i+1 }
  WRITE "Printed the top ",i-1," entities"
```

12

Filtering Sources

You can use filters to include or exclude sources supplied to an iKnow query. Often you do not wish to perform a query on all of the loaded sources in the domain. A filter allows you to limit the scope of the query to only those sources that meet the criteria of the filter. A filter selects sources based either on which entities are found in the source, or on some information associated with the source itself (metadata). A filter always includes or excludes an entire source and is specified in each query that uses that filter.

12.1 Supported Filters

iKnow supplies a number of predefined filters, and provides facilities to allow users to easily define their own filters.

- Source Id: the SourceIdFilter and ExternalIdFilter allow you to select sources based on the Id of the source. iKnow assigns these values as part of the source indexing process.

- Random Sources: the RandomFilter allows you to select a random sample of the sources in a domain. You can specify the sample either as an integer number of sources or as a percentage of the total sources.

- Sentence Count: the SentenceCountFilter allows you to select sources based on the minimum and/or maximum number of sentences in the source. iKnow counts the sentences in a source as part of the indexing process.

- Entity Match: iKnow provides several filters that allow you to select sources based on the entities (concepts and relations) found in each source. The DictionaryMatchFilter allows you to filter sources based on the minimum and/or maximum number of dictionary matches to the contents of the source. (This filter replaces the deprecated SimpleMatchFilter.) The DictionaryTermMatchFilter and DictionaryItemMatchFilter allow you to filter sources using the same kind of dictionary matching, but limiting the match set to components of a dictionary, rather than the whole dictionary. These dictionary filters can optionally perform standardized-form matching if the $$$IKPMATSTANDARDIZEDFORM domain parameter is specified.

  The ContainsEntityFilter allows you to filter sources by supplying a list of entities directly (rather than by defining them in a dictionary). Optionally, ContainsEntityFilter can also filter sources by entities similar to the listed entities. The ContainsRelatedEntitiesFilter allows you to filter sources by supplying a list of entities, two or more of which must be related, meaning that they must appear in either the same path (the default) or the same CRC. Optionally, ContainsRelatedEntitiesFilter can also filter sources by related entities similar to the listed entities.

- Indexed Date Metadata: iKnow automatically provides one metadata field for every source, the DateIndexed field. iKnow assigns this field value as part of the source indexing process. By using the SimpleMetadataFilter, this field allows you to select sources based on the date and time when they were indexed by iKnow.

- User-Defined Metadata: iKnow allows you to define filters using the SimpleMetadataFilter that select sources based on data values that you have associated with a source.

• SQL Query: the SqlFilter allows you to select sources based on the results of an SQL query.

You can use GroupFilter to logically combine the results of the filters you define.

12.2 Filtering by the ID of the Source

The most basic source filter is used to limit the sources supplied to a query by providing the Source Id or the External Id of each source that you wish to include in the filtered result set.

12.2.1 By External Id

The ExternalIdFilter includes those sources whose external Ids are listed in a %List structure. Any element in this list that is not a valid External Id, or is a duplicate external Id is silently passed over.

The following example filters sources by external Id. The external Id for Aviation.Event sources includes either the word "Accident" or "Incident"; this filter include only the sources whose external Id includes the word "Incident". It then lists the details of the filtered sources:

ObjectScript

```
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE { SET domoref=##class(%iKnow.Domain).%New(dname)
         DO domoref.%Save()
         GOTO SetEnvironment }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { GOTO SetEnvironment }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
SetEnvironment
  SET domId=domoref.Id
  IF ##class(%iKnow.Configuration).Exists("myconfig") {
        SET cfg=##class(%iKnow.Configuration).Open("myconfig") }
  ELSE { SET cfg=##class(%iKnow.Configuration).%New("myconfig",0,$LISTBUILD("en"),"",1)
         DO cfg.%Save() }
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseListerAndLoader
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
DefineExtIdFilter
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,100)
  SET i=1
  SET extlist=$LB("")
  WHILE $DATA(result(i)) {
        SET extId = $LISTGET(result(i),2)
        IF $PIECE(extId,":",3)="Incident" {
        SET extlist=extlist_$LB(extId) }
        SET i=i+1
  }
  SET filt=##class(%iKnow.Filters.ExternalIdFilter).%New(domId,extlist)
SourceCountQuery
  SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
  WRITE "The ",dname," domain contains ",numSrcD," sources",!
ApplyExtIdFilter
  SET numSrcFD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,filt)
  WRITE "The Id filter includes ",numSrcFD," sources:",!
```

```
DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,100,filt)
    SET j=1
    WHILE $DATA(result(j)) {
       SET intId = $LISTGET(result(j),1)
       SET extId = $LISTGET(result(j),2)
       WRITE intId," ",extId,!
       SET j=j+1
    }
    WRITE "End of list"
```

12.2.2 By Source Id

The SourceIdFilter includes those sources whose source Ids are listed in a %List structure. Source Ids are integers. They can be listed in any order. Any element in this list that is not a valid source Id, or is a duplicate source Id is silently passed over.

The following example takes SQL records as sources and filters in several sources by Source Id. Source Ids in this data set are numbered 1 through 100. The SourceIdFilter specifies five source Ids for inclusion, but only three of these source Ids correspond to records in the table. Therefore, the total filtered source count is 3:

ObjectScript

```
DefineAFilter
  SET srclist=$LB(10,14,74,110,2799)
  SET filt=##class(%iKnow.Filters.SourceIdFilter).%New(domId,srclist)
SourceCounts
  SET numsrc = ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
    WRITE "The ",dname," domain contains ",numsrc," sources",!
 SET numfsrc = ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,filt)
    WRITE "Source count after source Id filtering: ",numfsrc
```

The iKnow.Filters.Filter class includes the %iKnow.Filters.SourceIdRangeFilter subclass. The following example filters a range of sources by source Id. It returns sources with source Ids 5 through 10, inclusive:

ObjectScript

```
SET filt=##class(%iKnow.Filters.SourceIdRangeFilter).%New(domId,5,10)
```

12.3 Filtering a Random Selection of Sources

You can use the %iKnow.Filters.RandomFilter to select a random sample of your sources. A random sample allows you to perform tests on a manageable subset of your sources. It also allows you to divide your sources (or a subset of them) into "training" and "test" sets. You would use the "training" set to define iKnow analytics (dictionary matches, source categories, etc.), then would use the "test" set to determine how well these analytics apply to another set of data. In this way you can avoid "overfitting" the analytics to a particular set of data.

You can specify the size of the random subset in two ways:

- As a percentage: You specify a percentage (as a fractional number between 0 and 1), and this filter returns the corresponding percentage of the indexed sources in the specified domain (or filtered subset of the domain). For example, a value of ".5" means that 50% of the sources in the domain will be included in the filtered result. Halves are rounded up, so 50% of 5 sources is 3 sources. You specify 100% as ".999" with the appropriate number of fractional digits. This filter selects the requisite number of sources randomly.

- As an integer: You specify an integer, and this filter returns that number of indexed sources in the specified domain (or filtered subset of the domain). For example, a value of "7" means that 7 of the sources in the domain will be included in the filtered result. This filter selects the specified number of sources randomly.

The following example randomly selects 33% of 50 sources, returning 17 sources. You can run this example repeatedly to demonstrate that different sources are randomly sampled:

ObjectScript

```
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE { SET domoref=##class(%iKnow.Domain).%New(dname)
         DO domoref.%Save()
         GOTO SetEnvironment }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { GOTO SetEnvironment }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
SetEnvironment
  SET domId=domoref.Id
  IF ##class(%iKnow.Configuration).Exists("myconfig") {
        SET cfg=##class(%iKnow.Configuration).Open("myconfig") }
  ELSE { SET cfg=##class(%iKnow.Configuration).%New("myconfig",0,$LISTBUILD("en"),"",1)
         DO cfg.%Save() }
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT TOP 50 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseListerAndLoader
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
     IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
  SET stat=myloader.ProcessBatch()
     IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
DefineAFilter
  SET filt=##class(%iKnow.Filters.RandomFilter).%New(domId,.33)
SampledSourceQueries
  SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
  WRITE "The ",dname," domain contains ",numSrcD," sources",!
  SET numSrcFD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,filt)
  WRITE "Of these ",numSrcD," sources ",numSrcFD," were sampled:",!
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,20,filt)
  SET i=1
  WHILE $DATA(result(i)) {
     SET intId = $LISTGET(result(i),1)
     SET extId = $LISTGET(result(i),2)
     WRITE "sample #",i," is source ",intId," ",extId,!
     SET i=i+1 }
     WRITE "End of list"
```

The following example filters the sources in the domain by source Id, returning 11 sources. It then supplies this source Id filter when defining a random filter. Thus, the random filter returns 3 of these source-Id-filtered sources. You can run this example repeatedly to demonstrate that different sources are randomly sampled:

ObjectScript

```
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE { SET domoref=##class(%iKnow.Domain).%New(dname)
         DO domoref.%Save()
         GOTO SetEnvironment }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { GOTO SetEnvironment }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
SetEnvironment
  SET domId=domoref.Id
  IF ##class(%iKnow.Configuration).Exists("myconfig") {
        SET cfg=##class(%iKnow.Configuration).Open("myconfig") }
  ELSE { SET cfg=##class(%iKnow.Configuration).%New("myconfig",0,$LISTBUILD("en"),"",1)
         DO cfg.%Save() }
```

```
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT TOP 50 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseListerAndLoader
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
     IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
  SET stat=myloader.ProcessBatch()
     IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
DefineSourceIdFilter
  SET srclist=$LB(1,3,5,7,9,11,13,15,17,21,23)
  SET idfilt=##class(%iKnow.Filters.SourceIdFilter).%New(domId,srclist)
SourceCounts
  SET numsrc = ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
     WRITE "The ",dname," domain contains ",numsrc," sources",!
 SET numfsrc = ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,idfilt)
     WRITE "Source count after source Id filtering: ",numfsrc,!
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,20,idfilt)
  SET i=1
  WHILE $DATA(result(i)) {
     SET intId = $LISTGET(result(i),1)
     WRITE intId," "
     SET i=i+1 }
     WRITE !,"End of list",!
DefineRandomFilter
  SET rfilt=##class(%iKnow.Filters.RandomFilter).%New(domId,3,idfilt)
RandomSample
  SET numrsrc=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,rfilt)
  WRITE "From ",numfsrc," sources ",numrsrc," are randomly sampled:",!
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,20,rfilt)
  SET j=1
  WHILE $DATA(result(j)) {
     SET intId = $LISTGET(result(j),1)
     WRITE intId," "
     SET j=j+1 }
     WRITE !,"End of list",!
```

12.4 Filtering by Number of Sentences

iKnow divides a source text into sentences. The following example filters out sources that contain less than 75 sentences. It returns the total number of sources, then uses the filter to return the total number of sources containing 75 or more sentences, then uses the filter again to return the number of sentences in each of these sources:

ObjectScript

```
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE { SET domoref=##class(%iKnow.Domain).%New(dname)
         DO domoref.%Save()
         GOTO SetEnvironment }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { GOTO SetEnvironment }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
SetEnvironment
  SET domId=domoref.Id
  IF ##class(%iKnow.Configuration).Exists("myconfig") {
        SET cfg=##class(%iKnow.Configuration).Open("myconfig") }
  ELSE { SET cfg=##class(%iKnow.Configuration).%New("myconfig",0,$LISTBUILD("en"),"",1)
         DO cfg.%Save() }
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT TOP 50 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
```

```
    SET grpfld="Type"
    SET dataflds=$LB("NarrativeFull")
UseListerAndLoader
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
DefineAFilter
  SET filt=##class(%iKnow.Filters.SentenceCountFilter).%New(domId)
  SET nsent=75
  DO filt.MinSentenceCountSet(nsent)
SourceSentenceQueries
  SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
  WRITE "The domain contains ",numSrcD," sources",!
  SET numSrcFD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,filt)
  WRITE "Of these ",numSrcD," sources ",numSrcFD," contain ",nsent," or more sentences:",!
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,50,filt)
  SET i=1
  WHILE $DATA(result(i)) {
      SET numSentS = ##class(%iKnow.Queries.SentenceAPI).GetCountBySource(domId,result(i))
      SET intId = $LISTGET(result(i),1)
      SET extId = $LISTGET(result(i),2)
      WRITE "source ",intId," ",extId
      WRITE " has ",numSentS," sentences",!
      SET i=i+1 }
      WRITE i-1," sources listed"
```

To filter using both minimum and maximum number of sentences, invoke both instance methods. The following filter selects those sources containing between 10 and 25 sentences (inclusive):

ObjectScript

```
MinMaxFilter
  SET min=10
  SET filt=##class(%iKnow.Filters.SentenceCountFilter).%New(domId)
  DO filt.MinSentenceCountSet(min)
  DO filt.MaxSentenceCountSet(min+15)
```

12.5 Filtering by Entity Match

The following entity filters are provided:

- ContainsEntityFilter: filters sources using a specified list of entities, at least one of which must appear in the source. Optionally, ContainsEntityFilter can also filter sources by entities similar to the listed entities.

- ContainsRelatedEntitiesFilter: filters sources using a specified list of entities, two or more of which must be related entities, meaning that they must appear in either the same path (the filter default) or the same CRC. Optionally, ContainsRelatedEntitiesFilter can also filter sources by related entities similar to the listed entities.

- DictionaryMatchFilter: filters sources using one or more dictionaries containing lists of entities; by default, at least one of these entities must appear in the source. Optionally, a specified minimum number of these entity matches must occur for a source to be selected. Dictionary matching also supports standardized-form matching, if the $$$IKPMAT-STANDARDIZEDFORM domain parameter is specified for the current domain.

12.5.1 Filtering by Dictionary Match

The %iKnow.Filters.DictionaryMatchFilter class allows you to select sources based on the contents of one or more user-defined dictionaries. iKnow also supports filtering by matching to a list of dictionary terms (%iKnow.Filters.DictionaryTermMatchFilter) or to a list of dictionary items (%iKnow.Filters.DictionaryItemMatchFilter).

In the following simple example, the second filter parameter specifies that only one dictionary is applied; multiple dictionaries can be specified as elements of a %List. The third parameter is set to the default of 1, which means a single match of any dictionary item selects the source for inclusion. You might wish to set this higher to avoid selecting sources where a single dictionary match may be coincidental rather than significant, due to either a large number of items in the dictionary

and/or querying sources that contain a large amount of text. The fourth parameter takes the default (-1), which puts no maximum limit on number of matches. If the max parameter is smaller than the min parameter, all sources are selected by the filter, regardless of dictionary matches. The fifth parameter also takes its default: matching based on count of matches, not match score. The sixth parameter is the ensureMatched flag; here ensureMatched=2, so that instantiating the filter generates static match results which are then used each time the filter in invoked. This is the preferred usage. You would need to set ensureMatched to 1 if a dictionary is modified after the filter is instantiated; ensureMatched=1 takes into account changing dictionary contents by matching before every invocation of the filter. However, use of ensureMatched=1 can result is significantly slower performance.

ObjectScript

```
DomainCreateOrOpen
   SET dname="mydomain"
   IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
   ELSE { SET domoref=##class(%iKnow.Domain).%New(dname)
          DO domoref.%Save()
          GOTO SetEnvironment }
DeleteOldData
   SET stat=domoref.DropData()
   IF stat { GOTO SetEnvironment }
   ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
             QUIT}
SetEnvironment
   SET domId=domoref.Id
ListerAndLoader
   SET domId=domoref.Id
   SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
   SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT TOP 50 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseListerAndLoader
   SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
   SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
CreateDictionary
   SET dictname="EngineTerms"
   SET dictdesc="A dictionary of aviation engine terms"
   SET dictId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionary(domId,dictname,dictdesc)
   IF dictId=-1 {WRITE "Dictionary ",dictname," already exists",!
                 GOTO ResetForNextTime }
   ELSE {WRITE "created dictionary ",dictId,!}
PopulateDictionaryItem1
    SET itemId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryItem(domId,dictId,
       "engine parts",domId_dictId_1)
      SET term1Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId,
         "piston")
      SET term2Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId,
         "cylinder")
      SET term2Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId,
         "crankshaft")
      SET term2Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId,
         "camshaft")
DefineAFilter
   SET filt=##class(%iKnow.Filters.DictionaryMatchFilter).%New(domId,$LB(dictId),1,,,2)
SourceCountQuery
   SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
   WRITE "The ",dname," domain contains ",numSrcD," sources",!
SourcesFilteredByDictionaryMatch
   SET numSrcFD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,filt)
   WRITE "Of these ",numSrcD,", ",numSrcFD," match the ",dictname," dictionary:",!!
   DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,50,filt)
   SET i=1
   WHILE $DATA(result(i)) {
      SET intId = $LISTGET(result(i),1)
      SET extId = $LISTGET(result(i),2)
      WRITE "dictionary matches ",intId," ",extId,!
      SET i=i+1 }
   WRITE !,i-1," sources included by dictionary match",!
ResetForNextTime
   IF dictId = -1 {
      SET dictId=##class(%iKnow.Matching.DictionaryAPI).GetDictionaryId(domId,dictname)}
```

```
SET stat=##class(%iKnow.Matching.DictionaryAPI).DropDictionary(domId,dictId)
IF stat {WRITE "deleted dictionary ",dictId,! }
ELSE    { WRITE "DropDictionary error ",$System.Status.DisplayError(stat) }
```

12.6 Filtering by Indexing Date Metadata

Every iKnow source is assigned the DateIndexed metadata field. The value of this field is the date and time that a source was indexed by iKnow, in Coordinated Universal Time format (UTC) represented in $HOROLOG format. This is the same as the $ZTIMESTAMP time, except that DateIndexed does not include fractional seconds.

You can create a filter using DateIndexed to include or exclude sources based on when iKnow loaded the source. You can filter using a specific date and time, or filter for a specific date, which encompass all time values within that date. You can use BETWEEN logic to filter for a range of dates.

The following example filters for sources loaded today:

ObjectScript

```
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE { SET domoref=##class(%iKnow.Domain).%New(dname)
         DO domoref.%Save()
         GOTO SetEnvironment }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { GOTO SetEnvironment }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
SetEnvironment
  SET domId=domoref.Id
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT TOP 50 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseListerAndLoader
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
DefineAFilter
  SET tday = $PIECE($ZTIMESTAMP,",",1)
  SET filt=##class(%iKnow.Filters.SimpleMetadataFilter).%New(domId,"DateIndexed","=",tday)
DateIndexedValue
  WRITE "Today is ",$PIECE($ZTIMESTAMP,",",1),!
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,50)
  SET i=1
  WHILE $DATA(result(i)) {
     SET srcId = $LISTGET(result(i),1)
     SET extId = $LISTGET(result(i),2)
     SET idate = ##class(%iKnow.Queries.MetadataAPI).GetValue(domId,"DateIndexed",extId)
     WRITE "Source ",srcId," was indexed ",idate,!
     SET i=i+1 }
SourceSentenceQueries
  SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
  WRITE "The ",dname," domain contains ",numSrcD," sources",!
  SET numSrcFD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,filt)
  WRITE "Of these sources ",numSrcFD," were indexed today"
```

12.7 Filtering by User-defined Metadata

In iKnow, *data* is the contents of a source that iKnow processes and indexes. In iKnow, *metadata* can be any data associated with a source that is not iKnow indexed data. You use iKnow metadata to identify iKnow data. A metadata filter uses the value of a metadata field to determine which sources to supply to a query.

Note: The iKnow definition of "metadata" describes how data is *used*, not the intrinsic nature of the data. This concept differs somewhat from the way this word is used elsewhere in Caché software.

iKnow provides a default metadata management system that is independent of the query APIs. The %iKnow.Queries.MetadataAPI class and accompanying %iKnow.Filters.SimpleMetadataFilter provide implementations for basic metadata filtering. If you wish to implement a custom Metadata API, you should implement (at least) the %iKnow.Queries.MetadataI interface and register your class as the "MetadataAPI" domain parameter: DO domain.SetParameter("MetadataAPI","Your.Metadata.Class"). The example that follows uses the %iKnow.Filters.SimpleMetadataFilter class.

In Caché SQL, each record of an SQL table constitutes an iKnow source. Through the **ProcessList()** method (for small numbers of records) or **AddListToBatch()** method (for large numbers of records), you define the Lister parameters:

*   You define the RowID field as a component of the iKnow external Id. iKnow also generates a source Id for each row as a unique integer; this iKnow source Id is completely independent of the RowId or other SQL identifier values.

*   You define a field (or fields) that contain a string of text as a data field to be indexed as iKnow data.

*   You define a field (or fields) as an iKnow metadata field. iKnow can use the values of this metadata field to select sources for an iKnow query.

Note that it is possible to specify the same field as both one of the data fields and as a metadata field. You can optionally also define metakey fields that correspond to the metadata fields.

This is shown in the following example. The Aviation.Event table contains various fields in addition to the NarrativeFull text field. In this example, InjuriesTotal is used as a metadata field. This metadata field is used in three filters: two equality filters, which filter for InjuriesTotal>2 and InjuriesTotal=3, and a BETWEEN filter that filters for InjuriesTotal between 3 and 5 (inclusive). This example uses the **DropData(1)** method, because **DropData()** with no argument does not delete metadata. Also note that the **AddField()** method must be invoked before listing and loading the data.

ObjectScript

```
#include %IKPublic
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE { SET domoref=##class(%iKnow.Domain).%New(dname)
         DO domoref.%Save()
         GOTO SetEnvironment }
DeleteOldData
  SET stat=domoref.DropData(1)
  IF stat { GOTO SetEnvironment }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
SetEnvironment
  SET domId=domoref.Id
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT TOP 100 ID AS UniqueVal,Type,NarrativeFull,InjuriesTotal,InjuriesTotalFatal FROM
 Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
  SET dataflds=$LB("NarrativeFull")
```

```
    SET metaflds=$LB("InjuriesTotal","InjuriesTotalFatal")
AddMetaFields
  SET val=##class(%iKnow.Queries.MetadataAPI).AddField(domId,"InjuriesTotal",
                   $LB("=","<",">","BETWEEN"),$$$MDDTNUMBER)
  SET val=##class(%iKnow.Queries.MetadataAPI).AddField(domId,"InjuriesTotalFatal",
                   $LB("=","<",">","BETWEEN"),$$$MDDTNUMBER)
UseListerAndLoader
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds,metaflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
CountSources
   SET numsrc=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
ApplyFilter
  SET filt2=##class(%iKnow.Filters.SimpleMetadataFilter).%New(domId,"InjuriesTotal",
                   ">",2)
  SET numSrcF2=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,filt2)
  WRITE "Of these ",numsrc," sources ",numSrcF2," had three or more injuries",!
  SET filt3=##class(%iKnow.Filters.SimpleMetadataFilter).%New(domId,"InjuriesTotal",
                   "=",3)
  SET numSrcF3=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,filt3)
  WRITE "Of these ",numsrc," sources ",numSrcF3," had three injuries",!
  SET filtb=##class(%iKnow.Filters.SimpleMetadataFilter).%New(domId,"InjuriesTotal",
                   "BETWEEN","3;5")
  SET numSrcFb=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,filtb)
  WRITE "Of these ",numsrc," sources ",numSrcFb," had between 3 and 5 injuries",!
```

12.7.1 Metadata Filter Operators

You assign to each filter one or more equality operators. If the filter is matching against a string value, use the "=" equality operator. If the filter is matching against a numeric value, you can use one or more of the following operators: "=", "<", "<=", ">", ">=". Equality operators are always specified as quoted string elements in a list structure. Equality operators are matched against a single value. This is shown in the following example:

ObjectScript

```
SET filt=##class(%iKnow.Filters.SimpleMetadataFilter).%New(domId,metafldname,"=",today)
```

The BETWEEN operator is matched against a parameter string containing a pair of values that are separated by $$$MDVALSEPARATOR (the semicolon character). This is shown in the following example:

ObjectScript

```
SET filt=##class(%iKnow.Filters.SimpleMetadataFilter).%New(domId,metafldname,
                 "BETWEEN","yesterday;tomorrow")
```

12.8 Filtering by SQL Query

The %iKnow.Filters.SqlFilter class allows you to select SQL sources based on the results of an SQL query. This query can select on any of the following fields:

- SourceId: the (internal) Source ID of the sources to be selected.

- ExternalId: the full External ID of the sources to be selected.

- IdField and GroupField: the two columns used together as identifiers when adding the sources to the domain: Local Reference (IdField) and Group Name (GroupField). See also %iKnow.Source.SQL.Lister.

Note that these result column names are case-sensitive.

For example, the following filter selects for the SourceId of the 6th source retrieved (in this case SourceId 45):

ObjectScript

```
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE { SET domoref=##class(%iKnow.Domain).%New(dname)
         DO domoref.%Save()
         GOTO SetEnvironment }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { GOTO SetEnvironment }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
SetEnvironment
  SET domId=domoref.Id
  IF ##class(%iKnow.Configuration).Exists("myconfig") {
        SET cfg=##class(%iKnow.Configuration).Open("myconfig") }
  ELSE { SET cfg=##class(%iKnow.Configuration).%New("myconfig",0,$LISTBUILD("en"),"",1)
         DO cfg.%Save() }
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT TOP 50 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseListerAndLoader
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,50)
FilterSources
  SET filter=##class(%iKnow.Filters.SqlFilter).%New(domId,
      "SELECT '"_$LIST(result(6),1)_"' AS SourceId")
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.fresult,domId,0,0,filter)
  WRITE !,"Filtered results:",!
  SET j=1
  WHILE $DATA(fresult(j)) {
    WRITE $LISTTOSTRING(fresult(j)),!
    SET j=j+1 }
```

The following filter selects sources by ExternalId:

ObjectScript

```
  SET filter=##class(%iKnow.Filters.SqlFilter).%New(domId,
        "SELECT '"_$LIST(result(1),2)_"' AS ExternalId")
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(domId,0,0,filter)
  ZWRITE result
```

12.9 Filter Modes

The FilterMode argument specifies what statistical reprocessing should be performed after applying a filter. If no reprocessing is performed, the filter is applied but the frequency and spread statistics are the values calculated for the item before filtering and the sort sequence remains unchanged. The following are the available filter modes:

FilterMode	Integer code	Filter?	Recalculate Frequency?	Recalculate Spread?	Re-sort Results?
$$$FILTERONLY	1	YES	NO	NO	NO
$$$FILTERFREQ	3	YES	YES	NO	NO
$$$FILTERSPREAD	5	YES	NO	YES	NO
$$$FILTERALL	7	YES	YES	YES	NO
$$$FILTERFREQANDSORT	11	YES	YES	NO	YES
$$$FILTERSPREADANDSORT	13	YES	NO	YES	YES
$$$FILTERALLANDSORT	15	YES	YES	YES	YES

The default is $$$FILTERONLY.

Refer to Constants in the "iKnow Implementation" chapter for use of $$$ macros.

12.10 Using GroupFilter to Combine Multiple Filters

iKnow supplies a GroupFilter class that allows you to supply logic to combine the results of other filters. You must first create a GroupFilter instance that provides a defined logic, and then use the **AddSubFilter()** method to assign it one or more subfilter objects that are combined according to the GroupFilter logic. In this way you can combine multiple existing filters to select which sources are supplied to an iKnow query.

The simplest GroupFilter logic returns the inverse of a single filter. In the following example, the RandFilt selects 33% of the sources. The GroupFilter defines AND logic with a Negated boolean operator. When applied to a single filter, this logic returns all of the sources *not* selected by the filter:

ObjectScript

```
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE { SET domoref=##class(%iKnow.Domain).%New(dname)
         DO domoref.%Save()
         GOTO SetEnvironment }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { GOTO SetEnvironment }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
SetEnvironment
  SET domId=domoref.Id
QueryBuild
   SET myquery="SELECT TOP 50 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
     IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
  SET stat=myloader.ProcessBatch()
     IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
DefineARandomFilter
  SET randfilt=##class(%iKnow.Filters.RandomFilter).%New(domId,.33)
SampledSourceQueries
  SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
  WRITE "The ",dname," domain contains ",numSrcD," sources",!
```

```
   SET numSrcFD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,randfilt)
   WRITE "From ",numSrcD," sources randfilt sampled ",numSrcFD,!
GroupFilter
    SET grpfilt=##class(%iKnow.Filters.GroupFilter).%New(domId,"AND",1)
    DO grpfilt.AddSubFilter(randfilt)
   SET numSrcGrp=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,grpfilt)
   WRITE "From ",numSrcD," sources grpfilt sampled ",numSrcGrp
```

The following example uses a GroupFilter to combine the results of two other filters (in this case, both are random filters). Because the GroupFilter logic is AND, and the Negated=0, the GroupFilter results are those sources that are found in both RandomFilter sets. Because the results of these filters are random, the number of GroupFilter AND results will likely differ each time this example is executed:

ObjectScript

```
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
         GOTO DeleteOldData }
  ELSE { SET domoref=##class(%iKnow.Domain).%New(dname)
          DO domoref.%Save()
          GOTO SetEnvironment }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { GOTO SetEnvironment }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
             QUIT}
SetEnvironment
  SET domId=domoref.Id
QueryBuild
    SET myquery="SELECT TOP 50 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
    SET idfld="UniqueVal"
    SET grpfld="Type"
    SET dataflds=$LB("NarrativeFull")
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
DefineTwoRandomFilters
  SET randfilt1=##class(%iKnow.Filters.RandomFilter).%New(domId,.33)
  SET randfilt2=##class(%iKnow.Filters.RandomFilter).%New(domId,.25)
  SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
  WRITE "The ",dname," domain contains ",numSrcD," sources",!
  SET numSrcFD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,randfilt1)
  WRITE "From ",numSrcD," sources randfilt1 sampled ",numSrcFD,!
  SET numSrcFD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,randfilt2)
  WRITE "From ",numSrcD," sources randfilt2 sampled ",numSrcFD,!
GroupFilter
    SET grpfilt=##class(%iKnow.Filters.GroupFilter).%New(domId,"AND",0)
    DO grpfilt.AddSubFilter(randfilt1)
    DO grpfilt.AddSubFilter(randfilt2)
  SET numSrcGrp=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,grpfilt)
  WRITE "From ",numSrcD," sources grpfilt sampled ",numSrcGrp
```

You assign each GroupFilter either the AND ($$$GROUPFILTERAND) or the OR ($$$GROUPFILTEROR) logical operator. Therefore, to create a compound filter involving both AND and OR logic, you must create a GroupFilter with AND logic and a GroupFilter with OR logic.

The following example corresponds to the Boolean expression "(filter1 AND !(filter2 OR filter3))":

ObjectScript

```
#include %IKPublic
    SET domoref=##class(%iKnow.Domain).%New("MyDomain")
    DO domoref.%Save()
    SET domId=domoref.Id
    /* . . . */
Create3Filters
    /* . . . */
GroupFilters
  SET group1=##class(%iKnow.Filters.GroupFilter).%New(domId,$$$GROUPFILTERAND,0)
  SET group2=##class(iKnow.Filters.GroupFilter).%New(domId,$$$GROUPFILTEROR,1)
  DO group1.AddSubFilter(filter1)
  DO group2.AddSubFilter(filter2)
  DO group2.AddSubFilter(filter3)
  DO group1.AddSubFilter(group2)
```

13

Text Categorization

Text categorization allows you to assign category labels to source texts, based on the contents of those texts.

For example, suppose you anticipate having a large number of Aviation Event texts that you will wish to categorize by AircraftCategory label: "airplane", "helicopter", "glider", and so forth. By determining what iKnow entities in the contents of these sample texts correspond strongly to a category label, you can create a text classification model that you can apply to future Aviation Event texts that do not yet have an assigned category.

Defining appropriate categories is an essential preliminary to text categorization:

- Each source can only be assigned one category. A source is assigned a category; a category is not assigned sources. Every source must correspond unambiguously to one of the defined categories.

- The number of category values (labels) is fixed. Because you cannot add more category labels to a text classification model, all possible future sources should be assignable to one of the initial category labels.

- The number of category values (labels) should be small. Categories should be designed so that roughly equal numbers of sources will be assigned to each category value.

13.1 Text Categorization Implementation

iKnow supports two approaches to building a text classification model:

- Analytic: analyze a set of existing texts that have category labels to determine which entities within the texts are the strongest indicators of membership in each of these categories. This requires you to have a representative sample of texts that have already been categorized.

- Rules-based: perform iKnow entity queries on a set of existing texts; for example, determine the top TFIDF or BM25 entities. Define categories using **%AddCategory**(). Develop rules (boolean tests) about the presence of high-value entities within the texts, associating specific categories with specific rules. Applying these rules will collectively determine the membership of a text in a category: highest number of successful boolean tests determines the category. This *does not* require you to have a set of texts that have already been categorized.

The descriptions that follow apply to the analytic approach to building a text classification model. Note that an analytic approach can use any analytic method, such as Naive Bayes statistical analysis or user-defined decision rules.

To perform text categorization, you must first create a Text Classifier (a text classification model). This model is based on a training set of source texts that have already been assigned category labels. By analyzing the contents of these training set texts, iKnow determines which iKnow entities correspond strongly to which category. You build and test a Text Classifier

that statistically associates these iKnow text entities with categories. Once you have an accurate Text Classifier, you can then use it on new source texts to assign category labels to them.

Typically, categories are specified in a metadata field. Each text is associated with a single category label. The number of different category labels should be low relative to the number of source texts, with each category well represented by a number of texts in your training set.

iKnow text categorization starts from iKnow entities (not just words) within the source texts. It can use in its analysis not only the frequency of entities within the source texts, but the context of the entity, such as whether the entity is negated, and the entity's appearance in larger text units such as CRCs and sentences. By using the full range of iKnow semantic analysis, text categorization can provide precise and valuable categorization of texts.

Analytic text categorization consists of three activities:

- Build a Text Classifier. This requires a set of texts (the training set), each of which has been assigned a category label. In this step you select a set of terms (entities) that are found in these texts and may serve to differentiate them. You also select a ClassificationMethod (algorithm) that determines how to correlate the appearance of these terms in the training set texts with the associated category label.

- Test the Text Classifier to determine its fit. This requires another set of texts (the test set), each of which has been assigned a category label. Based on this test information, you can revisit the build step, adding or removing terms, and thus iteratively improving the accuracy of the text classifier model.

- Use the Text Classifier to categorize texts that do not have an assigned category.

13.1.1 Implementation Interfaces

You can implement a text classification model in either of two ways:

- Using User Interface (UI) tools in the Caché Management Portal. From the Management Portal **System Explorer** option, select the **iKnow** option, then select **Text Categorization**. This displays two options: **Model builder** and **Model tester**.

- Programmatically, using the %iKnow.Classification package: %iKnow.Classification.Builder, %iKnow.Classification.Optimizer, and %iKnow.Classification.Classifier.

13.2 Establishing a Training Set and a Test Set

Regardless of which interface you use, before building a Text Classifier you must load into a domain a group of data sources with associated category labels. These sources are used to train and test the Text Classifier.

Note: It is possible to create a rules-based Text Classifier that does not require a pre-existing group of sources with assigned category labels. However, in the examples in this chapter the use of training set and test set sources is required.

You need to be able to divide these loaded sources into (at least) two groups of sources. A training set of sources, and a test set of sources. You use the training set to establish what entities are good indicators for particular categories. You use the test set (or multiple test sets) to determine if this predictive assignment of category labels makes sense with sources other than the training set. This prevents "overfitting" the terms to a particular group of sources. It is desirable that the training set be the larger of the two sets, containing roughly 70% of the sources, with the remaining 30% as the test set.

One common method for dividing SQL sources into a training set and a test set is to use a field of the source as a metadata field. You supply less than (<) and greater than (>) operators to **AddField()** so that you can perform a boolean test on the values of that field, dividing the sources into two groups. This division of sources should be as random as possible; using the SQL RowID as the metadata field usually achieves this goal.

The Management Portal **Text Categorization Model builder** is designed to use the values of a metadata field to divide a group of sources into a training set and a test set.

The following example establishes the SQL RowID as a metadata field that can be used to divide the loaded sources into a training set and a test set:

ObjectScript

```
SET myquery="SELECT ID,SkyConditionCeiling,Type,NarrativeFull FROM Aviation.Event"
SET idfld="ID"
SET grpfld="Type"
SET dataflds=$LB("NarrativeFull")   // text data field
SET metaflds=$LB("SkyConditionCeiling","ID")
DO ##class(%iKnow.Queries.MetadataAPI).AddField(domId,"SkyConditionCeiling")  // categories field
DO ##class(%iKnow.Queries.MetadataAPI).AddField(domId,"ID",$LB("=","<=",">")) // set divider field
```

You can also divide loaded sources of any type into groups using iKnow source ID values. You can use the %iKnow.Filters.SourceIdFilter class to divide a group of sources into a training set and a test set. The following example uses modulo division on the source IDs to place two-thirds of the loaded sources in tTrainingSet, and the remaining sources in tTestSet:

ObjectScript

```
FilterBySrcId
    SET numsrc = ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
    DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,numsrc)
      SET j=1
      SET filtlist=""
      WHILE $DATA(result(j)) {
        SET intId = $LISTGET(result(j),1)
        IF intId#3 > 0 {SET filtlist=filtlist_","_intId }
        SET j=j+1
    }
  SET tTrainingSet=##class(%iKnow.Filters.SourceIdFilter).%New(domId,filtlist)
  SET tTestSet = ##class(%iKnow.Filters.GroupFilter).%New(domId, "AND", 1) // NOT filter
  DO tTestSet.AddSubFilter(tTrainingSet)
DisplaySourceCounts
  SET trainsrc = ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,tTrainingSet)
    WRITE "The training set contains ",trainsrc," sources",!
  SET testsrc = ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId,tTestSet)
    WRITE "The test set contains ",testsrc," sources",!
```

Note that %iKnow.Filters.RandomFilter is another way to divide a group of sources. However, each time you invoke %iKnow.Filters.RandomFilter the resulting training set consists of different sources.

13.3 Building a Text Classifier Programmatically

To build a text classifier, you use a %iKnow.Classification.Builder object. The description that follows applies to the analytic approach to building a Text Classifier.

13.3.1 Create a Text Classifier

To create a Text Classifier, you must first instantiate a Builder object, supplying it the domain name and the oref for the training set. You then configure the ClassificationMethod algorithm that the Text Classifier will use. The easiest-to-use algorithm is based on the Naive Bayes theorem. Naive Bayes combines individual entities' probabilities for each category in the training set to calculate the overall probability of a new text belonging to that category:

ObjectScript

```
SET tBuilder = ##class(%iKnow.Classification.IKnowBuilder).%New("mydomian",tTrainingSet)
SET tBuilder.ClassificationMethod="naiveBayes"
```

You then specify the categories that the Text Classifier will use. If your sources supply the category labels as a metadata field, you can make a single call to the **%LoadMetadataCategories()** method. You do not need to specify either the category values or even the number of categories. In the following example, the AircraftCategory metadata field of Aviation.Aircraft is used as a category field assigning each record to a category: "Airplane", "Helicopter", "Glider", "Balloon", etc. The following example shows the use of this metadata field to specify categories:

ObjectScript

```
SET myquery="SELECT TOP 100 E.ID,A.AircraftCategory,E.Type,E.NarrativeFull "_
"FROM Aviation.Aircraft AS A,Aviation.Event AS E "_
"WHERE A.Event=E.ID"
SET idfld="ID"
SET grpfld="Type"
SET dataflds=$LB("NarrativeFull")
SET metaflds=$LB("AircraftCategory")
SET mstat=##class(%iKnow.Queries.MetadataAPI).AddField(domId,"AircraftCategory")
IF mstat=1 {
SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds,metaflds) }
.
.
.
WRITE tBuilder.%LoadMetadataCategories("AircraftCategory")
```

Note that this is a useful (but not an ideal) category field, because a large percentage of the records (>80%) are assigned to "Airplane", whereas most other labels have only a handful of records assigned to them. Ideally, each category label should correspond to roughly equivalent numbers of texts. As long as each category represents at least 10% of the training set, most classification methods should work fine. A category label must be associated with more than one source text; it may, therefore, be useful to combine potential category values with very low numbers of texts into a catch-all category, with a category label such as "miscellaneous".

13.3.2 Populate the Terms Dictionary

Once you have established categories, you select terms which the Text Classifier will locate in each text and use to determine what category label to assign to it. You can either assign terms individually, or use **%PopulateTerms()** to add multiple terms found in the texts according to some metric.

%PopulateTerms() allows you to automatically specify a number of terms based on their frequency in the texts. By default the terms are selected using the Naive Bayes algorithm (most differentiating per-category probability):

ObjectScript

```
SET stat=tBuilder.%PopulateTerms(50)
```

Implementations for metrics other than Naive Bayes can be provided by subclasses. You can use **%PopulateTerms()** to specify the top X number of terms from the training set documents using the BM25 or TFIDF algorithm.

You will typically use a combination of **%PopulateTerms()** and **%AddEntity()** methods to create the desired set of terms.

You specify individual terms to include in the Text Classifier, using the **%AddEntity()**, **%AddCRC()**, and **%AddCooccurrence()** methods:

%AddEntity() can add an entity as a single term, or add multiple entities as a single composite term by supplying the entities as an array or list. iKnow aggregates the counts and scores of these entities, allowing you to capture synonyms or group variants of a term.

ObjectScript

```
DO tBuilder.%AddEntity("hang glider")
DO tBuilder.%AddEntity("fixed wing aircraft","explicit","partialCount")
  SET tData(1)="helicopter",tData(2)="helicopters",tData(3)="twin-rotor helicopter"
DO tBuilder.%AddEntity(.tData)
```

%AddEntity() can optionally specify how to handle negation and how to handle partial matches, as shown in the second **%AddEntity()** in the previous example.

%AddCRC() can add a CRC as a single term. Because text classification depends on the frequency of matches amongst the source texts, it is unusual for a CRC to be common enough to be useful as a Text Classifier term. However, if there is a very specific sequence of entities (a CRC) that is a strong indicator for a particular category, adding CRCs can make sense.

%AddCooccurrence() allows you to add as a single term the appearance of two specified entities in the same sentence (in any order). You can optionally specify how to handle negation and how to handle partial matches:

ObjectScript

```
WRITE tBuilder.%AddCooccurrence($LISTBUILD("landed","helicopter pad"))
```

Note that these terms are *not* associated with a particular category. The Builder will automatically calculate how well each text containing these terms correlates to each category.

13.3.3 Run the Classification Optimizer

When developing a Text Classifier, you do not have to add or remove terms by trial and error. You can use the methods of the %iKnow.Classification.Optimizer class to include those entities that will have the largest impact on predictive accuracy.

1. Create an Optimizer object and use its Builder property to specify the %iKnow.Classification.Builder object to associate with it. Optionally, set the ScoreMetric property to specify how you want to measure performance (the default is MacroFMeasure).

   ### ObjectScript

   ```
   SET tOpt = ##class(%iKnow.Classification.Optimizer).%New(domId,tBuilder)
   SET tOpt.ScoreMetric="MicroPrecision"
   ```

2. Include a large number of candidate terms, either from an array (using **LoadTermsArray()**) or using an SQL query (using **LoadTermsSQL()**.

3. Run the **Optimize()** method. This will automatically add terms and remove terms based on their ScoreMetric values. **Optimize()** performs the specified number of rounds of adding potentially high-value terms, calculating their impact, then removing low-value terms.

13.3.4 Generate the Text Classifier

Once you have identified categories and terms, you generate a Text Classifier class. This text classifier class contains code to identify the most appropriate category label based on the terms found in the source. You specify a class name for your Text Classifier:

ObjectScript

```
WRITE tBuilder.%CreateClassifierClass("User.MyClassifier")
```

The operation performed by this method depends on the ClassificationMethod you specified. For Naive Bayes, the Builder first creates a matrix containing the match score/count for each term in each source text for which we also know the actual category. This builds a model of how well the specified terms are predictive of the assigned category.

In the example used here, the categories were taken from the AircraftCategory metadata field values. Each term is correlated with each source to determine how predictive that term is in determining the category. For example, the appearance of the term "helipad" is strongly predictive of a source with AircraftCategory=helicopter. The term "engine" is indicative of several categories — airplane or helicopter, but not glider or balloon — and is thus weakly predictive of a single category.

However, including a term of this type may be helpful for eliminating some categories. The term "passenger" is only weakly predictive of any category, and is therefore probably not a good term for your text classifier model. You can use **%AddEntity()** and **%RemoveTerm()** to fit your dictionary of terms based on their contribution to the determination of a category.

13.4 Testing a Text Classifier

Your text classifier model has been fitted to its training set of documents so that the set of terms in its term dictionary accurately determine the category. You now need to test the model on a separate set of documents to determine if it is accurate for documents other than those in the training set. For this, you use the test set of documents. Like the training set, these documents also have a defined category label.

You can use the **%TestClassifier()** method to return a single accuracy value. The accuracy is the number of right predictions made divided by the total records tested. The higher the accuracy against the test set documents, the better the model.

ObjectScript

```
WRITE tBuilder.%TestClassifier(tTestSet,,.accuracy),!
WRITE "model accuracy: ",$FNUMBER(100*accuracy,"L",2)," percent"
```

It is likely that the predictive accuracy for all categories is not the same. You should therefore test the accuracy for individual categories.

The following example returns both the overall accuracy and the individual incorrect prediction results:

ObjectScript

```
TestClassifier
  WRITE tBuilder.%TestClassifier(tTestSet,.testresult,.accuracy),!
  WRITE "model accuracy: ",$FNUMBER(accuracy*100,"L",2)," percent",!
  SET n=1
  SET wrongs=0
  WHILE $DATA(testresult(n)) {
    IF $LISTGET(testresult(n),2) '= $LISTGET(testresult(n),3) {
      SET wrongcnt=wrongcnt+1
      WRITE "WRONG: ",$LISTGET(testresult(n),1)
      WRITE " actual ",$LISTGET(testresult(n),2)
      WRITE " pred. ",$LISTGET(testresult(n),3),! }
    SET n=n+1 }
  WRITE wrongcnt," out of ",n-1,!
```

Predictive accuracy for a category is calculated based on four possible outcomes of matching a prediction to a known category:

- True Positive (TP): predicted as Category X, actually in Category X.

- False Positive (FP): predicted as Category X, actually in some other category.

- False Negative (FN): predicted as some other category, actually in Category X.

- True Negative (TN): predicted as some other category, actually in some other category.

These counts are used to generate the following ratios:

Precision is the ratio of correct results to the number of results returned for a particular category: TP / (TP+FP). For example, the term "helipad" would contribute to a high precision ratio for the category Helicopter; nearly all texts that mention "helipad" are in the category Helicopter.

Recall is the ratio of correct results to the number of results that should have been returned for a particular category: TP / (TP+FN). For example, the term "helipad" is not likely to improve the recall ratio for the category "Helicopter" because only a few of these texts mention "helipad".

The *F-measure* (F1) of the model for Category X combines the Precision and Recall values and derives the harmonic mean value of the two. Note that an increase in Precision may cause a decrease in Recall, and vice versa. Which of the two you wish to maximize depends on your use case. For example, in a medical screening application you may wish to accept more False Positives to minimize the number of False Negatives.

Note: If a category value was not found in the training set, the Text Classifier cannot predict that category for a text in the test set. In this case, both the True Positive (TP) and False Positive (FP) will be zero, and False Negative (FN) with be the full count of texts with that category specified.

13.4.1 Using Test Results

If there is a significant discrepancy between the accuracy of the training set and the accuracy of the test set, the terms dictionary has been "overfitted" to the training set. To correct this problem, go back to the Build process and revise the term dictionary. You can generalize the term dictionary by replacing an individual term with a term array:

ObjectScript

```
SET stat=tBuilder.%RemoveTerm("Bell helicopter")
SET tData(1)="Bell helicopter",tData(2)="Bell 206 helicopter",tData(3)="Bell 206A helicopter",
    tData(4)="Bell 206A-1 helicopter",tData(5)="Bell 206L helicopter",tData(6)="Bell 206L LongRanger"

SET stat=tBuilder.%AddEntity(.tData)
```

You can also generalize the term dictionary by changing an individual term to allow for partial matches ("partialCount"), rather than only an exact match.

13.5 Building a Text Classifier Using the UI

You can build a Text Classifier using the Caché Management Portal. From the Management Portal **System Explorer** option, select the **iKnow** option, then select **Text Categorization**. This displays two options: **Model builder** and **Model tester**.

All iKnow domains exist within a specific namespace. Therefore, you must specify which namespace you wish to use. A namespace must be iKnow-enabled before it can be used. Selecting an iKnow-enabled namespace displays the iKnow **Text Categorization** option.

Note: You cannot use the %SYS namespace for iKnow operations. The Management Portal **iKnow** option is non-functional (greyed out) while in the %SYS namespace. Therefore, you must specify which existing namespace you wish to use by clicking the **Switch** option at the top of any Management Portal interface page before using the **iKnow** option.

You may also need to activate %iKnow UI classes for your web application. Open the Terminal and run the activation utility for the desired namespace, as follows: `DO EnableIKnow^%SYS.cspServer("/csp/samples/")`. (This example activates the Samples namespace.)

In this interface, you can either open an existing Text Classifier or build a new one. To build a new Text Classifier, you must already have a defined domain containing data sources. The data sources must contain a category field.

13.5.1 Define a Data Set for the UI

To create a new Text Classifier, you must have created a domain and populated it with data sources that can be used as the training set and the test set. Commonly, data from these sources should specify the following:

- One or more data fields containing the text to be analyzed by the Text Classifier.

- A metadata field containing the categories used by the Text Classifier. The data sources must contain at least one source for every possible category value.

- A metadata field containing values that can be used to divide the data sources into a training set and a test set. This field should have no connection to the actual contents of the data, and thus enable a random division of the sources. For example, the source Id number, or a date or time value. In many cases, you will need to specify less than (<) and greater than (>) operators to enable the division of sources into sets.

The following is an example of data source field definitions:

ObjectScript

```
SET myquery="SELECT TOP 200 E.ID,A.AircraftCategory,E.Type,E.NarrativeFull "_
    "FROM Aviation.Aircraft AS A,Aviation.Event AS E "_
    "WHERE A.Event=E.ID"
SET idfld="ID"
SET grpfld="Type"
SET datafdls=$LB("NarrativeFull")
SET metaflds=$LB("AircraftCategory","ID")
DO ##class(%iKnow.Queries.MetadataAPI).AddField(domId,"AircraftCategory")
DO ##class(%iKnow.Queries.MetadataAPI).AddField(domId,"ID",$LB("=","<=",">"))
```

13.5.2 Build a Text Classifier

To create a new text classifier, select the **New** button. This displays the **Create a new classifier** window. Create a new **Class name** for your Text Classifier. Select an iKnow **Domain** from the drop-down list of existing domains. Select the **Category field** containing the category labels from the drop-down list of metadata fields defined for the domain. For **Training Set**, select a metadata field from the drop-down list, an operator from the drop-down list, and specify a value. For example: `EventDate <= 2004`. For the **Test Set**, select the same metadata field, and a complementary operator and value. For example: `EventDate > 2004`. Alternatively, you can specify the **Training Set** and **Test Set** using **SQL** with an SQL query.

For **Populate terms** select a method of deriving terms from the drop-down list, and specify the number of top terms to derive. For example **Top n terms by NB differentiation** for Naive Bayes (NB). Then click **Create**.

This displays a screen with three panels:

The right panel displays the **Model properties**. Normally, you would not change these values. Clicking on the Data source domain name allows you to change the **Category field**, **Training Set**, or **Test Set** specified in the previous step. If you click the **Gears** icon in the button bar, you'll display some additional advanced controls.

The central panel (**Selected terms**) shows a tree view of the terms that you have already selected as part of the model. The left panel (**Add terms**) allows you to add terms to the model.

13.5.2.1 Terms Selection

Here are the most common ways to add entities to a terms dictionary:

- **Entities** tab: type a substring in the box provided. All of the entities that contain that substring will be listed with their frequency and spread counts.

- **Top** tab: select a **metric** (BM25 or TFIDF) from the drop-down list. The top entities according to that metric will be listed with their calculated score.

You can use the check boxes to select individual entities as terms, or you can scroll to the bottom of the list and click **select all** for the current page of listed entities. You can go on to additional pages, if desired. Once you have selected entities, scroll back up to the top of the **Add terms** list and click the **Add** button. This adds the selected entities to the **Selected terms** list in the central panel. (You may get an informational message if you have selected duplicate terms.)

13.5.2.2 Optional Advanced Term Selection

The **Append** button allows you to create composite terms. In a composite term, multiple similar entities all resolve to the same single term. Defining composite terms helps to make the Text Classifier more robust and general, less "fitted" to specific entities in the training set. You can also create a composite term in the **Selected terms** list by dragging and dropping one term atop another.

By default, all multiword entities you select are added to the terms dictionary as an exact occurrence: all of the words in the entity must appear in the specified order in a text to qualify as a match. However, you can instead specify individual entities as partial match entities. When adding a multiword entity, click the **Gears** icon to display the **Count** drop-down box. Select partial match count. Defining common multiword entities as partial match terms helps to make the Text Classifier more robust and general, less "fitted" to specific entities in the training set.

You can use the **CRCs** tab or the **Cooccurrences** tab to add these entities to the terms dictionary. Type an entity in the box provided, then press **Enter**. The top CRCs and Cooccurrences appear in the **Add terms** list in descending order. (A cooccurence is the appearance of two Concept entities, in any order, in the same sentence.) Commonly, most CRCs and cooccurences are too specific to add to a term dictionary. However, you may wish to add the most common CRCs and/or cooccurrences to the term directory.

You can remove terms from the **Selected terms** list in the central panel; click on an individual term (or entity within a composite term), then click the **Remove** button.

13.5.2.3 Save

When you have finalized the **Model properties** and list of **Selected terms**, click the **Save** button (or the **Save as** button). This builds your Text Classifier.

13.5.3 Optimize the Text Classifier

Once you have saved a Text Classifier, you can use the Optimizer to automatically optimize the term dictionary. The Optimizer takes a list of additional candidate terms, tests each term, and adds those terms with the highest impact on the accuracy of the Text Classifier.

Click the **Optimize** button. This displays the **Optimize term selection** popup window. Select a **Relevance metric** (BM25, TFIDF, or Dominance) from the drop-down list. Specify a number of candidate terms using that metric, and click **Load**. The right panel lists the **Candidate terms to test**. Click **Next**.

This displays the **Settings** panel with default values. You can accept these defaults and click **Start**. This runs the Optimizer, adding and removing terms. When the optimization process completes, you can close the **Optimize term selection** popup window. Note that the **Selected terms** list in the central panel has changed. Click the **Save** button to save these additions to your terms dictionary.

You can run the Optimizer several times with different settings. After each optimization you can test the Text Classifier, as described in the following section.

13.5.4 Test the Text Classifier against a Test Set of Data

Click the **Test** button. This displays the **Model tester** in a new browser tab. The **Model tester** allows you to test your Text Classifier against test data. After testing it, you can return to the **Model builder** browser tab and add or remove terms, either manually using **Add terms** , or by running (or re-running) the Optimizer.

The **Model tester** provides two ways of testing your Text Classifier:

- **Domain tab**: the **Domain**, **Category field**, and **Test filter** fields should take their values from the **Model builder**. In the **Model tester** click the **Run** button. This displays overall test results, and Detail test results for each category.

- **SQL tab**: you can specify an SQL query to supply the test data in the Data Source SQL section. Use the _Text and _Category column aliases to identify the source text and the metadata category columns, as shown in the following example:

SQL

```
SELECT E.ID,A.AircraftCategory AS _Category,E.Type,E.NarrativeFull AS _Text
FROM Aviation.Aircraft AS A,Aviation.Event AS E WHERE A.Event=E.ID
```

Then click **Test**. This compares the actual category value with the category determined by the Text Classifier. It displays the overall test results, and details for each category.

13.5.5 Test the Text Classifier on Uncategorized Data

You can use the Text Classifier on a text string to test how it derives a category. Select the **Test** button. This displays the **Text input** window. Specify a text string, then press **Categorize!**.

- The **Text** tab displays the text highlighted with the terms from the terms dictionary.

- The **Categories** tab displays score bars for each category, with green representing correct and brown representing incorrect.

- The **Trace info** tab displays probability bars for each term found in the input text. By using the Weights for category drop-down list you can determine the probability for each term for all categories (the default), or for individual categories.

Th

13.6 Using a Text Classifier

Once you have built an accurate text classifier, you will want to apply it to source texts that have not yet been assigned a category label. Using methods of %iKnow.Classification.Classifier, your text classifier can be used to predict the category of any unit of text.

To create a Text Classifier, you must first instantiate the subclass of %iKnow.Classification.Classifier that represents the Text Classifier you wish to run. Once created, a text classifier is completely portable; you can use this text classifier class independently of the domain that contains the training set and test set data.

- Use **%Categorize()** for iKnow source texts. If a source text to be categorized has already been indexed in an iKnow domain, you can use **%Categorize()** to match against the categories. It returns a match score for each category, in descending order by score.

ObjectScript

```
SET tClassifier = ##class(User.MyClassifier).%New("iKnow","MyDomain")
WRITE tClassifier.%Categorize(.categories,srcId)
ZWRITE categories
```

- Use **%CategorizeText()** for a text specified as a string. If a source text to be categorized is a string, you can use **%CategorizeText()** to match an input string against the categories. It returns a match score for each category, in descending order by score.

ObjectScript

```
SET tClassifier = ##class(User.MyClassifier).%New()
WRITE tClassifier.%CategorizeText(.categories,inputstring)
ZWRITE categories
```

`ZWRITE categories` returns match score data such as the following:

```
categories=4
categories(1)=$lb("AIRPLANE",.4314703807485703701)
categories(2)=$lb("HELICOPTER",.04128622469233822948)
categories(3)=$lb("GLIDER",.0228365968611826442)
categories(4)=$lb("GYROCRAFT",.005880588058805880587)
```

In SQL you can execute a Text Classifier against a text string using a method stored procedure, as follows:

```
SELECT User.MyClassifier_sys_CategorizeSQL('input string') AS CategoryLabel
```

This returns a category label.

The following Embedded SQL example uses the Sample.MyTC Text Classifier to determine a category label for the first 25 records in Aviation.Event:

ObjectScript

```
ZNSPACE "Samples"
FOR i=1:1:25 {
    SET rec=i
    &sql(SELECT %ID,NarrativeFull INTO :id,:inputstring FROM Aviation.Event WHERE %ID=:rec)
        WRITE "Record ",id
    &sql(SELECT Sample.MyTC_sys_CategorizeSQL(:inputstring) INTO :CategoryLabel)
        WRITE " assigned category: ",CategoryLabel,!
}
```

Once texts have been classified, you can use this classification to filter texts using the %iKnow.Filters.SimpleMetadataFilter class. See Filtering by User-defined Metadata for further details.

14

Dominance and Proximity

The iKnow semantics package consists of two classes: %iKnow.Semantics.DominanceAPI and %iKnow.Semantics.ProximityAPI. These classes with their parameters and methods are described in the *InterSystems Class Reference*.

This chapter describes:

- Semantic Dominance
- Semantic Proximity

14.1 Semantic Dominance

Semantic dominance is the overall importance of an entity within a source. iKnow determines semantic dominance by performing the following tests and obtaining statistical results:

- The number of times that an entity appears in the source (the frequency).
- The number of times that each component word of an entity appears in the source.
- The number of words in the entity.
- The type of entity (concept or relation).
- The diversity of entities in the source.
- The diversity of component words in the source.

Note: Japanese uses a different, language-specific set of statistics to calculate the semantic dominance of an entity. See iKnow Japanese.

iKnow generates these values when the sources are loaded as part of iKnow indexing. It combines these values to produce a dominance score for each entity. The higher the dominance score, the more dominant the entity is within the source.

For example, for the concept "cardiovascular surgery" to be semantically dominant in a document, a statistical analysis would be performed. It determines that this concept appears 20 times in the source. A dominant concept is not the same as a top concept. In this source the concepts "doctor" (60 times), "surgery" (50 times), "operating room" (40 times), and "surgical procedure" (30 times) are far more common. However, the component words of "cardiovascular surgery" appear over twice as many times as the concept itself: "cardiovascular" (50 times) and "surgery" (80 times), which lends support to this being a dominant concept. In contrast, the concept "operating room" appears 40 times, but its component words appear barely more often than the concept: "operating" 60 times and "room" only 45 times. This indicates that this source is much more concerned with cardiovascular matters and surgery than it is with rooms.

iKnow gives these frequency counts greater or lesser weight based on the number of words in the original entity and whether that entity is a concept or a relation. (There is a much smaller number of commonly-occurring relations than concepts.)

However, to determine how dominant a concept really is, iKnow has to compare it to the total number of concepts in the source. If 5% of the concepts in the source contain the words "cardiovascular" and "surgery", and these words do not combine in other concepts nearly as frequently as they do together, we know that these words not only appear frequently in the source, but that the source does not have a wide range of subject matter. If, however, the source contains a nearly equal occurrence of the word "surgery" in concepts with "hand", "kidney" and "brain" and the word "cardiovascular" appears nearly as often with the words "exercise" and "diet" it is apparent that the source contains a wide range of subject matter. The concept "cardiovascular surgery" and its component words may appear more frequently than others, but may not significantly dominate the subject matter of the source.

By performing these statistical calculations, iKnow can determine the dominant concepts in a source — the subjects that are of greatest interest to you. iKnow performs this analysis without using an external reference corpus (such as pre-existing table of the relative frequency of words in a "typical" medical text). iKnow determines dominance using only the contents of the actual source text, and thus can be used on sources on any topic without any prior knowledge of the subject matter.

14.1.1 Dominance in Context

iKnow calculates the dominance of entities (concepts and relations) within a source and assigns each an integer value. It assigns the most dominant concept(s) in a source a dominance value of 1000. You can use the Indexing Results tool to list the dominance values for concepts in a single source.

iKnow calculates the dominance of CRCs within a source. The algorithm uses the dominance values of the entities within the CRC. CRC dominance values are intended only for comparison with other CRC dominance values; CRC dominance values should not be compared with entity dominance values. You can use the Indexing Results tool to list the dominance values for CRCs in a single source.

iKnow calculates the dominance of concepts across all loaded sources as a weighted average. These concept dominance scores are fractional numbers, with the largest possible number being 1000. You can use the Knowledge Portal tool **Dominant Concepts** option to list the dominance scores for concepts in all loaded sources. You can also use the %iKnow.Semantics.DominanceAPI **GetTop()** method to list the dominance scores for concepts in all loaded sources.

14.1.2 Concepts of Semantic Dominance

The following are the key elements of semantic dominance:

- Profile: the counts of elements that are used to calculate a dominance score.

- Typical: a typical source is a source in which the dominant entities in that source are most similar to the dominant elements of the group of sources. This is the opposite of Breaking.

- Breaking: a breaking source is a source in which the dominant entities in that source are *least* similar to the dominant elements of the group of sources. This is the opposite of Typical. For example, a breaking news story would likely be least similar to the dominant entities in all of the news stories from the previous month.

- Overlap: the number of occurrences of an entity in different sources.

- Correlation: a comparison of the entities in a source with a list of entities, returning a correlation percentage for each source.

14.1.3 Semantic Dominance Examples

This chapter describes and provides examples for the following semantic dominance queries:

- Dominance profile counts for the domain

- Dominance scores for top concepts in the domain

- Dominance scores for CRCs in the domain

- Dominance score for a specified entity

- Overlap counts between sources

- Typical sources

- Breaking sources

- Sources by correlation

14.1.3.1 Dominance Profile Counts

The following example uses the **GetProfileCountByDomain()** method to return the total count of unique values for an entity type in the specified domain. The available entity types are: 0=concept ($$$SDCONCEPT or $$$SDENTITY); 1=relation ($$$SDRELATION); 2=CRC ($$$SDCRC); 4=aggregate ($$$SDAGGREGATE, the default) which is the total of all concepts, relations, and CRCs.

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
    { WRITE "The ",dname," domain already exists",!
      SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
      GOTO DeleteOldData }
  ELSE
    { WRITE "The ",dname," domain does not exist",!
      SET domoref=##class(%iKnow.Domain).%New(dname)
      DO domoref.%Save()
      WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
      GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT Top 25 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
  SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
    IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
    IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
ProfileCounts
  WRITE ##class(%iKnow.Semantics.DominanceAPI).GetProfileCountByDomain(domId,$$$SDCONCEPT)," total
concepts",!
  WRITE ##class(%iKnow.Semantics.DominanceAPI).GetProfileCountByDomain(domId,$$$SDRELATION)," total
relations",!
  WRITE ##class(%iKnow.Semantics.DominanceAPI).GetProfileCountByDomain(domId,$$$SDCRC)," total CRCs",!

  WRITE ##class(%iKnow.Semantics.DominanceAPI).GetProfileCountByDomain(domId)," aggregate total",!
```

14.1.3.2 Concepts with Top Dominance Scores in the Domain

The **GetTop()** method returns the top concepts (or relations) by dominance scores for all loaded sources.

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { WRITE "The ",dname," domain already exists",!
       SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
       GOTO DeleteOldData }
  ELSE
     { WRITE "The ",dname," domain does not exist",!
       SET domoref=##class(%iKnow.Domain).%New(dname)
       DO domoref.%Save()
       WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
       GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT Top 25 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
     IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
     IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceCount
  SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
  WRITE "The domain contains ",numSrcD," sources",!!
DominantConcepts
  DO ##class(%iKnow.Semantics.DominanceAPI).GetTop(.profresult,domId,1,50)
  WRITE "Top Concepts in Domain by Dominance Score",!
  SET j=1
  WHILE $DATA(profresult(j),list) {
     WRITE $LISTGET(list,2)
     WRITE ": ",$LISTGET(list,3),!
     SET j=j+1 }
  WRITE !,"Printed ",j-1," dominant concepts"
```

14.1.3.3 CRC Dominance Scores in the Domain

The **GetProfileByDomain()** method returns the dominance scores for CRCs in a domain. For each entity returns the following:

- The entity source Id, a unique integer.

- The entity value (the external Id). Because this method can return not only single concepts, but CRCs made up of multiple concepts and relations, this value is returned as a list structure.

- The entity type, an integer code: 0=concept ($$$SDCONCEPT or $$$SDENTITY, the default); 1=relation ($$$SDRELATION); 2=CRC ($$$SDCRC). The value 4=aggregate ($$$SDAGGREGATE) returns first all concepts, then all relations, then all CRCs.

   Note: Paths are not supported at this time

- The calculated dominance value. Dominance values for concepts and relations are always integers. Dominance values for CRCs may be integers, or fractional numbers, such as 3480.5, that are half of an integer.

The following example uses the **GetProfileByDomain()** method to return the dominant profile entities, in this case CRCs. For each CRC it returns the value, the source Id, the type, and the dominance score. Because they are all CRCs, the type is always 2:

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT Top 25 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceSentenceQueries
  SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
  WRITE "The domain contains ",numSrcD," sources",!
  SET numSentD=##class(%iKnow.Queries.SentenceAPI).GetCountByDomain(domId)
  WRITE "These sources contain ",numSentD," sentences",!!
DominanceProfile
  DO ##class(%iKnow.Semantics.DominanceAPI).GetProfileByDomain(.profresult,domId,1,20,$$$SDCRC)
  SET j=1
  WHILE $DATA(profresult(j),list) {
      WRITE " ",$LISTTOSTRING($LISTGET(list,2))
      WRITE " [Id:",$LISTGET(list,1)
      WRITE " type:",$LISTGET(list,3)
      WRITE " Dominance:",$LISTGET(list,4),"]",!
      SET j=j+1 }
  WRITE !,"Printed ",j-1," dominant profile CRCs"
```

14.1.3.4 Dominance Score for a Specified Entity

iKnow uses the **GetDomainValue()** method to return the dominance value for a specified entity. You specify the entity by its entity Id (a unique integer), and specify the entity type by a numeric code. The default entity type is 0 (concept).

A single set of unique entity Ids is used for concepts and relations; therefore, no concept has the same entity Id as a relation. A separate set of unique entity Ids is used for CRCs; therefore, a CRC may have the same entity Id as a concept or a relation. This numbering is entirely coincidental; there is no connection between the entities.

The following example takes the top 12 entities and determines the dominance score for each entity. As one can see from the results of this example, the top (most frequently occurring) entities do not necessarily correspond to the entities with the highest dominance scores:

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
```

```
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT Top 25 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
  SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
TopEntitiesDominanceScores
  DO ##class(%iKnow.Queries.EntityAPI).GetTop(.result,domId,1,12)
  SET i=1
  WHILE $DATA(result(i)) {
      SET topstr=$LISTTOSTRING(result(i),",",1)
      SET topid=$PIECE(topstr,",",1)
      SET val=$PIECE(topstr,",",2)
        SET spc=25-$LENGTH(val)
      WRITE val
      WRITE $JUSTIFY("top=",spc),i
      WRITE " dominance="
      WRITE ##class(%iKnow.Semantics.DominanceAPI).GetDomainValue(domId,topid,0),!
      SET i=i+1 }
  WRITE "Top ",i-1," entities and their dominance scores"
```

14.1.3.5 Overlap Between Sources

The following example uses the **GetOverlap()** method to return the overlap score for each entity. This score is the count of overlapping occurrences in all of sources in the domain for that entity. Note that before you can invoke **GetOverlap()**, you must first invoke **BuildOverlap()**. When displaying the results, type=0 (concepts) for all of the returned values in this example and is therefore omitted from the display; the $JUSTIFY function is used in this example to align display of the overlap counts:

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT Top 25 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
```

```
     SET dataflds=$LB("NarrativeFull")
UseLister
   SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
       IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
   SET stat=myloader.ProcessBatch()
       IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceSentenceQueries
   SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
   WRITE "The domain contains ",numSrcD," sources",!
   SET numSentD=##class(%iKnow.Queries.SentenceAPI).GetCountByDomain(domId)
   WRITE "These sources contain ",numSentD," sentences",!!
Overlap
   DO ##class(%iKnow.Semantics.DominanceAPI).BuildOverlap(domId)
   DO ##class(%iKnow.Semantics.DominanceAPI).GetOverlap(.result,domId,1,17)
DisplayOverlapResults
   SET i=1
   WHILE $DATA(result(i)) {
       SET datastr=$LISTTOSTRING(result(i),",",1)
         SET spc=40-$LENGTH(datastr)
       WRITE $PIECE(datastr,",",1)," "
       WRITE $LISTTOSTRING($PIECE(datastr,",",2))
       /* WRITE $PIECE(datastr,",",3)," " */
         WRITE $JUSTIFY(" overlap=",spc)
       WRITE $PIECE(datastr,",",4),!
       SET i=i+1 }
```

14.1.3.6 Typical Sources

The following example lists the ten most typical sources with their dominance scores. In order to retrieve typical sources you must issue the **GetOverlap()** method, followed by the **FindMostTypicalSources()** and **GetTypicalSources()** methods.

ObjectScript

```
#include %IKPublic
   ZNSPACE "Samples"
DomainCreateOrOpen
   SET dname="mydomain"
   IF (##class(%iKnow.Domain).NameIndexExists(dname))
       { WRITE "The ",dname," domain already exists",!
         SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
         GOTO DeleteOldData }
   ELSE
       { WRITE "The ",dname," domain does not exist",!
         SET domoref=##class(%iKnow.Domain).%New(dname)
         DO domoref.%Save()
         WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
         GOTO ListerAndLoader }
DeleteOldData
   SET stat=domoref.DropData()
   IF stat { WRITE "Deleted the data from the ",dname," domain",!!
             GOTO ListerAndLoader }
   ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
             QUIT}
ListerAndLoader
   SET domId=domoref.Id
   SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
   SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT Top 25 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseLister
   SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
       IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
   SET stat=myloader.ProcessBatch()
       IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceCountQueries
   WRITE ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)," total sources",!!
DominanceTypicalSources
   SET dstat = ##class(%iKnow.Semantics.DominanceAPI).BuildOverlap(domId)
   IF dstat '=1 {WRITE "BuildOverlap() error"  QUIT}
   SET dstat = ##class(%iKnow.Semantics.DominanceAPI).FindMostTypicalSources(domId)
   IF dstat '=1 {WRITE "FindMostTypicalSources() error"  QUIT}
   SET dstat = ##class(%iKnow.Semantics.DominanceAPI).GetTypicalSources(.srcresult,domId,1,10)
   IF dstat '=1 {WRITE "GetTypicalSources() error"  QUIT}
   SET k=1
```

```
      WHILE $DATA(srcresult(k)) {
            WRITE $LISTTOSTRING(srcresult(k)),!
            SET k=k+1 }
```

14.1.3.7 Breaking Sources

Breaking sources are those sources that differ significantly from the other sources in the domain (or filtered subset of the domain). In order to retrieve breaking sources you must issue the **GetOverlap()** method, followed by the **FindBreakingSources()** and **GetBreakingSources()** methods.

In the following program, after identifying each breaking source, the program uses **GetBySource()** to return the dominant entities for each breaking source. For each entity it returns:

- the entity Id, which is a unique integer for that entity type. However, because GetBySource() returns multiple entity types (concepts, relations, CRCs), this entity Id is only unique and meaningful within the specified type.

- the entity value, which can be a single concept, or a list of concepts and relations. This list can be a CRC. (*To simplify the display, the entity value is here commented out.*)

- the entity type: 0 for a concept, 1 for a relation, 3 for a CRC, or an odd number > 3 for a path. The path value corresponds to the number of elements in the path. If the number of elements in the path is an even number, the entity type is the next larger odd number.

- the dominance value, which is a calculated positive number greater than 1. The higher the dominance value, the more dominant the entity is within that source. The dominance value for a concept or relation is an integer; the dominance value for a CRC or path may contain a decimal fraction.

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT Top 25 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
  SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceCountQueries
  WRITE ##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)," total sources",!!
DominanceBreakingSources
  SET dstat = ##class(%iKnow.Semantics.DominanceAPI).BuildOverlap(domId)
  IF dstat '=1 {WRITE "BuildOverlap() error"  QUIT}
  SET dstat = ##class(%iKnow.Semantics.DominanceAPI).FindBreakingSources(domId)
  IF dstat '=1 {WRITE "FindBreakingSources() error"  QUIT}
  DO ##class(%iKnow.Semantics.DominanceAPI).GetBreakingSources(.profresult,domId,1,10)
  SET j=1,k=1
  WHILE $DATA(profresult(j),srclist) {
```

```
        SET src = $LISTGET(srclist)
        WRITE !,"Source id: ",src,!
        DO ##class(%iKnow.Semantics.DominanceAPI).GetBySource(.srcresult,domId,src,1,10)
        WHILE $DATA(srcresult(k),list) {
                WRITE "id:",$LISTGET(list)
                /* WRITE "  values:",$LISTTOSTRING($LISTGET(list,2)) */
                WRITE "  type:",$LISTGET(list,3)," dominance:",$LISTGET(list,4),!
                SET k=k+1 }
        SET k=1
        SET j=j+1 }
   WRITE !!,"Printed ",j-1," breaking sources"
```

14.1.3.8 Sources By Correlation

GetSourcesByCorrelation() returns those sources that correlate to a list of entities, calculating the a percentage of correlation between the source's entities and the list entities. A correlation percentage of 1 (100%) means that there are as many correlations in the source as there are listed entities; it does not mean that every listed entity appears in that source. For this reason, it is possible to return a correlation percentage greater that 1. Sources with 0% correlation are not listed.

In order to retrieve sources by correlation you must issue the **GetOverlap()** method, followed by the **GetSourcesByCorrelation()** method. You can supply a filter to **GetOverlap()** to limit the set of sources to be tested for correlation.

The following example tests sources by comparing them to a list of concepts that refer to helicopters. It returns the source ID, the external Id, and the percentage of correlation:

ObjectScript

```
#include %IKPublic
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { WRITE "The ",dname," domain already exists",!
       SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
       GOTO DeleteOldData }
  ELSE
     { WRITE "The ",dname," domain does not exist",!
       SET domoref=##class(%iKnow.Domain).%New(dname)
       DO domoref.%Save()
       WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
       GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT Top 25 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
  SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
     IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
     IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceSentenceQueries
  SET numSrcD=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
  WRITE "The domain contains ",numSrcD," sources",!
  SET numSentD=##class(%iKnow.Queries.SentenceAPI).GetCountByDomain(domId)
  WRITE "These sources contain ",numSentD," sentences",!!
SourcesByCorrelation
  DO ##class(%iKnow.Semantics.DominanceAPI).BuildOverlap(domId)
  SET heliterms=$LB(796,1048,1706,2484,2571,2637,2642,2653,3284,3987,4037,4038,4054,4957)
  DO ##class(%iKnow.Semantics.DominanceAPI).GetSourcesByCorrelation(.result,domId,heliterms)
DisplaySourcesByCorrelation
  SET i=1
  WHILE $DATA(result(i)) {
      WRITE $LISTTOSTRING(result(i),",",1),!
      SET i=i+1 }
```

14.2 Semantic Proximity

Semantic proximity is a calculation of the semantic "distance" between two entities within a sentence. The higher the proximity integer, the closer the entities.

As a demonstration of this semantic distance, given the sentence:

"The giraffe walked with long legs to the base of the tree, then stretched his long neck
 up to reach the lowest leaves."

the proximity of the concept "giraffe" might be as follows: long legs=64, base=42, tree=32, long neck=25, lowest leaves=21.

Semantic proximity is calculated for each entity in each sentence, then these generated proximity scores are added together producing an overall proximity score for each entity for the entire set of source texts. For example, given the sentences:

"The giraffe walked with long legs to the base of the tree, then stretched his long neck
 up to reach the lowest leaves. Having eaten, the giraffe bent his long legs and stretched
 his long neck down to drink from the pool."

the proximity of the concept "giraffe" might be as follows: long legs=128, long neck=67, base=42, tree=32, pool=32, lowest leaves=21.

Entity proximity is commutative; this means that the proximity of entity1 to entity2 is the same as the proximity of entity2 to entity1. iKnow does not calculate a semantic proximity of an entity to itself. For example, the sentence "The boy told a boy about another boy." would not generate any proximity scores, but the sentence "The boy told a younger boy about another small boy." generates the proximity scores younger boy=64, small boy=42. If the same entity appears multiple times in a sentence, the proximity score is additive. For example, the proximity for the concept "girl" in the sentence "The girl told the boy about another boy." is boy=106, the total the two proximity scores 64 and 42.

14.2.1 Japanese Semantic Proximity

iKnow semantic analysis of Japanese uses an algorithm to create Entity Vectors. An entity vector is an ordering of entities in the sentence that follow a predefined logical sequence. When iKnow converts a Japanese sentence into an entity vector it commonly rearranges the order of entities. Semantic proximity for Japanese uses the entity vector entity order, not the original sentence entity order.

14.2.2 Proximity Examples

Refer to A Note on Program Examples for details on the coding and data used in the examples in this book.

The following example uses the **GetProfile()** method to return the proximity of the concept "student pilot" to other concepts in sentences in all of the sources in the domain. **GetProfile()** supports filters and blacklists:

ObjectScript

```
#include %IKPublic
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
    { WRITE "The ",dname," domain already exists",!
      SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
      GOTO DeleteOldData }
  ELSE
    { WRITE "The ",dname," domain does not exist",!
      SET domoref=##class(%iKnow.Domain).%New(dname)
      DO domoref.%Save()
      WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
      GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
```

```
   ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
             QUIT}
ListerAndLoader
   SET domId=domoref.Id
   SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
   SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT Top 25 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseLister
   SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
   SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
ProximityForEntity
      SET entity="student pilot"
      DO ##class(%iKnow.Semantics.ProximityAPI).GetProfile(.eresult,domId,entity,1,20)
        SET k=1
        WHILE $DATA(eresult(k)) {
            SET item=$LISTTOSTRING(eresult(k))
            WRITE $PIECE(item,",",1)," ^ "
            WRITE $PIECE(item,",",2)," ^ "
            WRITE $PIECE(item,",",3),!
            SET k=k+1 }
      WRITE !,"all done"
```

The following example uses the **GetProfileBySourceId()** method to list the concepts with the greatest proximity to a given entity for each source. Each concept is listed by entity Id, value, and proximity score:

ObjectScript

```
#include %IKPublic
DomainCreateOrOpen
   SET dname="mydomain"
   IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
   ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
   SET stat=domoref.DropData()
   IF stat { WRITE "Deleted the data from the ",dname," domain",!!
             GOTO ListerAndLoader }
   ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
             QUIT}
ListerAndLoader
   SET domId=domoref.Id
   SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
   SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT Top 25 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseLister
   SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
   SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
SourceCountQuery
   SET totsrc=##class(%iKnow.Queries.SourceAPI).GetCountByDomain(domId)
GetEntityID
   SET entId=##class(%iKnow.Queries.EntityAPI).GetId(domId,"student pilot")
QueryBySource
   DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,totsrc)
   SET j=1,k=1
   WHILE $DATA(result(j),srclist) {
     SET src = $LISTGET(srclist)
     WRITE !,"Source id: ",src,!
     SET entity="student pilot"
    DO ##class(%iKnow.Semantics.ProximityAPI).GetProfileBySourceId(.srcresult,domId,entId,src,1,totsrc)

        WHILE $DATA(srcresult(k)) {
```

```
        SET item=$LISTTOSTRING(srcresult(k))
        WRITE $PIECE(item,",",1)," ^ "
        WRITE $PIECE(item,",",2)," ^ "
        WRITE $PIECE(item,",",3),!
        SET k=k+1 }
  SET k=1
  SET j=j+1 }
WRITE !!,"Printed all ",j-1," sources"
```

15

Custom Metrics

You can customize iKnow to allow users to add their own custom metrics for entities, CRCs, CCs, paths, sentences and/or sources. These metrics can be populated for all sources in the domain, or for select sources. You can then query by these custom metric values.

Note: Custom Metrics software is a Technology Preview, a new category of software at 2013.1. A Technology Preview is intended as a way of introducing and providing access to new software capabilities that InterSystems believes will be useful in enhancing the effectiveness of existing and future applications.

The capabilities listed here are ready for use by customers, but they are not yet complete in functionality and design. Customers who take advantage of these capabilities must understand:

- InterSystems makes no backward compatibility guarantee with future updates;

- Customers may incorporate these capabilities in deployed applications, but must first check with InterSystems to determine best course of action;

- Customers who deploy these capabilities in their applications must commit to upgrading to the final released version.

InterSystems strongly encourages those who incorporate these items in their software to provide feedback on their experiences.

15.1 Implementing Custom Metrics

Implementation of custom metrics involves the following steps:

1. Define a metric: To define one or more metrics, implement a subclass of %iKnow.Metrics.MetricDefinition and add an XData block named "Metrics" that specifies the properties for one or more custom metrics.

2. Define the metric computation: To compute the values of these metrics, implement the calculations in a %iKnow.Metrics.MetricBuilder subclass. This subclass should implement one or more of the **Calculate***Metrics()** methods to support calculating custom metrics for the corresponding target and type. For example, **CalculateEntUniMetrics()** calculates a metric targeting entities, applicable to the whole domain.

3. Register the metric definition: To associate a metric definition with a particular domain, call the **Register()** method of the %iKnow.Metrics.MetricDefinition class to register your %iKnow.Metrics.MetricBuilder subclass as the "builder class" in the definition. This registers all metrics, their properties and targets, as defined in the "Metrics" XData block. Registering the metric definition configures any data structures required to store the metric values.

If the domain was created as a subclass of %iKnow.DomainDefinition and metrics were defined for the domain, you do not have to call the **Register()** method to register your metric definition.

4. Build the metric values: To build the metric values using a %iKnow.Metrics.MetricBuilder implementation, call the **Build()** or **BuildAll()** method of the builder class. These methods build all applicable metrics, forwarding the actual metric calculations to the **Calculate***Metrics()** implementations. You can specify a filter to limit the custom metrics build to a defined subset of sources.

    Unless your %iKnow.Metrics.MetricBuilder subclass overrides the SUPPORTSMULTIPROCESS parameter (default = 1), the **Build()** method will divide the work over multiple processes, each instantiating separate MetricBuilder objects and processing batch after batch of entries using those builder objects. For source-indexed target types, these are batches of sources, for which all the target elements are processed by the same builder process.

5. Optimize the build process: %iKnow.Metrics.MetricBuilder provides a number of callback utility methods, such as **OnBeforeBuildMetrics()**, **OnBeforeBuildMetricsBatch()**, **OnBeforeBuildMetricsForSource()** and the corresponding **OnAfter***()** methods that can help in optimizing performance by pre-calculating values for a whole batch, which are then selected in the individual **Calculate***Metrics()** methods. The individual **Calculate***Metrics()** methods return their values through an output parameter pValues, which is to be populated indexed by metric name (as a single MetricBuilder class can support and calculate multiple metrics in one pass): pValues("MyMetric") = [MyValue].

6. Get the results: %iKnow.Metrics.MetricAPI provides a number of methods that you can use to interact with custom metrics registered in a particular domain. You can use these methods to return the custom metrics results.

    When **Calculate***Metrics()** methods don't return any values for a particular target element, the existing metric value is retained (if any). Returning "" for a value will overwrite (erase) the existing value.

For examples, refer to the Aviation.Metrics.Builder and the Aviation.Metrics.Definition classes in the SAMPLES namespace.

15.2 Types and Targets

- A metric has a type which specifies whether the metric values apply within the context of the whole domain, or just one particular source or metadata group. The available values are $$$IKMTRTYPEDOMAIN, $$$IKMTRTYPE-SOURCE, and $$$IKMTRTYPEGROUP.

    $$$IKMTRTYPEGROUP metrics are valid in the context of a group of sources, as defined by a particular metadata field. Metric Builder classes for this type of metric can use the CurrentFieldName and CurrentFieldValue property values as part of their Calculate***Metrics() implementation.

- A metric has one or more targets, which are the elements the custom metric applies to. The available values are $$$IKMTRENTITY, $$$IKMTRCRC, $$$IKMTRCC, $$$IKMTRPATH, $$$IKMTRSENTENCE, and $$$IKMTRSOURCE. The target is specified by which **Calculate***Metrics()** method you specify. For example, **CalculateEntUniMetrics()** applies to target $$$IKMTRENTITY; how it is applied depends upon the type.

15.3 Copying Metrics

You can copy metrics from one domain to another within the current namespace.

- You can use the **CopyMetrics()** method of the %iKnow.Utils.CopyUtils class to copy all metric definitions in a domain to another domain. Optionally, this method also copies the metric values from one domain to another.

- You can use the **CopyMetric()** method of the %iKnow.Utils.CopyUtils class to copy a single metric definition in a domain to another domain. This method does not copy metric values.

- You can use the **CopyDomain**() method of the %iKnow.Utils.CopyUtils class to copy all metric definitions as part of a domain copy operation. Optionally, this method also copies the metric values as part of the domain copy operation.

16

Smart Matching: Creating a Dictionary

Smart Matching means combining the results of the iKnow Indexing process with some external knowledge you have in the form of a dictionary, taxonomy, or ontology. What makes iKnow matching "smart" is that those Indexing results help you judge the quality of a match because they identify which words belong together to form concepts and relations. For example, iKnow can identify if a match for your dictionary term "flu" is actually referring to the concept "flu" or the concept "bird flu" in your indexed text source. In the latter case, which is called a partial match, it is clear the match should or could be treated differently than the full match where the dictionary term corresponds exactly to the entity in the indexed text source.

To perform Smart Matching, you must create or acquire a dictionary. If you are creating a dictionary, you must then populate it with the items and terms that you wish to use for matching. Once you have a populated dictionary, you can perform matching operations using the contents of the dictionary.

Note: Dictionary definition is not supported for Japanese at this time.

This chapter describes:

- Dictionary conceptual structure and terminology

- Creating a dictionary and populating it with items and terms

- Domain-dependent and domain-independent dictionaries

- Creating dictionary format terms

- Listing dictionaries

- Copying dictionaries

- Extending dictionary constructs

16.1 Introducing Dictionary Structure and Matching

To populate an iKnow dictionary you first create an item, then associate one or more terms with that item. Commonly a dictionary consists of multiple items, with each item associated with multiple terms. An item is a word or phrase that is a relevant tag for many entities in the source texts. When an entity in the source texts is determined to be a match, it is tagged with the item. For example, the item "ship" is a relevant tag for "ship", "boat", "sail", "oars", and so forth.

To perform this matching, you populate each item in the dictionary with match terms. A term can be single entity (like "motor boat") or a phrase or sentence (like "boats are rowed with oars or paddles"). iKnow indexes each term in the dictionary using the same language model used for the source texts. iKnow then matches each term with the same content unit in the

source texts (a Concept term is matched against a Concept in source text; a CRC term is matched against a CRC in source text). If iKnow identifies a match between a term and a unit of source text, iKnow tags the source text passage with the associated dictionary item. This matching frequently is not identical, but requires iKnow to use a scoring algorithm to determine if the term and source text warrant being tagged as a match.

The iKnow dictionary facility supports stemming if stemming has been activated for the current domain. This means that a single dictionary term can match any other form of the same word in a source text.

16.1.1 Terminology

A *Dictionary* is a way to group different terms that have something to do with one another in a logical way. A dictionary could for example be Cities, ICD10 codes, or French wines. As a dictionary is the level of aggregation used within the matching APIs, it is specific to the use case to decide what level of real-world grouping should correspond to a dictionary. Taking a higher level (such as "all ICD10 codes") will yield better performance and use lower disk space, but a lower level (such as "a separate one for all ICD10 categories") might offer grouped results with greater granularity. Each dictionary has a name and a description.

A *Dictionary Item* is a uniquely identifiable item in your dictionary. Examples of a dictionary item could be cities, the individual codes in ICD10 or individual chateaux. Each dictionary typically has many dictionary items (lots of small dictionaries with few items can decrease performance). A dictionary item has a URI, which should be unique within the domain and can be used as an external identifier, and an optional description. This URI can be used when building rules to interpret matching results later on.

A *Dictionary Term* is a string that could appear somewhere in a text and represent the Dictionary Item it belongs to. For example, "Antwerp", "Anvers" and "Antwerpen" could be different terms associated with the same dictionary item representing the city of Antwerp. Dictionary terms are the free text strings on which the actual matching is based when doing string-based matching and could be different spellings, translations or synonyms of what your Dictionary Item stands for. These strings are passed through the engine and, when containing more than just a single entity, will automatically be transformed into a more complex structure to be able to match across the boundaries of a single concept (CRC or Path). A dictionary term should also have a language associated with it, if it needs to be processed by the engine.

When processing a new dictionary term by passing it through the iKnow engine, one or more *Dictionary Elements* are generated to represent the different entities identified within the term. For example, a dictionary term "failure of the liver" would be translated into the three elements "failure", "of" and "liver", with "the" being discarded as non-relevant. These elements are generated and managed automatically and only figure in some types of output, so you shouldn't worry too much about them.

If you want to identify dates, numbers or other formatted pieces of string, you can use *Dictionary Formats* to specify them, and these can then be included in a Dictionary Term, either representing the complete term, or just a single element within a more complex one. A format is a meaningful pattern of characters, such as a date format. You could associate the formats "nn/nn/nnnn" and "nnnn-nn-nn" with the item named Date. iKnow tags any occurrence of these formats in the source texts with the Date item.

16.2 Creating a Dictionary

To define a dictionary use the %iKnow.Matching.DictionaryAPI class methods to define and populate a dictionary, as described in this section. You can define a dictionary specifically for a domain, or define a dictionary that is domain-independent and can be used by any domain in the current namespace.

%iKnow.Matching.DictionaryAPI has a number of methods to create a new dictionary and to assign it items, terms, and formats:

- **CreateDictionary()** is used to create an iKnow dictionary.

The 1st argument specifies the domain Id as an integer. To assign the dictionary to a domain, specify its domain ID as a positive integer. To define the dictionary as domain-independent, specify 0 as its domain ID. The 2nd argument allows you to specify a meaningful dictionary name. The remaining arguments are optional. The 3rd argument allows you to provide a description of the dictionary, the 4th allows you to specify the language (default is English), and the 5th a custom matching profile. **CreateDictionary()** returns the dictId, a unique integer. This dictionary ID is used by subsequent smart matching methods. If a dictionary with the specified name already exists, **CreateDictionary()** returns -1.

- **CreateDictionaryItem()** is used to create an item within a dictionary. You specify the dictId. **CreateDictionaryItem()** returns the dictItemId, a unique integer.

- **CreateDictionaryTerm()** is used to associate a term with an existing item. You supply the dictItemId. **CreateDictionaryTerm()** returns the dictTermId, a unique integer.

- **CreateDictionaryItemAndTerm()** is a shortcut that can be used in a specific case. It can be used to create an item and to create a term associated that item when both the term and the item have the same value. For example the item "flu" might have several associated terms ("influenza", "le grippe", bird flu", "H1N1"); you can use **CreateDictionaryItemAndTerm()** to create the item "flu" and assign it the associated term "flu". You could, of course, perform the same operation using two method calls: **CreateDictionaryItem()** and **CreateDictionaryTerm()**.

- **CreateDictionaryTermFormat()** is used to associate a term that consists of a format with an existing item. You supply the dictItemId. **CreateDictionaryTermFormat()** returns the dictTermId, a unique integer.

16.2.1 Dictionaries and Domains

Each dictionary you create can either be specific to a domain, or can be domain-independent and usable by any domain in the current namespace:

- A domain-specific dictionary is assigned to a domain by specifying a domainId in the **CreateDictionary()** method. You specify the same domainId for the dictionary's items, terms, and formats. This method returns a dictId as a sequential positive integer. Matching methods that use this dictionary reference it by this dictId.

- A domain-independent dictionary is not assigned to a domain. Instead, you specify a domainId of 0 in the **CreateDictionary()** method. You also specify a domainId of 0 for the dictionary's items, terms, and formats. This method returns a dictId as a sequential positive integer. Matching methods that use this dictionary reference it by a negative dictId; for example, the dictionary identified by dictId 8 is referenced by the dictId value -8.

  Note: Domain-independent dictionaries are only available to domains with version 4 (or higher). Domains created prior to Caché 2014.1 must be upgraded to version 4 to use domain-independent dictionaries.

  Using a domain-independent dictionary has important consequences for stemming. When you create a domain-specific dictionary of ordinary terms, iKnow automatically stems the dictionary terms if the domain is configured as stemmed, and therefore dictionary terms and source text match. When you create a domain-independent dictionary, stem conversion of the dictionary terms is not performed. You can either create a dictionary of ordinary (unstemmed) terms, or a dictionary of stemmed terms. A domain-independent dictionary of ordinary terms cannot be matched against a stemmed domain. A domain-independent dictionary of stemmed terms cannot be matched against an unstemmed domain.

Just as several domains can all have a domain-specific dictionary with the same dictId value, both a domain-specific dictionary and a domain-independent dictionary can have the same integer dictId value. Dictionary match operations can use any combination of domain-specific dictionaries (specified as positive integer IDs) and domain-independent dictionaries (specified as negative integer IDs).

Queries in the Matching API returning matching results will return negative identifiers (for the dictId, itemId, and termId) when the match corresponds to an entry in a domain-independent dictionary. All queries will return the combined results for domain-specific and domain-independent dictionary matches, with the exception of **GetDictionaryMatches()** and

GetDictionaryMatchesById(), which only return results for either domain-specific or domain-independent dictionaries, depending on the values specified in the dictIds parameter. The default is domain-specific dictionary matches.

16.2.2 Dictionary Creation Examples

The following example creates a dictionary named "AviationTerms" and populates it with two items and their associated terms. This dictionary is assigned to a specific domain.

ObjectScript

```
  SET domId=##class(%iKnow.Domain).GetOrCreateId("mydomain")
  /* ... */
CreateDictionary
  SET dictname="AviationTerms"
  SET dictdesc="A dictionary of aviation terms"
  SET dictId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionary(domId,dictname,dictdesc)
  IF dictId=-1 {WRITE "Dictionary ",dictname," already exists",!
                 GOTO ResetForNextTime }
  ELSE {WRITE "created a dictionary ",dictId,!}
PopulateDictionaryItem1
  SET itemId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryItem(domId,dictId,
     "aircraft",domId_dictId_"aircraft")
    SET term1Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId,
     "airplane")
    SET term2Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId,
     "helicopter")
PopulateDictionaryItem2
 SET itemId2=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryItemAndTerm(domId,dictId,
     "weather",domId_dictId_"weather")
    SET i2term1Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId2,
     "meteorological information")
    SET i2term2Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId2,
     "visibility")
    SET i2term3Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId2,
     "winds")
DisplayDictionary
  SET stat=##class(%iKnow.Matching.DictionaryAPI).GetDictionaryItemsAndTerms(.result,domId,dictId)
  SET i=1
  WHILE $DATA(result(i)) {
     WRITE $LISTTOSTRING(result(i),",",1),!
     SET i=i+1 }
  WRITE "End of items in dictionary ",dictId,!!
  /* ... */
ResetForNextTime
  IF dictId = -1 {
     SET dictId=##class(%iKnow.Matching.DictionaryAPI).GetDictionaryId(domId,dictname)}
  SET stat=##class(%iKnow.Matching.DictionaryAPI).DropDictionary(domId,dictId)
  IF stat {WRITE "deleted dictionary ",dictId,! }
  ELSE    { WRITE "DropDictionary error ",$System.Status.DisplayError(stat) }
```

The following example creates a the same dictionary as the previous example, except that this dictionary can be used by any domain within the current namespace:

ObjectScript

```
CreateDictionary
  SET dictname="AviationTerms"
  SET dictdesc="A dictionary of aviation terms"
  SET dictId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionary(0,dictname,dictdesc)
  IF dictId=-1 {WRITE "Dictionary ",dictname," already exists",!
                 GOTO ResetForNextTime }
  ELSE {WRITE "created a dictionary ",dictId,!}
PopulateDictionaryItem1
  SET itemId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryItem(0,dictId,
     "aircraft",0_dictId_"aircraft")
    SET term1Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(0,itemId,
     "airplane")
    SET term2Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(0,itemId,
     "helicopter")
PopulateDictionaryItem2
 SET itemId2=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryItemAndTerm(0,dictId,
     "weather",0_dictId_"weather")
    SET i2term1Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(0,itemId2,
     "meteorological information")
    SET i2term2Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(0,itemId2,
     "visibility")
```

```
    SET i2term3Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(0,itemId2,
        "winds")
DisplayDictionary
  SET domId=##class(%iKnow.Domain).GetOrCreateId("mydomain")
  SET stat=##class(%iKnow.Matching.DictionaryAPI).GetDictionaryItemsAndTerms(.result,0,dictId)
  SET i=1
  WHILE $DATA(result(i)) {
      WRITE $LISTTOSTRING(result(i),",",1),!
      SET i=i+1 }
  WRITE "End of items in dictionary ",dictId,!!
   /* ... */
ResetForNextTime
  IF dictId = -1 {
      SET dictId=##class(%iKnow.Matching.DictionaryAPI).GetDictionaryId(0,dictname)}
  SET stat=##class(%iKnow.Matching.DictionaryAPI).DropDictionary(0,dictId)
  IF stat {WRITE "deleted dictionary ",dictId,! }
  ELSE    { WRITE "DropDictionary error ",$System.Status.DisplayError(stat) }
```

16.2.3 Defining a Format Term

The %iKnow.Matching.Formats package provides three simple format classes:

- %iKnow.Matching.Formats.SimpleDateFormat, which matches dates and times in a variety of formats. For a list of the supported date and time formats, refer to the $ZDATETIMEH function in the *Caché ObjectScript Reference*.

- %iKnow.Matching.Formats.SimplePrefixFormat, which matches any entity that begins with a specified prefix string.

- %iKnow.Matching.Formats.SimpleSuffixFormat, which matches any entity that ends with a specified suffix string.

You can create additional format classes as needed.

The following example uses %iKnow.Matching.Formats.SimpleSuffixFormat. It first defines a dictionary containing one item: speed. The "speed" item contains two terms: "excessive speed" and the suffix format term "mph" (miles per hour). This suffix format will match any entity that ends with the suffix "mph", for example "65mph":

ObjectScript

```
  SET domId=##class(%iKnow.Domain).GetOrCreateId("mydomain")
  /* ... */
CreateDictionary
  SET dictname="Traffic"
  SET dictdesc="A dictionary of traffic enforcement terms"
  SET dictId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionary(domId,dictname,dictdesc)
  IF dictId=-1 {WRITE "Dictionary ",dictname," already exists",!
                GOTO ResetForNextTime }
  ELSE {WRITE "created a dictionary ",dictId,!}
CreateDictionaryItemAndTerms
  SET
itemlId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryItem(domId,dictId,"speed",domId_dictId_"speed")

  SET term1Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,item1Id,
          "excessive speed")
  SET term2Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTermFormat(domId,
          item1Id,"%iKnow.Matching.Formats.SimpleSuffixFormat",$LB("mph",0,3))
  WRITE "dictionary=",dictId,!,"item=",item1Id,!,"terms=",term1Id," ",term2Id,!!
   /* ... */
ResetForNextTime
  IF dictId = -1 {
      SET dictId=##class(%iKnow.Matching.DictionaryAPI).GetDictionaryId(domId,dictname)}
  SET stat=##class(%iKnow.Matching.DictionaryAPI).DropDictionary(domId,dictId)
  IF stat {WRITE "deleted dictionary ",dictId,! }
  ELSE { WRITE "DropDictionary error ",$System.Status.DisplayError(stat) }
```

16.2.4 Multiple Formats in a Dictionary Term

You can input dictionary formats directly as part of a dictionary term. This allows you to create a dictionary term containing multiple elements, including one or more format elements, as well as string elements.

To use this feature, you specify a "coded" description of the format as part of the string submitted to the **CreateDictionaryTerm()** method. This coded description has the following format:

```
@@@User.MyFormatClass@@@param1@@@param2@@@
```

This description consists of the full class name of the format class (implementing %iKnow.Matching.Formats.Format), a @@@ separator, and a @@@-delimited list of the format parameters to be passed to the format class. The entire description is delimited with @@@ markers at the beginning and end.

If the format class takes no parameters, or the defaults are to be used, specify the format class name delimited by @@@ markers.

When including this format in a dictionary term string, you must make sure that iKnow will recognize it as a single entity. For examples, the term "was born in @@@User.MyYearFormat@@@" is interpreted as a single entity, but the term "was born in the year @@@User.MyYearFormat@@@" is not.

If iKnow cannot find the specified format class, the @@@ usage is considered intentional and the whole entity is treated as a simple string element.

Using this syntax makes it easier to load dictionaries from files or tables without requiring separate steps or actions for the formats.

16.3 Listing and Copying Dictionaries

The %iKnow.Matching.DictionaryAPI class has a number of methods to count or list existing dictionaries and their items and terms.

The %iKnow.Utils.CopyUtils class has a number of methods to copy a dictionary or all dictionaries from one domain to another.

16.3.1 Listing Existing Dictionaries

The following example lists all of the dictionaries in the domain. For the purpose of demonstration, this example first creates two empty dictionaries, one in English (the default language) and one in French:

ObjectScript

```
  SET domId=##class(%iKnow.Domain).GetOrCreateId("mydomain")
  SET dictname1="Diseases",dictname2="Maladies"
  SET dictdesc1="English disease terms",dictdesc2="French disease terms"
CreateFirstDictionary
  SET dictId1=##class(%iKnow.Matching.DictionaryAPI).CreateDictionary(domId,dictname1,dictdesc1)
  IF dictId1 = -1 {
     SET dictId=##class(%iKnow.Matching.DictionaryAPI).GetDictionaryId(domId,dictname1)
     SET stat=##class(%iKnow.Matching.DictionaryAPI).DropDictionary(domId,dictId)
     IF stat '= 1 { WRITE "DropDictionary error ",$System.Status.DisplayError(stat)
                    QUIT }
     GOTO CreateFirstDictionary }
   ELSE {WRITE "created a dictionary ",dictId1,!}
CreateSecondDictionary
  SET dictId2=##class(%iKnow.Matching.DictionaryAPI).CreateDictionary(domId,dictname2,dictdesc2,"fr")
  IF dictId2 = -1 {
     SET dictId=##class(%iKnow.Matching.DictionaryAPI).GetDictionaryId(domId,dictname2)
     SET stat=##class(%iKnow.Matching.DictionaryAPI).DropDictionary(domId,dictId)
     IF stat '= 1 { WRITE "DropDictionary error ",$System.Status.DisplayError(stat)
                    QUIT }
     GOTO CreateSecondDictionary }
  ELSE {WRITE "created a dictionary ",dictId2,!}

GetDictionaries
  SET stat=##class(%iKnow.Matching.DictionaryAPI).GetDictionaries(.dicts,domId)
  WRITE "get dictionaries status is:",$System.Status.DisplayError(stat),!!
  SET k=1
```

```
WHILE $DATA(dicts(k)) {
    WRITE $LISTTOSTRING(dicts(k)),!
    SET k=k+1 }
WRITE "End of list of dictionaries"
```

GetDictionaries() lists the Id, name, description, and language for each dictionary.

16.3.2 Copying Dictionaries

You can copy dictionaries from one domain to another within the current namespace.

- You can use the **CopyDictionaries()** method of the %iKnow.Utils.CopyUtils class to copy all defined dictionaries in a domain to another domain. By default this method also copies matching profiles from one domain to another.

- You can use the **CopyDictionary()** method of the %iKnow.Utils.CopyUtils class to copy a single defined dictionary in a domain to another domain. By default this method also maps matching profile Ids from one domain to another.

- You can use the **CopyDomain()** method of the %iKnow.Utils.CopyUtils class to copy dictionaries as part of a domain copy operation.

16.4 Extending Dictionary Constructs

Though iKnow only describes simple dictionaries in the Matching API, this does not restrict you from using more advanced tools like ontologies, taxonomies or other more hierarchical constructs. The goal of the Matching API is to provide the hooks for just the matching, rather than yet another generic structure that tries to cover every construct. Therefore, you should just flatten the structure of the ontology or taxonomy you have. By appropriately choosing your dictionary item URIs, you'll be able to reconstruct or interpret the matching results within the context of your ontology or taxonomy.

In the Matching API, the formatting bits are pluggable in the sense that you can provide your own implementation of a class that does for example regular expression matching by implementing the %iKnow.Matching.Formats.Format interface.

17

Smart Matching: Using a Dictionary

Smart Matching means combining the results of the iKnow Indexing process with some external knowledge you have in the form of a dictionary, taxonomy, or ontology. What makes iKnow matching "smart" is that those Indexing results help you judge the quality of a match because they identify which words belong together to form concepts and relations. For example, iKnow can identify if a match for your dictionary term "flu" is actually referring to the concept "flu" or "bird flu" or "no flu symptoms" in your indexed text source. In the second case, which is called a partial match, it is clear the match should or could be treated differently than a full (exact) match where the dictionary term corresponds exactly to the entity in the indexed text source. In the third case, the match is identified as a partial match with negation.

To perform Smart Matching, you must create or acquire a dictionary. Creating and populating a dictionary is described in the previous chapter. Once you have a populated dictionary, you can perform matching operations using the contents of the dictionary.

17.1 How Dictionary Matching Works

iKnow matches each term in the dictionary with the same level construct in the source texts (a Concept term is matched against a Concept in source text; a CRC term is matched against a CRC in source text). These matches can be exact, or can be partial matches. If the match between term and source text is exact, iKnow tags the source text passage with the dictionary item associated with that term. If the match between term and source text is not exact, iKnow scores the degree of match between them. This match scoring involves calculating a match score for each component entity (concept or relation), then, if required, using these entity match scores to calculate the match score for a CRC, path, or sentence. If the scoring of a partial match achieve a configured minimal match score, iKnow tags the source text passage with the dictionary item associated with the term (see the *MinimalMatchScore* property in %iKnow.Matching.MatchingProfile).

17.1.1 Match Scoring

iKnow generates a match score, a floating point number, which is calculated from all of the entity matches detected between the dictionary term and the unit of source text. For an entity match, this match score can range between 0 (no match) and 1 (an exact match). For CRC or path matches, this match includes this range, but can be greater than 1. The algorithm used to calculate this score is complex, but includes the following considerations:

- A full match of an entity can be exact (same words in same order) or scattered (same words in different order). The match score for a scattered match is determined by multiplying 1 (exact match) by the value of the *ScatteredMatchMultiplier* property of the matching profile.

- A partial match of an entity (concept or relation) is assigned a percentage based on the percentage of the source text string that matches the dictionary term.

- In a partial match of an entity, a matching relation is assigned only half as much value as a matching concept. You can change this ratio (for example, to an equal evaluation of relation and concept matches) by setting the *RelationshipScoreMultiplier* property of the matching profile.

- When matching a CRC, path, or sentence, the match scores from the entity matches are added, then divided by the length of the dictionary term, multiplied by the number of matching entities, then multiplied by the *DisorderMultiplier* property, a value representing the degree of disorder (difference in the sequence of entities) between the unit of source text and the dictionary term.

- An entity match score can be modified if the entity is part of a negation (iKnow version 3 and higher). By default, the *NegationMultiplier* property value is 1 which causes the match score calculation to treat positive entities and a negated entities as equivalent. This default is generally recommended. You can set the NegationMultiplier to 0, which causes the match score calculation to skip negated entities. In most cases, a 0 value leads to these matches being skipped altogether, unless it's a composite match with enough non-negated matched entities to get the score above the Minimal-MatchScore threshold. You can also set this property to a value between 0 and 1, which modifies the entity-level match scores for negated entities, causing them to be considered partial matches. For example, a value of 0.5 will halve the entity-level score for a negated entity.

The above is not an exact formula for obtaining a match score. It is provided to show the principal considerations used when iKnow calculates a match score.

iKnow provides a matching profile (%iKnow.Matching.MatchingProfile), which consists of the numeric properties mentioned above, and others. iKnow uses this matching profile when calculating a match score. iKnow provides default property values for the matching profile. Unless otherwise specified, the default matching profile is assigned to each dictionary. This default matching profile provides accurate matching for most applications. Creating and assigning a custom matching profile is described later in this chapter.

17.2 Matching A String

You can use the %iKnow.Matching.MatchingAPI class to perform matches between a text string and a populated dictionary (or multiple dictionaries). There are two types of string matches:

- Matching an entity-length string

- Matching a string containing multiple entities. This string may contain one or more sentences.

17.2.1 Matching an Entity String

The **GetDictionaryMatches()** method matches a single-entity string against a dictionary and returns the match items. Because this method treats the string as a single entity, it can provide very specific match information. Because **GetDictionaryMatches()** takes a string variable, you do not have to index a single-entity string to match it against a dictionary.

ObjectScript

```
SET domn="entitytestdomain"
IF (##class(%iKnow.Domain).NameIndexExists(domn))
    { SET domo=##class(%iKnow.Domain).NameIndexOpen(domn)
      SET domId=domo.Id }
ELSE {
    SET domo=##class(%iKnow.Domain).%New(domn)
    DO domo.%Save()
    SET domId=domo.Id }
/* ... */
CreateDictionary
  SET dictname="AviationTerms"
  SET dictdesc="A dictionary of aviation terms"
```

```
    SET dictId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionary(domId,dictname,dictdesc)
    IF dictId=-1 {WRITE "Dictionary ",dictname," already exists",!
                 GOTO ResetForNextTime }
    ELSE {WRITE "created a dictionary ",dictId,!}
PopulateDictionary
    SET itemId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryItem(domId,dictId,
        "aircraft",domId_dictId_"aircraft")
    SET term1Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId,
        "airplane")
    SET term2Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId,
        "single-engine airplane")
    SET term3Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId,
        "helicopter")
DisplayDictionary
    SET stat=##class(%iKnow.Matching.DictionaryAPI).GetDictionaryItemsAndTerms(.result,domId,dictId)
    SET i=1
    WHILE $DATA(result(i)) {
        WRITE $LISTTOSTRING(result(i),",",1),!
        SET i=i+1 }
    WRITE "End of items in dictionary ",dictId,!!
DoMatching
    SET mystring="A small single-engine two-person airplane cabin"
    SET stat=##class(%iKnow.Matching.MatchingAPI).GetDictionaryMatches(.num,domId,mystring,$LB(dictId))
    IF stat'=1 {WRITE "get matches status is:",$System.Status.DisplayError(stat),!
            QUIT }
    WRITE "The string is: ",mystring,!
    WRITE "The matches are:",!
    SET j=1
    WHILE $DATA(num(j)) {
        WRITE "match number ",j," is ",$LISTTOSTRING(num(j)),!
        SET j=j+1 }
    WRITE "End of match items for dictionary ",dictId,!!
ResetForNextTime
    IF dictId = -1 {
        SET dictId=##class(%iKnow.Matching.DictionaryAPI).GetDictionaryId(domId,dictname)}
    SET stat=##class(%iKnow.Matching.DictionaryAPI).DropDictionary(domId,dictId)
    IF stat {WRITE "deleted dictionary ",dictId,! }
```

GetDictionaryMatches() provides a matched word bit map that shows the match of each word in the string with the dictionary term. For example, 00101 shows the match of the dictionary term "single-engine airplane" to the string "A small single-engine two-person airplane / cabin". The bit map stops when the match completes, so there is no bit for the word "cabin". In this example, the isScattered boolean is 0 because the words "single-engine" and "airplane" are in the same order in the dictionary term and in the string.

Each dictionary match returns a list of match elements. The match in the previous example returns a list of match elements as follows (your id numbers may differ):

Position	Meaning	Value in Example
1	dictionary Id	6
2	item Id	5
3	dictionary item URI (domain Id + dict Id + item value)	26aircraft
4	term Id	13
5	term value	single-engine airplane
6	element Id (same as term Id)	13
7	type of match (term, format, or unknown)	term
8	match score	.333333
9	matched word bits	00101
10	is scattered boolean	0
11	format output	null

17.2.2 Matching a Sentence String

The **GetMatchesBySource()** method matches a multiple-entity string against a dictionary and returns the match items. (Commonly such a string is sentence-length (or longer)). You must first index the string, then match it against the dictionary. The following example matches a string against the AviationTerms dictionary:

ObjectScript

```
DomainCreateOrOpen
  SET dname="onestringdomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        GOTO LoadString }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { GOTO LoadString }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
LoadString
  SET domId=domoref.Id
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
  SET ^mystring="A single-engine airplane reported poor visibility and strong gusty winds. "_
                "No winds were predicted by local airport ground personnel."
  DO myloader.BufferSource("ref",^mystring)
  DO myloader.ProcessBuffer()
GetExtId
  DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId,1,20)
  SET i=1
  WHILE $DATA(result(i)) {
     SET extId = $LISTGET(result(i),2)
     SET i=i+1 }
CreateDictionary
  SET dictname="AviationTerms"
  SET dictdesc="A dictionary of aviation terms"
  SET dictId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionary(domId,dictname,dictdesc)
  IF dictId=-1 {WRITE "Dictionary ",dictname," already exists",!
                GOTO ResetForNextTime }
  ELSE {WRITE "created a dictionary ",dictId,!}
PopulateDictionaryItem1
  SET itemId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryItem(domId,dictId,
     "aircraft",domId_dictId_"aircraft")
    SET term1Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId,
     "single-engine airplane")
    SET term2Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId,
     "helicopter")
PopulateDictionaryItem2
 SET itemId2=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryItemAndTerm(domId,dictId,
     "weather",domId_dictId_"weather")
    SET i2term1Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId2,
     "strong winds")
    SET i2term2Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId2,
     "visibility")
    SET i2term3Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(domId,itemId2,
     "winds")
DisplayDictionary
  SET stat=##class(%iKnow.Matching.DictionaryAPI).GetDictionaryItemsAndTerms(.result,domId,dictId)
  SET i=1
  WHILE $DATA(result(i)) {
     WRITE $LISTTOSTRING(result(i),",",1),!
     SET i=i+1 }
  WRITE "End of items in dictionary ",dictId,!!
DoMatching
  SET stat=##class(%iKnow.Matching.MatchingAPI).GetMatchesBySource(.num,domId,extId,$LB(dictId))
  IF stat'=1 {WRITE "get matches status is:",$System.Status.DisplayError(stat),!
             QUIT }
  WRITE "The string is: ",^mystring,!
  WRITE "The matches are:",!
  SET j=1
  WHILE $DATA(num(j)) {
     WRITE "match ",j,": ditem ",$LISTGET(num(j),4),
           " dterm ",$LISTGET(num(j),5),
           " matchscore ",$LISTGET(num(j),8),
           " negated? ",$LISTGET(num(j),15),!
             SET j=j+1 }
  WRITE "End of match items for dictionary ",dictId,!!
```

```
ResetForNextTime
  IF dictId = -1 {
     SET dictId=##class(%iKnow.Matching.DictionaryAPI).GetDictionaryId(domId,dictname)}
  SET stat=##class(%iKnow.Matching.DictionaryAPI).DropDictionary(domId,dictId)
  IF stat {WRITE "deleted dictionary ",dictId,! }
  ELSE    { WRITE "DropDictionary error ",$System.Status.DisplayError(stat) }
```

The following is a match between "No winds" in the string and dictionary term "winds" in item "weather". (This match string is returned in %List format, here shown as a comma-separated string):

5,6,12,26weather,33,0,6,.5,1,0,1,1,1,1,1. These element values are explained below:

Position	Meaning	Value in Example
1	match number	5
2	dictionary Id	6
3	item Id	12
4	dictionary item URI (domain Id + dict Id + item value)	26weather
5	term Id	33
6	target type	0
7	target Id	6
8	match score	.5
9	matching concept count	1
10	matching relation count	0
11	partial match count	1
12	first matched position in path	1
13	last matched position in path	1
14	is ordered	1
15	negated entity count	1

17.3 Matching Sources

The **GetTotalItemScoresBySource()** method matches a source against a dictionary and returns the match scores for each dictionary item. The following two examples use a domain-independent dictionary. In the dictionary definition, and in any %iKnow.Matching.DictionaryAPI method referencing the dictionary, specify the domain Id as 0. In any %iKnow.Matching.MatchingAPI method using the dictionary within a domain, specify the dictionary Id as a negative number. Hence, SET dictId=-^mydictId.

The following example matches all of the sources in the domain against the AviationTermsND dictionary and returns each source's match scores for each dictionary item. You must create the dictionary before running this program.

ObjectScript

```
CreateDictionary
  SET ^mydictname="AviationTermsND"_$HOROLOG
  SET dictdesc="A dictionary of aviation terms"
  SET ^mydictId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionary(0,^mydictname,dictdesc)
  IF ^mydictId=-1 {WRITE "Dictionary ",^mydictname," already exists",!
             QUIT }
```

```
    ELSE {WRITE "created a dictionary ",^mydictId,!}
PopulateDictionaryItem1
  SET itemId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryItem(0,^mydictId,
      "aircraft",0_^mydictId_"aircraft")
    SET term1Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(0,itemId,
      "single-engine airplane")
    SET term2Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(0,itemId,
      "helicopter")
PopulateDictionaryItem2
 SET itemId2=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryItemAndTerm(0,^mydictId,
      "weather",0_^mydictId_"weather")
    SET i2term1Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(0,itemId2,
      "strong winds")
    SET i2term2Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(0,itemId2,
      "visibility")
    SET i2term3Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(0,itemId2,
      "winds")
PopulateDictionaryItem3
  SET itemId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryItem(0,^mydictId,
      "flight plan",0_^mydictId_"flight plan")
    SET term3Id=##class(%iKnow.Matching.DictionaryAPI).CreateDictionaryTerm(0,itemId,
      "flight plan")
DisplayDictionary1
  SET stat=##class(%iKnow.Matching.DictionaryAPI).GetDictionaryItemsAndTerms(.result,0,^mydictId)
  WRITE "Status is: ",stat,!
  SET i=1
  WHILE $DATA(result(i)) {
      WRITE $LISTTOSTRING(result(i),",",1),!
      SET i=i+1 }
  WRITE "End of items in dictionary ",^mydictId,!!
```

ObjectScript

```
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
            GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT Top 10 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
  SET datiaflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
DoMatching
  SET dictId=-^mydictId
  SET num=0
  FOR j=1:1:10 {SET extId=##class(%iKnow.Queries.SourceAPI).GetExternalId(domId,j)
   SET mstat=##class(%iKnow.Matching.MatchingAPI).GetTotalItemScoresBySource(.mresult,domId,
                    extId,$LB(dictId))
    IF mstat '=1 {WRITE "End of sources",!  QUIT }
    SET k=1
    IF $DATA(mresult(k))=0 {WRITE "no dictionary matches for this source",!}
    ELSE {
    WHILE $DATA(mresult(k)) {
        WRITE $PIECE($LISTTOSTRING(mresult(k)),",",2)," "
        WRITE $PIECE($LISTTOSTRING(mresult(k)),",",4)
        WRITE " matches: ",$PIECE($LISTTOSTRING(mresult(k)),",",6)
        WRITE " score: ",$PIECE($LISTTOSTRING(mresult(k)),",",7),!
        SET k=k+1 }
    }
```

```
        SET srcname=$PIECE($PIECE(extId,":",3,4),"\",$l(extId,"\"))
        WRITE "End of ",srcname," match items for dictionary ",dictId,!!
    }
```

The **GetMatchesBySource()** method matches each source against a dictionary and returns the match items.

The following example matches all of the sources in the domain against the domain-independent AviationTermsND dictionary (defined above) and returns each source's match items, with match score and negation (if present):

ObjectScript

```
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
            QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
  SET myquery="SELECT Top 10 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
  SET idfld="UniqueVal"
  SET grpfld="Type"
  SET dataflds=$LB("NarrativeFull")
UseLister
  SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
  SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
DoMatching
  SET dictId=-^mydictId
  FOR j=1:1:10 {SET extId=##class(%iKnow.Queries.SourceAPI).GetExternalId(domId,j)
      SET stat=##class(%iKnow.Matching.MatchingAPI).GetMatchesBySource(.num,domId,extId,$LB(dictId))
      IF stat'=1 {WRITE "get matches status is:",$System.Status.DisplayError(stat),!
            QUIT }
      WRITE "The matches are:",!
      SET k=1
      WHILE $DATA(num(k)) {
        IF $LISTGET(num(k),15)>0 {WRITE "neg. "}
        WRITE "match ",k,": ditem ",$LISTGET(num(k),4),
            " dterm ",$LISTGET(num(k),5),
            " matchscore ",$LISTGET(num(k),8),!
            SET k=k+1 }
    WRITE !,"Next Source",!
    }
  WRITE "End of match items for dictionary ",dictId,!
```

17.4 Defining a Matching Profile

Whether a dictionary term and a unit of source text constitute a match is determined by the matching profile properties found in %iKnow.Matching.MatchingProfile. There are eight properties that together determine whether something is a match or not. All of these properties take an appropriate default value. By default, every dictionary is assigned the default matching profile.

You can create a custom matching profile with one or more properties whose value differs from the default. You can then assign this custom matching profile to a dictionary. Any properties not specified in the custom matching profile take default values. You can create any number of custom matching profiles. The same matching profile can be applied to multiple dictionaries. You can define the matching profile to be specific to a domain, or to be available to all domains in the namespace.

The following example creates three custom matching profiles. The first is specific to a domain and specifies the optional matching profile name; iKnow assigns it a positive integer Id. The second and third are available to all domains in the namespace; iKnow assigns each of them a negative integer Id. Because the third specifies the optional matching profile name, it must specify 0 as a placeholder for the domain Id parameter:

ObjectScript

```
   SET domn="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(domn))
     { SET domo=##class(%iKnow.Domain).NameIndexOpen(domn)
       SET domId=domo.Id }
  ELSE {
     SET domo=##class(%iKnow.Domain).%New(domn)
     DO domo.%Save()
     SET domId=domo.Id }
MatchingProfile
  SET domprof=##class(%iKnow.Matching.MatchingProfile).%New(domId,"mydomainCMP")
     WRITE "profile Id=",domprof.ProfileId,!
     WRITE "profile domain=",domprof.DomainId,!
     WRITE "profile name=",domprof.Name,!!
  SET allprof1=##class(%iKnow.Matching.MatchingProfile).%New()
     WRITE "profile Id=",allprof1.ProfileId,!
     WRITE "profile domain=",allprof1.DomainId,!
     WRITE "profile name=",allprof1.Name,!!
 SET allprof2=##class(%iKnow.Matching.MatchingProfile).%New(0,"namespaceCMP")
     WRITE "profile Id=",allprof2.ProfileId,!
     WRITE "profile domain=",allprof2.DomainId,!
     WRITE "profile name=",allprof2.Name
```

The following example shows how to define a custom matching profile and assign it to a dictionary. It specifies the custom matching profile instance oref (in this case, customprofile) to the **CreateDictionary()** method to override the default.

ObjectScript

```
#include %IKPublic
   SET domn="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(domn))
     { SET domo=##class(%iKnow.Domain).NameIndexOpen(domn)
       SET domId=domo.Id }
  ELSE {
     SET domo=##class(%iKnow.Domain).%New(domn)
     DO domo.%Save()
     SET domId=domo.Id }
CustomizeMatchingProfile
  SET customprofile=##class(%iKnow.Matching.MatchingProfile).%New()
     WRITE "MinimalMatchScore initial value=",customprofile.MinimalMatchScore,!
     SET customprofile.MinimalMatchScore=".4"
     WRITE "MinimalMatchScore custom value=",customprofile.MinimalMatchScore,!
CreateDictionary
  SET dictId=##class(%iKnow.Matching.DictionaryAPI).CreateDictionary(domId,"mydict","","en",customprofile)

  WRITE "created a dictionary with ID=",dictId,!!
  /* . . . */
CleanUpForNextTime
  SET stat=##class(%iKnow.Matching.DictionaryAPI).DropDictionary(domId,dictId)
  IF stat {WRITE "Dropped the dictionary"}
  ELSE {WRITE "DropDictionary error",$System.Status.DisplayError(stat)}
```

17.4.1 Matching Profile Properties

Whether the terms in a dictionary and a unit of source text constitute a match is determined by the matching profile property values you define in your custom matching profile. These properties determine the match score for each unit of text and the threshold which specifies the minimum match score to report as a match. Because the contents of a dictionary tends to vary depending on the use case, you may wish to customize one or more matching profile properties for your dictionary. Changing any of these properties may dramatically change the number of matches reported. Therefore, you might want to experiment with changes to property values while testing on a small subset of your data before registering the matching profile to match the dictionary to the whole dataset.

The meaning of the properties in %iKnow.Matching.MatchingProfile are explained in the class documentation.

17.4.1.1 MinimalMatchScore

The most common matching profile property to tune is the *MinimalMatchScore*. This property value is the lower threshold for matches to be saved. You can set it to a fractional value between 1 (only perfect matches saved) and 0 (all potential matches saved); the default is 0.33.

- By increasing this property value you can filter out low-quality matches if you have an excessive number of match candidates. This would be appropriate if your dictionary is fairly generic and contains many common terms. You may only wish to report CRCs and paths that are a very close match to several of these common terms.

- By decreasing this property value you can increase the number of match candidates. This would be appropriate if your dictionary is highly specific, consisting only of critical terms that should be flagged at all times. You may wish to report CRCs and paths that loosely match with a dictionary of technical terms, so as to avoid missing a loose, but significant, match.

Setting MinimalMatchScore to 0 returns all possible match results. If you're starting to work on matching a new dictionary to a small subset of your data, you can start by setting MinimalMatchScore to 0, then gradually increase it, filtering out low-quality matches, until you get a reasonable number of results.

17.4.2 Domain Default Matching Profile

You can specify a different domain-wide default matching profile by setting the MAT:DefaultProfile ($$$IKPMATDE-FAULTPROFILE) domain parameter. (The "MAT:" prefix indicates a domain parameter specific to matching operations.) The following example defines a domain-specific custom matching profile, then assigns it as the default matching profile for the `mydomain` domain:

ObjectScript

```
#include %IKPublic
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { SET domo=##class(%iKnow.Domain).NameIndexOpen(dname) }
  ELSE
      { SET domo=##class(%iKnow.Domain).%New(dname) DO domo.%Save()}
   SET domId=domo.Id
   WRITE "domain ",dname," has Id ",domId,!!
CreateProfile
  SET domprof=##class(%iKnow.Matching.MatchingProfile).%New(domId,"mydomainCMP")
    WRITE "profile Id=",domprof.ProfileId,!
    WRITE "profile domain assignment=",domprof.DomainId,!
    WRITE "profile name=",domprof.Name,!
    WRITE "profile properties:",!
    WRITE "default MinimalMatchScore=",domprof.MinimalMatchScore,!
    SET domprof.MinimalMatchScore=.27
    WRITE "changed to MinimalMatchScore=",domprof.MinimalMatchScore,!!
    DO domprof.%Save()
MakeProfileDomainDefault
    SET str=domo.GetParameter($$$IKPMATDEFAULTPROFILE,.dpin)
    WRITE "domain ",domId," DefaultProfile before SET=",dpin,!
    SET sc=domo.SetParameter($$$IKPMATDEFAULTPROFILE,"mydomainCMP")
    IF sc=1 {
    DO domo.GetParameter($$$IKPMATDEFAULTPROFILE,.dpout)
    WRITE "domain ",domId," DefaultProfile after SET=",dpout,! }
    ELSE {WRITE "SetParameter error",! }
CleanUp
  DO ##class(%iKnow.Domain).%DeleteId(domo.Id)
  WRITE "All done"
```

The following example defines a namespace custom matching profile (not specific to a domain), then assigns it as the default matching profile for the `mydomain` domain. Note the `0:` preface to the profile name in `SetParameter($$$IKPMATDEFAULTPROFILE,"0:nodomainCMP")`:

ObjectScript

```
#include %IKPublic
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
     { SET domo=##class(%iKnow.Domain).NameIndexOpen(dname) }
  ELSE
     { SET domo=##class(%iKnow.Domain).%New(dname) DO domo.%Save()}
   SET domId=domo.Id
   WRITE "domain ",dname," has Id ",domId,!!
CreateProfile
  SET domprof=##class(%iKnow.Matching.MatchingProfile).%New(0,"nodomainCMP")
     WRITE "profile Id=",domprof.ProfileId,!
     WRITE "profile domain assignment=",domprof.DomainId,!
     WRITE "profile name=",domprof.Name,!
     WRITE "profile properties:",!
     WRITE "default MinimalMatchScore=",domprof.MinimalMatchScore,!
     SET domprof.MinimalMatchScore=.27
     WRITE "changed to MinimalMatchScore=",domprof.MinimalMatchScore,!!
     DO domprof.%Save()
MakeProfileDomainDefault
     SET str=domo.GetParameter($$$IKPMATDEFAULTPROFILE,.dpin)
     WRITE "domain ",domId," DefaultProfile before SET=",dpin,!
     SET sc=domo.SetParameter($$$IKPMATDEFAULTPROFILE,"0:nodomainCMP")
     IF sc=1 {
     DO domo.GetParameter($$$IKPMATDEFAULTPROFILE,.dpout)
     WRITE "domain ",domId," DefaultProfile after SET=",dpout,! }
     ELSE {WRITE "SetParameter error",! }
CleanUp
   DO ##class(%iKnow.Domain).%DeleteId(domo.Id)
   WRITE "All done"
```

You can also set individual domain parameters that influence matching operations.

17.4.2.1 Matching Single Relations

By default, the iKnow matching algorithm skips dictionary terms that match a single relation entity. For example, if you have a dictionary term "to", iKnow does not attempt to match the entity "goes to" in the sentence "Pete goes to work". This default optimizes matching performance when your dictionary primarily contains and targets concepts.

However, if your dictionary deliberately targets single relation elements, you can change this domain parameter default by setting the MAT:SkipRelations ($$$IKPMATSKIPRELS) domain parameter to 0. This causes all entities to be matched against all dictionary terms, regardless of the type of entity.

This option is set as a domain parameter — not a matching profile property — because this skipping step occurs at a time when it is not yet known what the matched terms might be, and hence could not apply any dictionary-specific profiles. Therefore, it is implemented as a domain parameter, ensuring it applies to all dictionaries within the domain.

17.4.2.2 Other Matching Operation Domain Parameters

Several of the settable domain parameters can influence matching operations: MAT:SkipRelations, MatchScoreMargin, FullMatchOnly, and EntityLevelMatchOnly. If you change these domain parameters, any sources matched using a previous domain parameter setting will have to be explicitly re-matched to reflect the new domain parameter value.

For further information on these domain parameters, refer to the Domain Parameters appendix to this manual.

18

User Interfaces

The iKnow technology does not have a default user interface. This chapter describes a few sample user interfaces provided with iKnow. These can provide a convenient starting point when developing a query interface specific to your use of iKnow.

These sample user interfaces are found in the %iKnow.UI package, and are implemented as Zen pages, as described in the *InterSystems Class Reference*.

18.1 How to Display iKnow User Interfaces

1. from the Caché Cube, access Studio.

2. from the File drop-down menu, select Change Namespace. Select the %SYS namespace. Click OK.

3. from the File drop-down menu, select Open.

4. from the Open window, make sure that Include System Items is checked. Then select %iKnow->UI->and the desired user interface. Click Open. The source code for the user interface appears in the Class Editor (the main Studio window).

5. click the View Web Page icon (the planet Earth icon). The user interface displays in your default browser. (Note that abstract classes cannot be displayed.)

For further details, refer to the Using Studio manual.

18.2 Abstract Portal

This is the superclass for all other portals and pages in the %iKnow.UI package. It groups a lot of reusable materials such as handling of domain ID, the "selected" source ID, metadata filters and paging for iKnow query-driven tables and groups. It is abstract, and cannot be run by itself, but should not impose restrictions on subclasses or component names, as long as the corresponding panes (optDomainPane, txtTermPane, optSourcePane and filterPane) are used.

The %iKnow.UI.AbstractPortal class has five subclasses that fall into two types:

- The Loading Wizard, which provides a source file management interface.

- The query portal interfaces, which provide different displays of source data, showing iKnow indexing and dictionary matching.

18.3 Abstract Source Viewer

This class extends the AbstractPortal superclass. It contains the **ProcessInput()** and **DeleteCurrentVirtualSource()** methods. It is abstract, and cannot be run by itself.

18.4 Loading Wizard

A management and maintenance interface that allows a user to easily choose (or manage) Domain and Configuration objects and then load files from a filesystem directly into a domain using %iKnow.Source.File.Lister. It also provides functionality to load metadata values from a CSV (comma-separated values) file with rows corresponding to files previously loaded through the File Lister. This operation automatically creates previously nonexistent metadata fields on the fly.

You can use the Caché Management Portal to access the Loading Wizard. From the **System Explorer** option, go to **iKnow**, then **Domain Settings** (**[System] > [iKnow Domains]**). Select the **Sample Tools** tab, then click the **Loading Wizard** button. Once in the Loading Wizard, follow the directions on the screen.

For further details on Domain and Configuration objects, refer to the "Setting Up an iKnow Environment" chapter. For further details on listing and loading, refer to the "Loading Text Data into iKnow" chapter.

18.5 Knowledge Portal

This is a sample Zen page query display interface with broad application. It shows a wealth of information about the various language elements identified by iKnow, including entities, CRCs, CCs and paths, providing a contextual at-a-glance view of what's in your data. The generic filters option allows for easily selecting subsets of a domain based on metadata criteria and the summary option provides quick access to the contents of the sources themselves. This interface provides a sample of how iKnow Smart Indexing can be used to quickly overview and navigate a large set of documents.

The Knowledge Portal offers straightforward examples of calling the different iKnow Query APIs from Zen, including top entities, similar entities, and related entities, with the frequency and spread for each.

The Knowledge Portal supports the use of black lists.

For a full description of features, refer to Knowledge Portal in the "iKnow Architect" chapter of this book.

18.6 Basic Portal

This sample Zen page query display interface is a simplified version of the Knowledge Portal. It displays entities and sources only. It does not display CRCs, CCs and paths. It provides filtering and summary capabilities, but by default it shows the full text of the source.

The Basic Portal supports the use of black lists.

18.7 Indexing Results

This sample Zen page query display interface looks at the Smart Indexing results for a single document to verify the correctness of the analysis done by the iKnow analysis engine. It shows how iKnow cuts up each sentence into a sequence of concepts (bold and highlighted), relations (underlined), and non-relevants (italic). The page also shows a frequency-sorted list of detected Concepts and CRCs. This page provides an option for loading input manually. This page offers an example of how **%iKnow.Queries.SentenceAPI.GetParts()** can be used for custom highlighting.

For a full description of features, refer to Indexing Results in the "iKnow Architect" chapter of this book.

For definitions of concepts, relations, CRCs, and non-relevants, refer to the "Conceptual Overview" chapter.

18.8 Matching Results

This sample Zen page query display interface gives for each document an overview of the different matching results in the document. It allows for easy browsing through match results against a dictionary and allows you to display the details of each individual match. It uses **%iKnow.Matching.MatchingAPI.GetHighlightedSentences()** to display the text with dictionary matches highlighted in color, and **%iKnow.Matching.MatchingAPI.GetMatchElements()** to display the details of a specific match including the dictionary name, its item and term, the match score, type of match and the entity matched. This interface provides a sample of how iKnow Smart Matching can be used to combine predefined information with Smart Indexing results.

For further details on matching, refer to the "Smart Matching: Using a Dictionary" chapter.

19

iKnow Tools

The primary iKnow interface for creating a domain and populating it with data is the iKnow Architect. The principal interface for analyzing iKnow data is through APIs containing class object methods and properties which may be invoked from ObjectScript programs. The iKnow tools described in this chapter are meant to assist in maintaining iKnow functionality and in testing and examining iKnow data. All functionality provided here is also available using the iKnow APIs.

The iKnow tools described in this chapter are:

- Shell Interface
- Data Upgrade Utility

19.1 iKnow Shell Interface

The iKnow Shell can be used to return information about existing domains and indexed sources.

All iKnow operations occur within a namespace. Therefore, before invoking the iKnow Shell, you should specify the desired namespace using the ZNSPACE command.

From the Terminal you can activate the iKnow Shell interface as follows:

```
USER>DO $System.iKnow.Shell()
```

This returns the iKnow Shell prompt. Typing ? at the iKnow Shell prompt displays a list of the iKnow Shell commands. Typing an iKnow Shell command followed by space and a ? displays information about that command.

The iKnow Shell commands and options are case-sensitive and must be specified in all lowercase letters.

19.1.1 List, Show, and Summarize Sources

The following iKnow Shell example makes use of an existing domain named "mydomain" in the current namespace. The domain "mydomain" contains 100 sources. The `use domain mydomain` command specifies using "mydomain", making it the current domain for the iKnow Shell. The `list source` command lists the sources in the current domain. By default, a `list` command lists the first page, with a page size of ten items. To list additional pages, you can specify the > command, as shown in this example. To change the page size to 20 items, specify `use pagesize 20`.

The `show source 92` command lists the contents of source 92, broken into sentences. It lists the first page of 10 sentences; you can use the > command to list additional pages of sentences. The text of each sentence is prefaced by its sentence Id, and is followed by a boolean indicating whether the Shell had to truncate the displayed sentence text. (For the purpose of this example, I have added line wraps to the sentence texts.) The `show summary 92 6` command summarizes the contents

of source 92 to 6 sentences, and displays these 6 sentences. If the summary number is larger than the number of sentences in the source, `show summary` displays all of the sentences in the source.

```
SAMPLES>DO $System.iKnow.Shell()

Welcome to the iKnow shell
Type '?' for help
Type 'quit' to exit

iKnow> use domain mydomain
Current domain: mydomain (1)
iKnow> list source
 srcId          externalId
   100     :SQL:Accident:96
    99     :SQL:Accident:98
    98     :SQL:Accident:94
    97    :SQL:Accident:100
    96     :SQL:Accident:80
    95     :SQL:Accident:99
    94     :SQL:Accident:95
    93     :SQL:Accident:88
    92     :SQL:Accident:97
    91     :SQL:Incident:90
iKnow> >
 srcId          externalId
    90     :SQL:Accident:93
    89     :SQL:Accident:92
    88     :SQL:Accident:91
    87     :SQL:Accident:85
    86     :SQL:Accident:86
    85     :SQL:Accident:89
    84     :SQL:Accident:87
    83     :SQL:Accident:83
    82     :SQL:Accident:78
    81     :SQL:Accident:82
iKnow> show source 92
 sentId       sentenceValue
sentenceIsTruncated
   5090      On March 7, 2001, about 1500 Alaska standard time, a wheel/ski equipped
             Cessna 180 airplane, N9383C, sustained substantial damage during takeoff from
             a snow-covered area at Ophir, Alaska.                                         0
   5091      The airplane was being operated as a visual flight rules (VFR) cross-country
             personal flight to McGrath, Alaska, when the accident occurred.               0
   5092      The airplane was operated by the pilot.                                       0
   5093      The commercial certificated pilot, and the sole passenger, were not injured.  0
   5094      Visual meteorological conditions prevailed.                                   0
   5095      During a telephone conversation with the National Transportation Safety Board
             (NTSB) investigator-in-charge (IIC), on March 8, 2001, the pilot reported he
             landed near Ophir earlier in the day.                                         0
   5096      When he was planning to depart, the surface of the snow had become crusty.    0
   5097      The pilot said he began a takeoff run toward the south, but the airplane
             did not become airborne until it was within about 50 yards from several trees. 0
   5098      During the initial climb, the left horizontal stabilizer collided with a
             spruce tree about 25 feet above the ground.                                   0
   5099      The airplane began a descending left turn toward the ground, and collided
             with several trees while the pilot was making an emergency landing.           0
iKnow> show summary 92 6
 sentId       sentenceValue
sentenceIsTruncated
   5090      On March 7, 2001, about 1500 Alaska standard time, a wheel/ski equipped
             Cessna 180 airplane, N9383C, sustained substantial damage during takeoff from
             a snow-covered area at Ophir, Alaska.                                         0
   5091      The airplane was being operated as a visual flight rules (VFR) cross-country
             personal flight to McGrath, Alaska, when the accident occurred.               0
   5095      During a telephone conversation with the National Transportation Safety Board
             (NTSB) investigator-in-charge (IIC), on March 8, 2001, the pilot reported he
             landed near Ophir earlier in the day.
   5097      The pilot said he began a takeoff run toward the south, but the airplane
             did not become airborne until it was within about 50 yards from several trees. 0
   5099      The airplane began a descending left turn toward the ground, and collided
             with several trees while the pilot was making an emergency landing.           0
   5100      The airplane received damage to the left main landing gear, the wings,
             and the left stabilizer.                                                      0
iKnow> quit
Bye bye

SAMPLES>
```

19.1.2 Filter Sources

The following iKnow Shell example makes use of an existing domain named "mydomain" in the current namespace. The domain "mydomain" contains 100 sources. The `use domain mydomain` command specifies using "mydomain", making it the current domain for the iKnow Shell. The `list source` command lists the first 10 sources in the current domain. The `filter source 92 94 97 as myfilter` command defines a filter named "myfilter" which filters out all sources except those specified by source Id. The `use filter myfilter` command establishes "myfilter" as the current filter. Now when the iKnow Shell issues a `list source` command, it applies "myfilter" and lists only the three sources specified in "myfilter":

```
SAMPLES>DO $System.iKnow.Shell()

Welcome to the iKnow shell
Type '?' for help
Type 'quit' to exit

iKnow> use domain mydomain
Current domain: mydomain (1)
iKnow> list source
 srcId           externalId
   100     :SQL:Accident:96
    99     :SQL:Accident:98
    98     :SQL:Accident:94
    97     :SQL:Accident:100
    96     :SQL:Accident:80
    95     :SQL:Accident:99
    94     :SQL:Accident:95
    93     :SQL:Accident:88
    92     :SQL:Accident:97
    91     :SQL:Incident:90
iKnow> filter source 92 94 97 as myfilter
iKnow> use filter myfilter
Current filter: myfilter

iKnow> list source
 srcId           externalId
    97     :SQL:Accident:100
    94     :SQL:Accident:95
    92     :SQL:Accident:97
iKnow> quit
Bye bye

SAMPLES>
```

After a filter has been applied, a subsequent `use filter filtername` replaces the current filter with the new filter. To disable a filter, specify `use filter 0`.

19.2 iKnow Data Upgrade Utility

Each version of the iKnow data structures is assigned a system version number. Each iKnow domain is assigned a *Version* property value. All new domains are created with the same Version property as the current system version. Therefore, these two integer values are usually the same.

- When you update to a newer version of Caché, the iKnow data structures system version increments if iKnow indexing has changed. Therefore, updating to a newer version of Caché does not necessarily increment the iKnow data structures system version.

- When you create a domain, it always takes the current data structures system version as its Version property value. Therefore, existing domains created under an earlier system version have the Version property value of that earlier system version.

The following are the Caché versions, the corresponding iKnow system versions, and the principal changes to the indexing process:

Caché 2012.1	1	Initial iKnow indexing.
Caché 2012.2	2	Dominance and Proximity indexing added.
Caché 2013.1	3	Negation attribute indexing added.
Caché 2014.1 and 2015.1	4	Domain-independent dictionaries and blacklists.
Caché 2015.2 and subsequent	5	Stemming indexing added. Sentiment attribute indexing added.

You can use the **GetCurrentSystemVersion()** method of the %iKnow.Domain class to determine the iKnow data structures system version for the current Caché instance. You can use the **GetAllDomains** query to list all domains with their domain Version numbers, as shown in the Listing All Domains section of the "Setting Up the iKnow Environment" chapter.

If the iKnow data structures system version does not match the domain Version, these older iKnow domains cannot take advantage of the new iKnow features and performance improvements introduced with this new system version. Older domains will remain operational, but cannot take advantage of new iKnow data structure features until you upgrade the domain. Upgrading a domain increments its Version property. This upgrade operation requires the automatic re-indexing of the domain data. It does not require access to the original source texts. Each domain must be upgraded individually.

To upgrade a domain, use the **UpgradeDomain()** method of the %iKnow.Utils.UpgradeUtils class. Further details are provided in the *InterSystems Class Reference* documentation.

Note that the re-indexing that occurs when you upgrade a domain changes the domain Id, but does not change the domain name. Thus upgrading a domain may, in some cases, require changes to programs that reference the domain by a specific domain Id integer. For this reason, a domain should always be referenced by its Id property (or domain name). Coding practices that reference a domain by a literal integer Id value should be avoided.

19.2.1 Notes on Specific Version Upgrades

19.2.1.1 To Version 2

Upgrading a domain from iKnow data structures Version 1 to Version 2 substantially increases the space requirements for permanent and temporary globals. The space requirements for permanent globals increase from 8.7 times to 20 times the size of the original source texts. The space requirements for temporary globals increase from 10.7 times to 20 times the size of the original source texts. The space requirements for the cachetemp directory increase from 1.25 times to 4 times the size of the original source texts.

For this reason, you should only upgrade a large iKnow domain from iKnow data structures Version 1 to Version 2 if you have the available space and need the new iKnow data structure features (such as semantic dominance and semantic proximity).

19.2.1.2 To Version 4

Upgrading your system to Caché 2014.1 (or greater) automatically converts system-wide domain parameters. System-wide domain parameters defined prior to Version 4 were truly system-wide (applying to all domains in all namespaces). System-wide domain parameters at Version 4 and subsequent versions apply to all domains in the current namespace. Therefore, upgrading to Version 4 causes all existing system-wide domain parameters to be applied as namespace-wide domain parameters for all namespaces containing iKnow domains at the time of upgrade. Namespaces without iKnow domains, and subsequently created namespaces will not receive these system-wide domain parameter settings. Note that the SAMPLES namespace does not receive existing system-wide domain parameters, because upgrading your system overwrites SAMPLES, creating a namespace without iKnow domains.

20

iKnow Web Services

The %iKnow package provides web service classes that execute iKnow queries. This chapter provides an overview of these classes and describes how to use them. It discusses the following topics:

- Available iKnow web services
- How to use an iKnow web service
- Example
- Comparison of iKnow web services to the primary APIs
- Additional sources of information

20.1 Available Web Services

iKnow provides the following web service classes:

- %iKnow.Queries.CcWSAPI — provides web methods that you can use to get information about concept-concept (CC) pairs. These web methods are equivalent to the methods in the %iKnow.Queries.CcAPI class.
- %iKnow.Queries.CrcWSAPI — provides web methods that you can use to get information about Concept-Relation-Concept (CRC) triples. These web methods are equivalent to the methods in the %iKnow.Queries.CrcAPI class.
- %iKnow.Queries.EntityWSAPI — provides web methods that you can use to get information about entities. These web methods are equivalent to the methods in the %iKnow.Queries.EntityAPI class.
- %iKnow.Queries.EquivWSAPI — provides web methods that you can use to manage equivalences. These web methods are equivalent to the methods in the %iKnow.Queries.EquivAPI class.
- %iKnow.Queries.MetadataWSAPI — provides web methods that you can use to manage and get information about source metadata. These web methods are equivalent to the methods in the %iKnow.Queries.MetadataAPI class.
- %iKnow.Queries.PathWSAPI — provides web methods that you can use to get information about paths. These web methods are equivalent to the methods in the %iKnow.Queries.PathAPI class.
- %iKnow.Queries.SentenceWSAPI — provides web methods that you can use to get information about sentences. These web methods are equivalent to the methods in the %iKnow.Queries.SentenceAPI class.
- %iKnow.Queries.SourceWSAPI — provides web methods that you can use to get information about sources. These web methods are equivalent to the methods in the %iKnow.Queries.SourceAPI class.

The %iKnow.Queries package also contains classes that the compiler generated when these classes were compiled. For example, when the compiler compiled the class %iKnow.Queries.CcWSAPI, it generated the classes in the %iKnow.Queries.CcWSAPI package. These generated classes are not intended for direct use.

20.2 Using an iKnow Web Service

To use a web service, you create and use a web client that communicates with it. To do so for an iKnow web service, you use the same procedure as with any other web service:

1. Create a web client that can communicate with the web service. Typically, to do so, you generate the web client using a tool provided by the client technology and you provide the WSDL of the web service as input. This process generates a set of client classes.

   For example, in Caché, use the Studio SOAP wizard. For a Caché web service, Caché conveniently publishes the WSDL at the following URL:

   ```
   http://hostname:port/csp/namespace/web_service_class.cls?WSDL
   ```

   Where:

   - *hostname* is the server on which Caché is running.

   - *port* is the port on which the web server is running.

   - *namespace* is the namespace name.

   - *web_service_class* is the full package and class name of the web service with `.cls` at the end.

     For example, for the class %iKnow.Queries.EntityWSAPI, use `%25iKnow.Queries.EntityWSAPI.cls`

     Important: Be sure to replace the leading percent sign of the package with the URL escape sequence `%25` as shown here.

   For example:

   ```
   http://localhost:57772/csp/samples/%25iKnow.Queries.EntityWSAPI.cls?WSDL
   ```

2. Rather than editing the generated client classes, create an additional class or routine that uses them. The details depend on the technology.

   In Caché, to use a web client, you create an instance of the web client class and then invoke its instance methods. See the following example.

20.3 Example

For demonstration purposes, let us generate and use a Caché web client to work with an iKnow web service on the same machine. To do this in the SAMPLES namespace:

1. In Studio, click **Tools > Add-ins > SOAP Wizard**.

2. On the first screen, click **URL**.

3. Type the following URL:

    ```
    http://localhost:57772/csp/samples/%25iKnow.Queries.EntityWSAPI.CLS?WSDL
    ```

    If needed, replace 57772 with the port that your web server uses for this installation of Caché.

4. Click **Next**.

5. For **Proxy Class Package**, type a package name such as `MyClient`

6. Click **Next** twice.

    The wizard generates and compiles the classes and displays a list of these classes.

    This process generates the class `MyClient.iKnow.Queries.EntityWSAPISoap`. This class defines a set of methods, one for each web method defined in the web service on which this client is based. Each of these methods looks like this (with illegal line breaks added here):

    ```
    Method GetByFilter(domainid As %Integer, filter As %String, filtermode As %Integer, enttype As
    %Integer,
    blackListIds As
    %ListOfDataTypes(ELEMENTTYPE="%String",XMLITEMNAME="blackListIdsItem",XMLNAME="blackListIds"))
    As %XML.DataSet [ Final, ProcedureBlock = 1, SoapBindingStyle = document, SoapBodyUse = literal,
    WebMethod ]
    {
     Quit ..WebMethod("GetByFilter").

    Invoke($this,"http://www.intersystems.com/iKnow/Queries/EntityWSAPI/%iKnow.Queries.EntityWSAPI.GetByFilter",

            .domainid,.filter,.filtermode,.enttype,.blackListIds)
    }
    ```

    The signature of this method is the same as for the corresponding method in the web service.

    This process also generates a set of classes in the `MyClient.iKnow.Queries.EntityWSAPISoap`, which are not for direct use.

7. Click **Finish**.

8. To use the generated client, enter the following in the Terminal in the SAMPLES namespace:

    ```
    set client=##class(MyClient.iKnow.Queries.EntityWSAPISoap).%New()
    ```

    This creates an instance of the client class, which can then communicate with the web service.

9. Execute methods of this class. For example:

    ```
    write client.GetCountByDomain(2)
    ```

20.4 Comparison of iKnow Web Services with Primary iKnow APIs

The primary iKnow APIs use arguments that cannot be easily represented in SOAP messages, so the methods in iKnow web services have different signatures than do the methods in the primary iKnow APIs. In particular, note the following differences:

*   Instead of %Library.List, the web services (and their clients) use %Library.ListOfDataTypes. This means that to build a list for a web service, you create an instance of %Library.ListOfDataTypes and then use its **SetAt()** method to add items. You cannot use list functions such as **$LISTBUILD** with this instance.

- Instead of an instance of %iKnow.Filters.Filter, the web services (and their clients) use a string of the form returned by the **ToString()** method of that class.

- For APIs that return complex results, rather than returning results by reference as a multidimensional array, the web services (and their clients) return an instance of %XML.DataSet.

  Important: This structure is supported only in .NET and in Caché. Other web technologies do not recognize this format.

20.5 See Also

For information on web services and clients in Caché, see *Creating Web Services and Web Clients in Caché*.

21

iKnow KPIs and DeepSee Dashboards

DeepSee dashboard technology enables you to create web-based dashboards for your end users. Despite the specific name, these dashboards are not reserved solely for DeepSee users and can display data other than DeepSee data items. Among other items, dashboards can display iKnow KPIs (key performance indicators).

In general, a *KPI* is a query that can be executed and displayed on dashboards; when the dashboard is displayed, the query is executed. An *iKnow KPI* uses an iKnow ObjectScript query.

This section describes how to create iKnow KPIs and display them on dashboards. It discusses the following topics:

- KPI terminology
- Basic steps to define a KPI
- Available KPI filters
- How to override KPI properties
- Example KPI
- How to create a dashboard to display the KPI
- How to provide access to dashboards
- Additional sources of information

For information on using iKnow data within DeepSee cubes, see the *Advanced DeepSee Modeling Guide*. (Or, for information on the older form of cube integration, see the appendix "DeepSee Cube Integration (Deprecated Form).")

21.1 KPI Terminology

In a KPI, each row returned by the query is a separate *series* of the KPI. The following shows some of the series of a KPI (as seen on the KPI test page, discussed later in this chapter). The series names are shown in the first column of the **KPI Values** table.

For iKnow KPIs, the name of a series (by default), is the value in the first column returned by the query.

A KPI also contains *properties*, each of which corresponds to a column in the returned data. In the previous example, the KPI has five properties. When you define an iKnow KPI, you can override these property names and their order.

The KPI query can include parameters. These are called *KPI filters*, because they usually (but not always) filter the values returned by the KPI. The KPI test page shows all the available KPI filters. The iKnow KPI mechanism automatically provides a set of KPI filters for any iKnow KPI.

A KPI can also define *actions*, which execute custom code. When you add a KPI to a dashboard, you can add controls to the dashboard to execute these actions. For any iKnow KPI based on a query that uses paging, that KPI defines the **Previous page** and **Next page** actions.

21.2 Defining a KPI That Uses an iKnow Query

To define a KPI that uses an iKnow query, do the following:

1. Create a class that extends one of the following classes:

    - %iKnow.DeepSee.GenericKPI — Use this for most queries. You specify the query in the next step.

    - %iKnow.DeepSee.SourceListFilterKPI — Use this to display information about the sources. In this case, the system uses either the **GetByEntities()** or the **GetByDomain()** method of %iKnow.Queries.SourceAPI.

2. In this class, specify the iKnow domain to use. To do so, do either of the following:

    - Specify the integer ID of the domain. To do so, override the *IKDOMAINID* class parameter and set it equal to the integer ID of the iKnow domain.

    - Specify the DeepSee cube and iKnow measure that define the domain. To do so, override the following class parameters:

        – *IKCUBENAME* — Should equal the logical name of a DeepSee cube.

–   *IKMEASURENAME* — Should equal the local name of an iKnow measure in the given cube.

This technique is possible only if you are using the DeepSee cube integration; see the *Advanced DeepSee Modeling Guide*.

3. Also override the following class parameters, depending on the superclass you used:

    •   *IKPAGESIZE* — Should equal the number of rows to display on any page. The default is 10.

        Within DeepSee dashboards, this affects the number of rows shown per page in a dashboard widget.

    •   *IKQUERYCLASS* — (Only if you subclass %iKnow.DeepSee.GenericKPI) Should equal the complete package and class name of the iKnow API. Use one of the API classes that belong to the ObjectScript API. For example: %iKnow.Queries.EntityAPI

    •   *IKQUERYNAME* — (Only if you subclass %iKnow.DeepSee.GenericKPI) Should equal the name of a method in that class.

4. Add an XData block like the following to the class; this specifies the logical name and the display name for the KPI:

```
/// This XData definition defines the KPI.
XData KPI [ XMLNamespace = "http://www.intersystems.com/deepsee/kpi" ]
{
<kpi name="MyKPI" displayName="My KPI">
</kpi>
}
```

Specify the `name` and `displayName` values as needed. Note that `name` should not include any spaces or punctuation. The value of `displayName` is localizable. You can omit `displayName`; by default, `name` is used.

Also see the later subsection "Overriding the KPI Properties."

5. Compile the class.

    Optionally use the KPI test page. If you are currently viewing the class in Studio, click **View > Web Page**.

    To test the KPI, use the drop-down lists in the **Filters** section. Then click **Submit Query**.

    These drop-down lists include all the available KPI filters; see the next subsection.

21.3 Available KPI Filters

The query of the KPI can include parameters; these are called *KPI filters*, because they usually (but not always) filter the values returned by the KPI. For an iKnow KPI, the following KPI filters are automatically available:

•   A meaningful subset of iKnow query parameters for the query that you are using (some query parameters, such as the iKnow domain ID, are handled automatically and are not exposed)

•   All public iKnow source metadata fields for this domain

•   The NAMECOLUMN parameter, which lets you specify the column to use as the series name

Important: Many of the iKnow queries include a *filtermode* argument that you use to specify the statistical reprocessing to perform after applying a filter; see "Filter Modes," earlier in this book. When you use such queries directly, the default *filtermode* is $$$FILTERONLY, which performs no reprocessing.

When you expose such queries as an iKnow KPI, the *filtermode* argument is always specified as $$$FILTERALLANDSORT, so that the frequency and spread statistics are recomputed and the results are re-sorted. Therefore you might see different results when using these queries as iKnow KPIs than when using them directly, depending on how you specify *filtermode*.

21.4 Overriding the KPI Properties

By default, any KPI based on %iKnow.DeepSee.GenericKPI exposes all result columns in the same order as the query result, with the same names as in the query result. You can modify the order, change names, and hide columns.

Each result column is a KPI property. Users see the following when they create DeepSee dashboards:

- When a user adds a pivot table widget based on a KPI, the properties are shown in the same order and with the same names as defined in the KPI, by default.

- When a user adds a scorecard widget based on a KPI, the user chooses the properties to display. The drop-down list of choices contains the properties in the same order and with the same names as defined in the KPI.

- In both cases, users can modify the order of the KPI properties and change the titles displayed for them.

To modify the order of the KPI properties, change their names, and hide properties, add a set of `<property>` elements within the XData block, as follows:

Class Member

```
XData KPI [ XMLNamespace = "http://www.intersystems.com/deepsee/kpi" ]
{
<kpi name="MyKPI" displayName="My KPI">
    <property name="entity" displayName="Entity" />
    <property name="frequency" displayName="Frequency" />
    <property name="spread" displayName="Spread" />
</kpi>
}
```

For `<property>`, the value for `name` must exactly match the name of the field returned by the query. For `displayName`, use the name that should be visible to users. List the `<property>` elements in the desired order, and list all the properties that should be visible.

21.5 Example

The following shows an example KPI:

Class Definition

```
Class GIKNOW.TopEntitiesKPI Extends %iKnow.DeepSee.GenericKPI
{

Parameter IKDOMAINID = 1;

Parameter IKPAGESIZE As %Integer = 10;

Parameter IKQUERYCLASS = "%iKnow.Queries.EntityAPI";
```

```
Parameter IKQUERYNAME = "GetTop";

/// This XData definition defines the KPI.
XData KPI [ XMLNamespace = "http://www.intersystems.com/deepsee/kpi" ]
{
<kpi name="TopEntities" displayName="Top Entities in iKnow domain 1"  >
<property name="resultNumber" displayName="rank"/>
<property name="entity" displayName="entity"/>
<property name="entUniId" displayName="Id"/>
<property name="frequency" displayName="frequency"/>
<property name="spread" displayName="spread"/>
</kpi>
}

}
```

For another example, see "Example Dashboard with iKnow KPI," in the next section.

21.6 Creating a Dashboard to Display the KPI

To make a KPI available to users, you add it to a dashboard. Users can then access the dashboard in various ways; see "Providing Access to Dashboards," later in this chapter.

- How to create a dashboard

- Changing the series name

- Configuring the properties

- Adding previous page and next page buttons

- Example dashboard

21.6.1 Creating Dashboards: Basics

The following instructions briefly explain how to create a simple dashboard that displays a KPI:

1. Click **Home,DeepSee,User Portal** and then click **View**.

   The system then displays the User Portal, which lists any existing public dashboards and pivot tables in this namespace.

2. Click **Menu > New Dashboard**.

   The system displays a dialog box that prompts you for basic information about the new dashboard.

3. Type a value for **Dashboard Name**.

4. Optionally specify a value for **Folder**.

5. For **Page Layout**, click the third option (no workboxes).

6. Click **OK**.

   The system creates, saves, and displays the dashboard, which is initially empty.
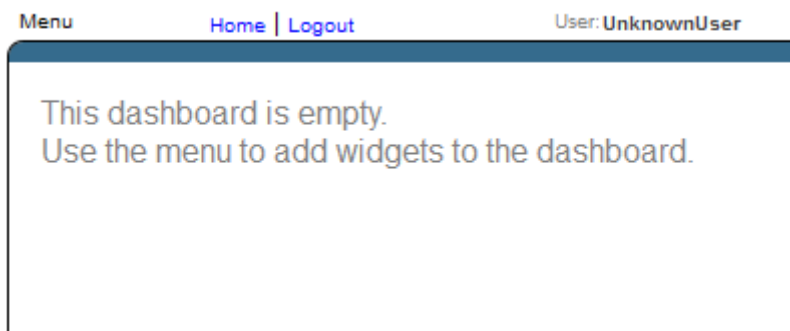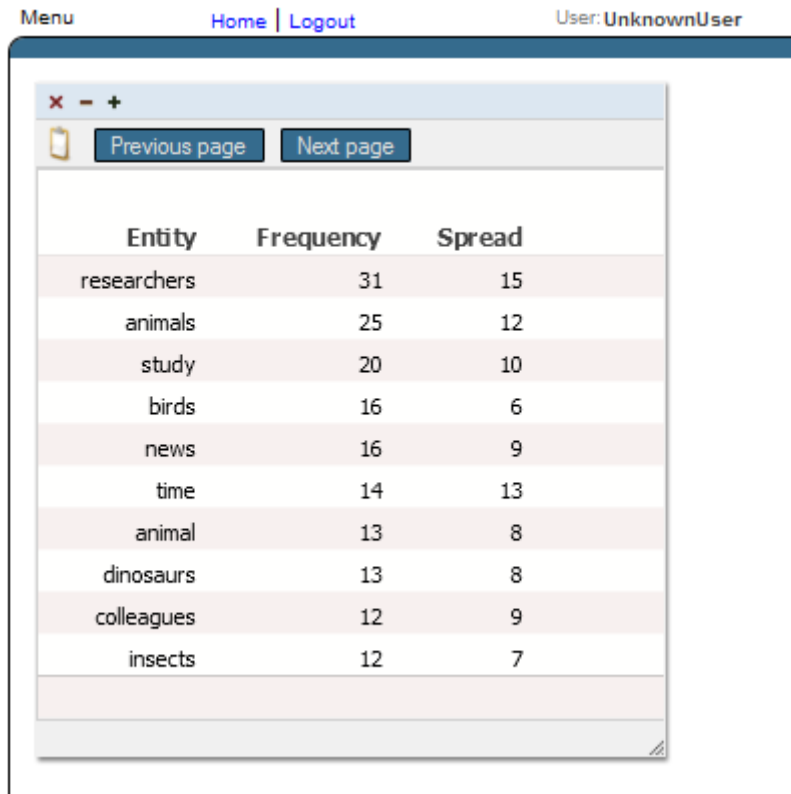
7. Add a widget to display a KPI. To do so:

   a. Click **Menu > Add New Widget...**.

   b. In the left area, click an item to show a list of choices.

       Select either a pivot table widget or a scorecard.

   c. Click the type of the widget to use.

   d.
       On the **Data Source** tab, click the Search button next to **Data source**.

   e. Click the name of the KPI.

   f. Click **OK**.

   g. If you chose a scorecard, see "Configuring the Properties," later in this section.

   h. Click **OK** to add this widget.

8. Resize the widget and click **Menu > Save** again.

9. For configuration options, see the next subsections.

The result might be as follows:
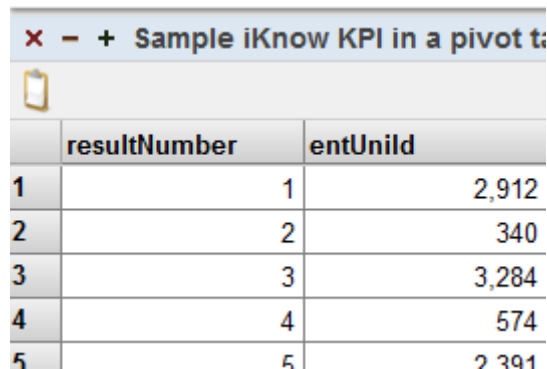
The last section of this chapter provides links to additional information.

21.6.2 Changing the Series Names

When you display an iKnow KPI in a pivot table widget, the series names are used as the names of the rows. By default, the series names are taken from the first column of the returned values, which may not be useful in this scenario. For example, for the following KPI, the first column is the result number:
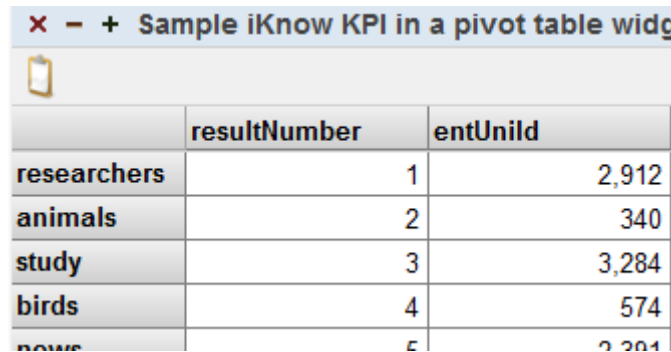
To choose a different column to use as the series names, do the following:

1. Click the Reconfigure button ▯ on the widget.

2. Click the **Controls** tab.

3. Click the Add button ⊕ to the right of the table, which is initially empty.

4. For **Type**, select **Hidden**.

5. For **Action**, select **Filter**.

6. For **Filter**, select **Series name column**.

7. For **Default Value**, click the Add button ⊕.

8. Select the name of the column to use (for example, **entity** for the KPI shown here).

9. Click **OK** to add the default value.

10. Click **OK** to add the control.

11. Click **OK** to complete the widget reconfiguration.

Now the row names have changed. For example:

21.6.3 Configuring the Properties

When you display a KPI in a widget, you may need to reconfigure the properties shown in that widget:

*   For a scorecard widget, you must configure the properties. No columns are shown by default.

*   For a pivot table widget, you might need to configure the properties. All columns are shown by default.

To configure the properties:

1. Click the Reconfigure button ⬚ on the widget.

2. Click the **Data Properties** tab.

    On this tab, you specify the properties (or columns) of the KPI.

3. To configure a property, click the Add button ⊕.

4. For **Data Value**, select a value from the drop-down list; this lists the result columns of the KPI.

5. Optionally type a caption into **Label**.

6. Optionally type a format string into **Format**. By default, numeric values are shown with the thousands separator used in your locale. To format the numbers without the thousands separator, type # into **Format**.

7. Click **OK**.

    After you have added each result column of the KPI, the dialog box should looks something like this, depending on your KPI:
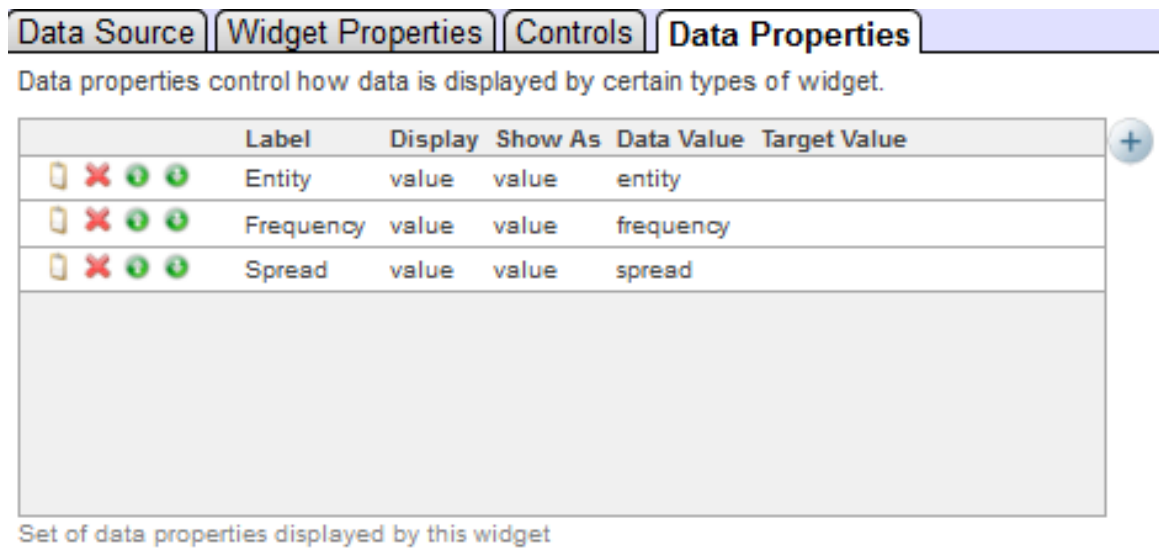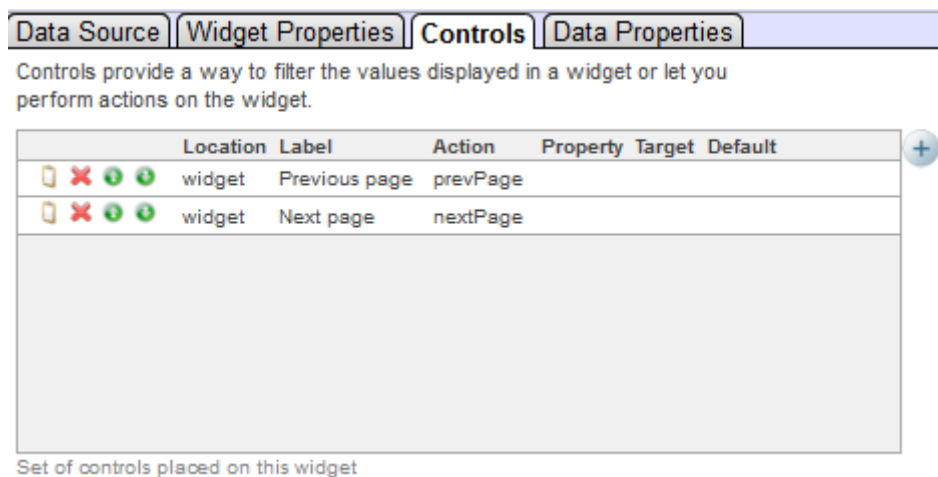
8. Click **OK** to complete the reconfiguration.

21.6.4 Adding Previous Page and Next Page Buttons

For any iKnow KPI based on a query that uses paging, that KPI defines the **Previous page** and **Next page** actions. You can configure the widget to include buttons that execute these actions. To do so:

1. Click the Reconfigure button ⬜ on the widget.

2. Click the **Controls** tab.

3. Add a **Previous Page** buttons to this widget, as follows

   a. Click the Add button ⊕ to the right of the table, which is initially empty.

      The system displays a dialog box where you specify the control.

   b. For **Action**, select **Previous page**.

   c. For **Control Label or Icon**, type `Previous Page`

   d. Click **OK** to add the control.

Use similar steps to add the **Next Page** button.

Now the dialog box looks like this:

4. Click **OK** to complete the widget reconfiguration.

    Now the widget includes buttons like this:
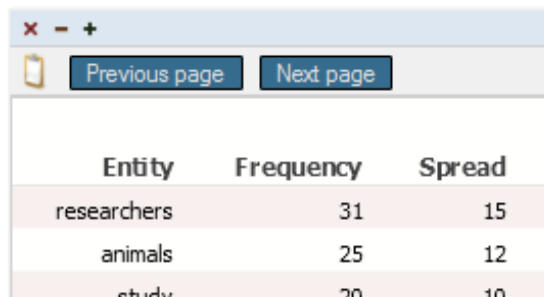
21.6.5 Example Dashboard with iKnow KPI

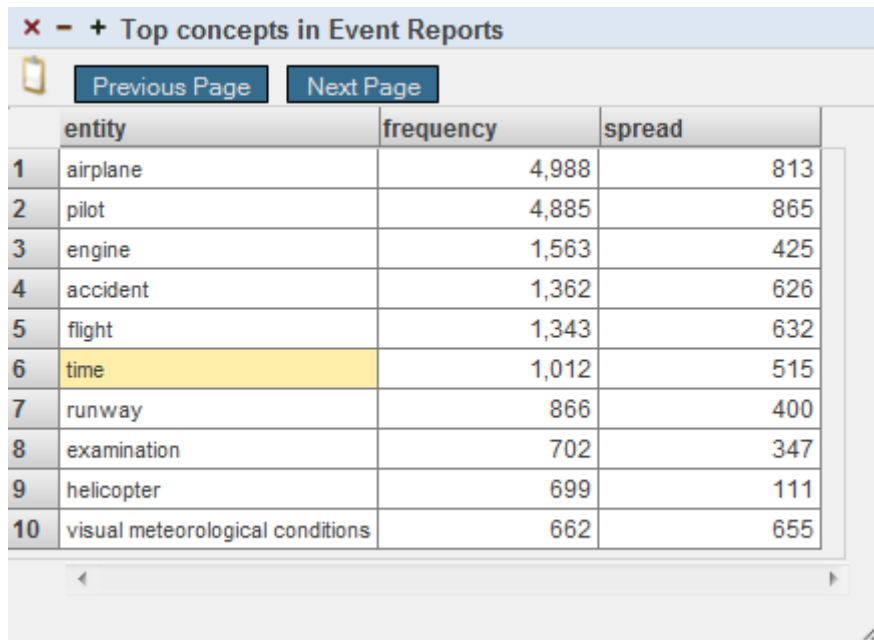The SAMPLES namespace provides an example dashboard that displays an iKnow KPI. To see this dashboard:

1. In the Management Portal, access the SAMPLES namespace.

2. Click **Home,DeepSee,User Portal** and then click **View**.

    The system then displays the User Portal, which lists any existing public dashboards and pivot tables in this namespace.

3. Click **Aviation event reports**.

    The system displays the requested dashboard.

In this dashboard, the widget **Top concepts in Event Reports** displays an iKnow KPI. This widget appears as follows:

This widget displays the KPI defined in the class Aviation.KPI.TopConcepts.

21.7 Providing Access to Dashboards

You can provide dashboards to your users in various ways:

- You can embed a dashboard in the iFrame of a Zen page. To do so, include something like this within the `<iframe>` definition:

```
src="_DeepSee.UserPortal.DashboardViewer.zen?DASHBOARD=Tutorial/My First Dashboard.dashboard"
```

  In this example, `Tutorial` is the name of the folder that contains the dashboard, and `My First Dashboard` is the name of the dashboard.

- Your application can provide direct links to dashboards.

- You can provide a user portal that has the ability to display dashboards. You can use the DeepSee User Portal or create your own.

  The DeepSee User Portal is intended for direct use by end users (in contrast to such back end tools as Studio and the Management Portal). This portal is intended for general use (and has a general appearance), despite its specific name. It is not labeled with "DeepSee."

21.8 See Also

- For information on Zen pages, see *Using Zen*.

- For information on creating DeepSee dashboards, see *Creating DeepSee Dashboards*.

- For information on accessing dashboards from your application, see "Accessing Dashboards from Your Application" in the *DeepSee Implementation Guide*.

- For information on packaging dashboard definitions into classes, see "Packaging DeepSee Elements into Classes" in the *DeepSee Implementation Guide*.

- For information on the DeepSee User Portal and dashboards, see the *DeepSee End User Guide*.

22

Customizing iKnow

You can customize how the iKnow semantic analysis engine loads and lists source texts. The following types of customization are supported:

- Lister. By default, iKnow provides several listers that correspond to common sources of data. You can create a custom lister appropriate for your data.

- Processor. By default, iKnow uses the processor that corresponds to the lister. You can create a custom processor and associate it with your custom lister, or you can associate an existing processor to your custom lister.

- Converter. A Converter is an object that can transform complex input text into the plain text expected by the iKnow engine. For example, a Converter can extract plain text from a PDF or RTF document, or select specific nodes from an XML document. By default, iKnow does not use a converter. You can create a custom converter and specify it in the Lister **SetConverter()** or **Init()** method.

A Lister can optionally specify its Configuration, Processor, and Converter. You can specify these using the **SetConfig()**, **SetProcessor()**, and **SetConverter()** methods, or specify all three using the **Init()** method.

The following example shows how to specify a custom processor and/or a custom converter using the optional Lister **Init()** method. You can specify an empty string ("") for any of the **Init()** method parameter to take the default for the specified Lister.

```
SET flister=##class(%iKnow.Source.File.Lister).%New(domId)
    DO flister.Init(configuration,myprocessor,processorparams,myconverter,converterparams)
```

22.1 Custom Lister

iKnow provides a base lister class and five subclasses containing listers specific to different types of input sources.

In order to implement a custom lister you begin with the base Lister class, %iKnow.Source.Lister, and override several of its defaults.

22.1.1 Lister name

In order to be able to work with the lister, the lister needs to specify the format in which the external id for each source is presented. The external id for a lister consists of the lister name and the full reference. The full reference consists of the groupName and the localRef. An external id is shown in the following example:

```
:MYLISTER:groupname:localref
```

In this example, MYLISTER is the lister name alias. If you don't provide an alias, the full classname of the lister class is used. To determine the alias for your lister, use the **GetAlias()** method.

A lister name alias must be unique within the current namespace. If you specify a lister name alias that already exists, iKnow generates a $$$IKListerAliasInUse error.

22.1.2 SplitFullRef() and BuildFullRef()

You must specify a **SplitFullRef()** instance method for your custom lister. This method is used to extract the groupName and the localRef from the fullRef string. Its results are supplied to the **SplitExtId** class method. Assume your lister has an external id format like this: `:MYLISTER:groupname:localref`.

In this simple example, fullRef consists of the string groupname:localref, so the groupName is `$PIECE(fullRef,":",1)` and localRef is `$PIECE(fullRef,":",2)`. Note that this is a very simple example; it does not work if the groupName or localRef parts contain ":" characters.

You must specify a **BuildFullRef()** instance method for your custom lister. This method is used to combine the groupName and the localRef to form the fullRef string. Its results are supplied to the **BuildExtId** class method.

22.1.3 Default Processor

A Processor is a class that takes as input the list populated by the Lister, reads the corresponding source text, and directs the source data to the iKnow engine for indexing. It can, optionally, pass this source data through a Converter.

iKnow Processors are subclasses of the %iKnow.Source.Processor class. Each processor subclass is designed to read sources of a specific type, such as the %iKnow.Source.File.Processor which reads files from a directory.

Every Lister has a default processor that is capable of processing the sources from that lister. By default, it uses a class called Processor in the same package as the Lister. If there is no processor corresponding to the specified lister, or if you wish to use the generic %iKnow.Source.Temp.Processor, you should override the **DefaultProcessor()** method and specify the desired default processor.

22.1.4 Expand List

The **ExpandList()** method is responsible for listing all sources that need to be indexed. This method should be overridden by user-defined subclasses that implement how to scan through the particular type of source location or structures for your custom Lister. The parameters for this method are the same as those used when invoking the corresponding **AddListToBatch()** method. The parameters may differ, depending on the Lister that you implement. Make sure that the Lister-specific **ExpandList()** parameters are documented, so that a user knows which parameters to supply to the **lister.AddListToBatch()** method or the **loader.ProcessList()** method.

The **ExpandList()** parameters are as follows (in order):

- Path: the location where the sources are located, specified as a string.

- Extensions: one or more file extension suffixes that identify which sources are to be listed. Specified as a %List of strings.

- Recursive: a boolean value that specifies whether to search subdirectories of the path for sources.

- Filter: a string specifying a filter used to limit which sources are to be listed.

For further details, refer to Lister Parameters in the chapter "Loading Text Data into iKnow".

22.2 Custom Processor

A processor can either copy the complete source into a temporary global for iKnow processing, or it can store a reference to the source in a temporary global. These temporary globals are used by the iKnow engine to index the text and store the results in iKnow globals.

If a Lister does not have a corresponding processor, the %iKnow.Source.Temp.Processor is the default processor. It copies the complete text of each source into a temporary global. The other supplied processors store a reference to the source in a temporary global. You can use ..StoreTemp to specify copying the source, or ..StoreRef to specify storing a reference to the source.

22.2.1 Metadata

While listing sources, the Lister is capable of extracting metadata that should be added to the sources. In order to let the system know which metadata the Lister will provide, you can call the function ..RegisterMetadataKeys(metaFieldNames). The metaFieldNames parameter is a %List containing the keys for the metadata key-value pairs. After that you can provide the metadata values by using the function ..SetMetadataValues(ref, metaValues). the metaValues parameter is a %List containing the values for the metadata key-value pairs. They should appear in the same order as the keys are listed.

After establishing the metadata in the Lister, you can access this metadata in your processor by implement the **GetMetadataKeys()** method. This method should return a %List of keys from the metadata key-value pairs. In the **FetchSource()** method the processor can then set the appropriate values for calling ..SetCurrentMetadataValues(values), where *values* is a %List of the values of the metadata key-value pairs, in the same order as the keys were reported.

22.3 Custom Converter

A Converter converts source text to plain text by removing tags from the source text. Tags are non-content elements used to format the text for display or printing. For example, you might use a converter to remove tags from RTF (Microsoft Rich Text Format) files, or to extract plain text from a PDF file. A converter is invoked by the Lister and applied prior to indexing the source text. Depending on the format of your source documents, the use of a source converter is an optional step.

iKnow provides one sample converter, the subclass %iKnow.Source.Converter.Html, which you can use to remove HTML tags from source text. This is a basic HTML converter; you may need to customize your instance of this converter to support full conversion of your HTML source texts.

In order to implement a custom Converter you need to override several methods from the base converter class %iKnow.Source.Converter.

22.3.1 %OnNew

The user-provided **%OnNew()** callback method is invoked by the **%New()** method. It takes as its parameter a %List of any parameters that the Converter requires.

22.3.2 Buffer String

The **BufferString()** method will be called as many times as needed to buffer the complete document into the Converter. Each call will provide a chunk of text by means of the *data* parameter (max 32K). When no more data is to be buffered, the *Convert()* method will be called.

22.3.3 Convert

The **Convert()** method is responsible for processing the buffered content and converting the data into plain text (for example, RTF file conversion), or extracting the required data from the buffer (for example, node extraction from xml). The converted or extracted data will need to be buffered, as the converted data can be larger than 32K.

22.3.4 Next Converted Part

The **NextConvertedPart()** method is called after the *Convert()* method. This method must return the converted data in chunks of 32K. Every time this method is called, you need to return the next chunk. If no more data is available, this method should return the empty string ("") to indicate that it has finished extracting the converted data.

23

Language Identification

This chapter describes how to configure and use Automatic Language Identification (ALI), which is applied at the sentence level. It also describes a few language-specific issues.

23.1 Configuring Automatic Language Identification

An iKnow Configuration establishes the language environment for source document content. A Configuration is independent of any specified set of source data. You can either define a Configuration, or take the default Configuration. If you do not specify a Configuration, the default is English-only, with no automatic language identification.

A configuration defines the following language options:

- What language(s) the source documents contain, and therefore which languages to test for and which language models to apply. The available options are Dutch (nl), English (en), French (fr), German (de), Japanese (ja), Portuguese (pt), Russian (ru), Spanish (es), Swedish (sv), and Ukrainian (uk). Specify a language using the ISO two-letter code. You can specify multiple languages as a Caché list structure.

- When specifying more than one language, specify a boolean value to activate automatic language identification.

The following example creates a configuration that assumes all source texts will be in English or French, and supports automatic language identification:

ObjectScript

```
SET myconfig="EnglishFrench"
IF ##class(%iKnow.Configuration).Exists(myconfig) {
    SET cfg=##class(%iKnow.Configuration).Open(myconfig)
    WRITE "Opened existing configuration ",myconfig,!
}
ELSE {
    SET cfg=##class(%iKnow.Configuration).%New(myconfig,1,$LISTBUILD("en","fr"),"",1)
    DO cfg.%Save()
    IF ##class(%iKnow.Configuration).Exists(myconfig)
    {WRITE "Configuration ",myconfig," now exists",! }
    ELSE {WRITE "Configuration creation error" QUIT }
}
    SET cfgId=cfg.Id
    WRITE "with configuration ID ",cfgId,!
 SET rnd=$RANDOM(2)
IF rnd {
    SET stat=##class(%iKnow.Configuration).%DeleteId(cfgId)
    IF stat {WRITE "Deleted the ",myconfig," configuration" }
    }
ELSE {WRITE "No delete this time",! }
```

23.2 Using Automatic Language Identification

iKnow performs automatic language identification on a per-sentence basis. When the current configuration has activated automatic language identification, iKnow tests each sentence in each source text to determine which of the languages specified in the Configuration is the language used in that sentence. This identification is a statistical probability. This has the following consequences:

- If a sentence contains text in more than one language specified in the Configuration, iKnow will assign the sentence to what it determines is the predominant language of the sentence.

- If a sentence is in a language *not* specified in the Configuration (or a language not supported by iKnow), iKnow will assign the sentence to one of the specified Configuration languages.

iKnow subsequently uses this language determination in determining CRCs and other iKnow analysis.

Thus, source texts and sentences within a source text can be in different languages. iKnow automatically determines which language model to apply. Automatic language identification also assigns a confidence level in its language identification as an integer indicating a percentage. These range from 100 (complete confidence) to 0 (indeterminate). If automatic language identification is not active, all sentences are assigned a confidence level of 0.

23.2.1 Language Identification Queries

The following example uses **GetTopLanguage()** to identify the language for a source and the degree of confidence in that identification. Because language identification is performed on the sentence level, the language for the source is the result of averaging the language identification confidence for the component sentences. This method returns the language as a two character abbreviation (in this case, "en"). Note that *totlangconf* (the total of the language confidence for the sentences) must be divided by *numlangsent*, not by *numsent*. These two sentence count numbers are usually, but not always, the same. This is because a source may contain sentences for which no language can be determined.

ObjectScript

```
Configuration
  SET myconfig="EnFr"
  IF ##class(%iKnow.Configuration).Exists(myconfig)
      {SET cfg=##class(%iKnow.Configuration).Open(myconfig) }
  ELSE {SET cfg=##class(%iKnow.Configuration).%New(myconfig,1,$LISTBUILD("en","fr"),"",1)
      DO cfg.%Save() }
  SET cfgId=cfg.Id
  ZNSPACE "Samples"
DomainCreateOrOpen
  SET dname="mydomain"
  IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
  ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
  SET stat=domoref.DropData()
  IF stat { WRITE "Deleted the data from the ",dname," domain",!!
          GOTO ListerAndLoader }
  ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
          QUIT}
ListerAndLoader
  SET domId=domoref.Id
  SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
  SET stat=flister.SetConfig(myconfig)
    IF stat '= 1 { WRITE "SetConfig error ",$System.Status.DisplayError(stat)
                QUIT }
  SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
    SET myquery="SELECT Top 10 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
```

```
     SET idfld="UniqueVal"
     SET grpfld="Type"
     SET dataflds=$LB("NarrativeFull")
UseLister
   SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
   SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
GetSources
   DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId)
   SET i=1
   WHILE $DATA(result(i)) {
      SET intId = $LISTGET(result(i),1)
      SET extId = $LISTGET(result(i),2)
     SET numsent = ##class(%iKnow.Queries.SentenceAPI).GetCountBySource(domId,result(i))
      WRITE !,extId," has ",numsent," sentences",!
     SET srclang = ##class(%iKnow.Queries.SourceAPI).GetTopLanguage(domId,intId,.totlangconf,.numlangsent)

     WRITE "Source language is ",srclang,!,"with a confidence % of ",totlangconf/numlangsent,!!
     SET i=i+1
     }
```

The following example uses **GetLanguage()** to identify the language for each sentence in a source and the degree of confidence in that identification. This method returns the language as a two character abbreviation (in this case, "en") and the confidence level as a percentage between 0 and 100. Note that the confidence level is rarely (if ever) 100%.

ObjectScript

```
Configuration
   SET myconfig="EnFr"
   IF ##class(%iKnow.Configuration).Exists(myconfig)
        {SET cfg=##class(%iKnow.Configuration).Open(myconfig) }
   ELSE {SET cfg=##class(%iKnow.Configuration).%New(myconfig,1,$LISTBUILD("en","fr"),"",1)
         DO cfg.%Save() }
   SET cfgId=cfg.Id
   ZNSPACE "Samples"
DomainCreateOrOpen
   SET dname="mydomain"
   IF (##class(%iKnow.Domain).NameIndexExists(dname))
      { WRITE "The ",dname," domain already exists",!
        SET domoref=##class(%iKnow.Domain).NameIndexOpen(dname)
        GOTO DeleteOldData }
   ELSE
      { WRITE "The ",dname," domain does not exist",!
        SET domoref=##class(%iKnow.Domain).%New(dname)
        DO domoref.%Save()
        WRITE "Created the ",dname," domain with domain ID ",domoref.Id,!
        GOTO ListerAndLoader }
DeleteOldData
   SET stat=domoref.DropData()
   IF stat { WRITE "Deleted the data from the ",dname," domain",!!
             GOTO ListerAndLoader }
   ELSE    { WRITE "DropData error ",$System.Status.DisplayError(stat)
             QUIT}
ListerAndLoader
   SET domId=domoref.Id
   SET flister=##class(%iKnow.Source.SQL.Lister).%New(domId)
   SET stat=flister.SetConfig(myconfig)
     IF stat '= 1 { WRITE "SetConfig error ",$System.Status.DisplayError(stat)
                    QUIT }
   SET myloader=##class(%iKnow.Source.Loader).%New(domId)
QueryBuild
   SET myquery="SELECT Top 10 ID AS UniqueVal,Type,NarrativeFull FROM Aviation.Event"
   SET idfld="UniqueVal"
   SET grpfld="Type"
   SET dataflds=$LB("NarrativeFull")
UseLister
   SET stat=flister.AddListToBatch(myquery,idfld,grpfld,dataflds)
      IF stat '= 1 {WRITE "The lister failed: ",$System.Status.DisplayError(stat) QUIT }
UseLoader
   SET stat=myloader.ProcessBatch()
      IF stat '= 1 {WRITE "The loader failed: ",$System.Status.DisplayError(stat) QUIT }
GetOneSource
   DO ##class(%iKnow.Queries.SourceAPI).GetByDomain(.result,domId)
   FOR i=1:1:10 {
    IF $DATA(result(i)) {
      SET intId = $LISTGET(result(i),1)
      SET extId = $LISTGET(result(i),2)
      SET myconf=0
     SET numSentS = ##class(%iKnow.Queries.SentenceAPI).GetCountBySource(domId,result(i))
```

```
   WRITE !,extId," has ",numSentS," sentences",!
GetSentencesInSource
    SET sentStat=##class(%iKnow.Queries.SentenceAPI).GetBySource(.sent,domId,intId)
    IF sentStat=1 {
        SET i=1
        WHILE $DATA(sent(i)) {
            SET sentnum=$LISTGET(sent(i),1)
            WRITE "sentence:",sentnum
            SET lang = ##class(%iKnow.Queries.SentenceAPI).GetLanguage(domId,sentnum,.myconf)
            WRITE " language:",lang," confidence:",myconf,!
            SET i=i+1
        }
    }
   }
   ELSE { WRITE !,"That's all folks!" }
  }
```

23.3 Overriding Automatic Language Identification

You can use the LanguageFieldName domain parameter to override Automatic Language Identification. If activated, this parameter determines which language to apply by accessing a metadata field for each source. This metadata field contains the ISO language code. If the metadata field data is present, Automatic Language Identification is overridden for that source. If the metadata field is empty or invalid, Automatic Language Identification is used for that source. The LanguageFieldName domain parameter is inactive by default. For further details, refer to the Domain Parameters appendix of this manual.

23.4 Language-Specific Issues

German: the German *eszett* ("ß") character is normalized as "ss". German commonly requires setting the EnableNgrams domain parameter.

24

iFind Search Tool

This chapter describes the iFind search facility, a tool for performing context-aware text search operations. To use iFind you must define an iFind index for each column containing text that you wish to search. You can then search the text records using a standard SQL query with a WHERE clause containing iFind SQL search syntax. The query will return all records that contain the specified search item or items. You can also highlight the matching text in the returned records.

Note: To use iFind you must have a Caché license that provides access to iKnow. (You can view the licence key from the Management Portal: select **System Administration** then **Licensing**.)

24.1 Indexing Sources for iFind Search

You can use iFind to search SQL text in %String data type or %Stream.GlobalCharacter (character stream) data type.

To perform an iFind search, the column to be searched must have a defined iFind bitmap index. There are three levels of iFind indexes. These levels are defined in nested subclasses. Each index level provides all of the features of the previous level, plus additional iFind features specific to that level. You can create any of the following iFind index types:

- Basic index (%iFind.Index.Basic): supports iFind word and word phrase search with wildcards.

- Semantic index (%iFind.Index.Semantic): supports iFind search of iKnow entities, and optionally supports the negation attribute.

- Analytic index (%iFind.Index.Analytic): supports the all of the iKnow features of Semantic index, as well as path, proximity, and dominance information.

Each index level supports all of the parameters of the previous level, and adds one or more additional parameters. Unspecified parameters take default values.

The following Class Definition example creates a table with a Semantic index on the Narrative property (column). The indexed property can be of data type %String or %Stream.GlobalCharacter:

Class Definition

```
  Class Aviation.TestSQLSrch Extends %Persistent [
      DdlAllowed,Owner={UnknownUser},SqlRowIdPrivate,
      SqlTableName=TestSQLSrch ]
{
Property UniqueNum As %Integer;
Property CrashDate As %TimeStamp [ SqlColumnNumber=2 ];
Property Narrative As %String(MAXLEN=100000) [ SqlColumnNumber=3 ];
Index NarrSemanticIdx On (Narrative) As %iFind.Index.Semantic(INDEXOPTION=0,
    LANGUAGE="en",LOWER=1);
Index UniqueNumIdx On UniqueNum [ Type=index,Unique ];
}
```

An iFind index of any type includes support for the following parameters:

- INDEXOPTION specifies whether or not to index to allow for stemming or decompounding. Because indexing to support these operations significantly increases the size of the index, it is recommended that you specify INDEXOP-TION=0 unless stemming or decompounding will very likely be used. The default is 0.

- LANGUAGE specifies the language to use when indexing records. For example, `"en"` specifies English. Use `"*"` to enable automatic language identification. The default is `"en"`.

- LOWER specifies whether query search is case-sensitive. By default, iFind indexing is not case-sensitive; iFind normalizes the text to lowercase before indexing it. The LOWER parameter determines whether or not to perform this lowercase normalization (LOWER=1, the default, normalizes to lowercase). Because language conventions commonly capitalize words at the beginning of a sentence or when used in a title, normalizing to lowercase is preferable in most applications. If you specify LOWER=0, the query *search_items* string is case-sensitive. For example, if you specify LOWER=0, the query *search_items* string 'turkey' will only match turkey and not Turkey. If you specify LOWER=1, the query *search_items* string 'turkey' will match both turkey and Turkey.

- USERDICTIONARY allows you to specify the name of a user-defined UserDictionary that is applied to the texts before indexing. This parameter is optional and is for advanced use only.

For a full list of supported parameters refer to %iFind.Index.Basic in the *InterSystems Class Reference*.

A Semantic index (%iFind.Index.Semantic) also supports the following optional parameter:

- IFINDATTRIBUTES allows you to specify whether or not to identify negation in the text and store the negation attribute. IFINDATTRIBUTES=1 identifies and indexes negation. The default is IFINDATTRIBUTES=0.

24.1.1 Indexing a JSON Object

You can create an iFind index for text stored in a JSON object. This index specifies the starting position in the JSON structure. iFind recursively indexes all text at that level and all nested levels below it. Specify $ to index the entire JSON object. Specify $.key2 to index the JSON values at key2 and below.

24.1.2 Populating a Table

Like any SQL index, a defined iFind index (by default) is built when you populate a new table, and maintained when you subsequently insert, update, or delete data. You can defer building of an index when populating a table using %NOINDEX, and then use the **%Build()** method to build the index. You can add an index to a table that is already populated with data and then build that index. Refer to Defining and Building Indices for further details.

The following example populates the Aviation.TestIFind table from the Aviation.Events table in the Samples namespace. This example inserts a large amount of text, so running it may take a minute or so:

```
INSERT OR UPDATE INTO Aviation.TestIFind (UniqueNum,CrashDate,Narrative)
    SELECT %ID,EventDate,NarrativeFull FROM Aviation.Event
```

This example uses **INSERT OR UPDATE** with a field defined with a unique key to prevent repeated execution from creating duplicate records.

24.2 Performing iFind Search

You use iFind syntax in an SQL query WHERE clause to perform a text search for one or more text items. These text items may be words or sequences of words (Basic index) or iKnow semantic entities (Semantic index). Multiple text items are an implicit AND search; all of the specified items must appear in the text, in any order. The syntax for iFind search is as follows:

```
WHERE %ID %FIND search_index(indexname,'search_items',search_option,'language')
```

- *indexname* is the name of a defined iFind index for a specific column.

- *search_items* is the list of text items (either words or iKnow entities) to search for, enclosed with quotes. Text items are separated by spaces. An item consists of an alphanumeric string and optional wildcard syntax characters. Text items are not case-sensitive. Available search_items syntax is described below.

- *search_option* is the index option integer that specifies the type of search to perform. Available values include 0 (syntax search), 1 (syntax search with stemming), 2 (syntax search with decompounding and stemming), 3 (syntax search with fuzzy search), and 4 (syntax search with regular expressions). If *search_option*=4, *search_items* is assumed to contain a single Regular Expression string. For further details, refer to the Regular Expressions chapter in *Using ObjectScript*. Note that for performance reasons, iFind does not support some of the more esoteric regular expression syntactic forms; these syntactic forms are supported by the $LOCATE ObjectScript function.

- *language* is the iKnow-supported language model to apply, specified as a two-character string. For example, `'en'` specifies English. If you specify `'*'`, the query performs automatic language identification.

When performing an Basic index search, iFind identifies words by the presence of one or more space characters. Sentence punctuation (a period, comma, semicolon, or colon followed by a space) is ignored. iFind treats all other punctuation as literals. For example, iFind treats "touch-and-go" as a single word. Punctuation such as hyphens or a decimal point in a number are treated as literals. Quote characters and apostrophes must be specified. You specify a single quote character by doubling it.

You can perform any Basic index search (word, co-occurrence, or positional phrase) with a Semantic index. Attempting to perform a Semantic index search with a Basic index results in an SQLCODE -149 error.

24.2.1 SQL search_items Syntax

Basic index *search_items* can contain the following syntax:

Word Search:

word1 word2 word3	Specifies that these exact words must appear (in any order) somewhere in the text. (Logical AND). You can specify a single word, or any number of words separated by spaces.
word1 OR word2 NOT word3 word1 OR (word2 AND word3)	*search_items* can contain AND, OR, and NOT logical operators. AND is the same as separating words with spaces (implicit AND). NOT is logically equivalent to AND NOT. *search_items* can also use parentheses to group logical operators. Explicit AND is needed when specifying multiple words in grouping parentheses: (word2 AND word3). If the explicit AND was omitted, (word2 word3) would be interpreted as a positional phrase. You can use the \ escape character to specify AND, OR, NOT as literals rather than logical operators: \and
word word *word* w*d	An asterisk wildcard specifies 0 or more non-space characters of any type. An asterisk can be used as a prefix, suffix, or within a word. You can use \ escape character to specify * as a literal: *

Co-occurrence Word Search:

[word1,word2,...,range]	Co-occurrence search. Specifies that these exact words must appear (in any order) within the proximity window specified by *range*. You can specify any number of words or multi-word phrases. A multi-word phrase is specified as words separated by spaces with no delimiting punctuation. Words (or positional phrases) are separated by commas, the last comma-separated element is an optional numeric *range*. Words can specify asterisk wildcards.
	A *range* can be specified as min–max or simply as max with a default min of 1. For example, 1–5 or 5. *range* is optional; if omitted, it defaults to 1–20. A range count is inclusive of all of the specified words.
	Co-occurrence search cannot be used with *search_option*=4 (Regular Expressions).

Positional Phrase Search:

Note: You can use double quotes `"word1 word2 word3"` or parentheses `(word1 word2 word3)` to delimit a positional phrase. Because parentheses are also used to group logical operators, the use of double quotes is preferred.

"word1 word2 word3"	These exact words must appear sequentially in the specified order. Words are separated by spaces. Note that no semantic analysis is performed; for example, the words in a "phrase" may be the final word of a sentence and the beginning words of the next sentence. Asterisk wildcards can be applied to individual words in a phrase. A literal parentheses character in the *search_items* must be enclosed with quotes.
"word1 ? word3" "word1 ? ? ? word5"	A question mark indicates that exactly one word is found between the specified words in a phrase. You can specify multiple single question marks, each separated by spaces.
"word1 ?? word6"	A double question mark (with no space between) indicates that from 0 to 6 words are found between the specified words in a phrase.
"word1 [1–3] word5"	Square brackets indicate an interval number of words between the specified words in a phrase: *min-max*. This interval is specified as a variable range, in this case from 1 to 3 missing words.

Semantic index *search_items* can contain the following iKnow entity search syntax in addition to the Basic index syntax:

Full Entity and Partial Entity Search:

{entity}	Specifies the exact wording of an iKnow entity. Asterisk wildcards can be applied to individual words in an entity.
<{entity}	A less-than sign prefix specifies an iKnow entity ending with the specified word(s). There must be one or more words in the entity appearing before the specified word(s).
{entity}>	A greater-than sign suffix specifies an iKnow entity beginning with the specified word(s). There must be one or more words in the entity appearing after the specified word(s).

You can perform any Basic index search (word, co-occurrence, or positional phrase) with a Semantic index. Attempting to perform a Semantic index search with a Basic index results in an SQLCODE -149 error.

Multiple *search_items* can be specified, separated by spaces. This is an implicit AND test. For example:

SQL

```
SELECT Narrative FROM Aviation.TestIFind WHERE %ID %FIND
search_index(NarrSemanticIdx,'<{plug electrode} (flight plan) instruct*',0,'en')
```

means that a Narrative text must include one or more iKnow entities that end with "plug electrode", AND the literal phrase "flight plan", AND the word "instruct" with a wildcard suffix, allowing for "instructor", "instructors", "instruction", "instructed", etc. These items can appear in any order in the text.

24.2.2 Validating a search-items String

You can use the **%iFind.Utils.TestSearchString()** method to validate a *search_items* string. This method enables you to detect syntax errors and ambiguous use of logical operators. For example, `"word1 AND word2 OR word3"` fails validation because it is logically ambiguous. Adding parentheses clarifies this string to either `"word1 AND (word2 OR word3)"` or `"(word1 AND word2) OR word3"`. **TestSearchString()** returns 1 to indicate a valid *search_items* string.

The following example invokes this iFind utility as an SQL function:

```
SELECT %iFind.TestSearchString('orange AND (lemon OR lime)')
```

TestSearchString() returns a %Status value: A valid *search_items* string returns a status of 1. An invalid *search_items* string returns an object expression that begins with 0, followed by encoded error information.

24.2.3 Fuzzy Search

iFind supports *fuzzy search* to match records containing elements (words or entities) that "almost" match the search string. Fuzzy search can be used to account for small variations in writing (color vs. colour), misspellings (collor vs color), and different grammatical forms (color vs. colors).

iFind evaluates fuzzy matches by comparing the Levenshtein distance between the two words. The Levenshtein distance is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other. The maximum number of single-character edits required is know as the maximum edit distance. The iFind maximum edit distance defaults to 2 characters. The maximum edit distance is separately applied to each element in the search string. For Basic iFind index, it is applied to each word in the search string. For Semantic iFind index, it is applied to each iKnow entity in the search string. (The examples that follow assume a Basic iFind index.)

For example, the phrase "analyse programme behaviour" is a fuzzy search match for "analyze program behavior" when maximum edit distance=2, because each word in the search string differs by an edit distance of (at most) 2 characters: analyse=analyze (1 substitution), programme=program (2 deletions), behaviour=behavior (1 deletion).

A word with that is lesser than or equal to the maximum edit distance is a fuzzy search match for any word with an equal or lesser number of characters. For example, if the edit distance is 2, the word "ab" would match any two-letter word (2 substitutions), any one-letter word (1 substitution, 1 deletion), any three-letter word containing either "a" or "b" (1 substitution, 1 insertion), and any four-letter word containing both "a" and "b" in that order (2 insertions).

- Fuzzy search is supported on all iFind index types: Basic, Semantic, and Analytic. On a Basic index it performs fuzzy search on individual words. On a Semantic index it performs fuzzy search on individual iKnow entities.

- Fuzzy search cannot be combined with wildcard search.

To specify fuzzy search for **search_index()** specify *search_option* as 3 for fuzzy search with the default edit distance of 2, or 3:n for fuzzy search with an edit distance specified as *n* characters. For example, setting 3:1 sets the edit distance=1, which in English is appropriate for matching most (but not all) singular and plural words. Setting 3:0 sets the edit distance=0, which is the same as iFind search without fuzzy search.

To specify fuzzy search for iFind methods, set the pSearchOption = $$$IFSEARCHFUZZY.

24.2.4 Stemming and Decompounding

Basic index, Semantic index, and Analytic index can all support stemming and decompounding. Stemming and decompounding are word-based, not iKnow entity-based operations. To enable an index for stem-aware searches, specify INDEXOPTION=1; to enable both stem-aware searches and decompounding-aware searches, specify INDEXOPTION=2.

24.2.4.1 Stemming

Stemming identifies the Stem Form of each word. The stem form unifies multiple grammatical forms of the same word. When using searchOption=1 at query time, iFind performs search and match operations using the Stem Form of a word, rather than the actual text form. By using searchOption=0, you can use the same index for regular (non-stemmed) searches.

24.2.4.2 Decompounding

Decompounding divides up compound words into their constituent words. iFind always combines decompounding with stemming; once a word is divided into its constituent parts, each of these parts is automatically stemmed. When using a decompounding search (searchOption=2), iFind compares the decompounding stems of the search term with the decompounded stems of the words in the indexed text field. iFind matches a decompounded word only when the stems of *any* of its constituent words match *all* constituent words of the search term.

For example, the search terms "thunder", "storm", or "storms" would all match the word "thunderstorms". However, the search term "thunderstorms" would not match the word "thunder", because its other constituent word ("storm") is not matched.

The iFind decompounding algorithm using a language-specific dictionary that identifies possible constituent words. This dictionary should be populated through the %iKnow.Stemming.DecompoundingUtils class. For example, by pointing it to a text column prior to indexing. You may also wish to exempt specific words from decompounding. You can exempt individual words, character sequences, and training data word lists from decompounding using %iKnow.Stemming.DecompoundUtils.

24.2.5 Languages Not Supported by the iKnow Engine

You can use iFind Basic indices to index and search texts in languages for which there is no corresponding iKnow language model.

Because stemming is not dependent on iKnow semantic indexing, you can also perform Basic index word searches on stem forms of words, if a stemmer is available. You must specify INDEXOPTION=1 or INDEXOPTION=2 to perform stem searches. For example, Italian is not an iKnow supported language, but Caché provides a %Text stemmer for Italian.

The following limitations and caveats apply to iFind searching for languages not supported by iKnow:

- An iKnow license is required to use this feature.

- The language must separate words using spaces. Languages that do not uses word separators cannot be searched. However, Japanese (which does not uses word separators) can be searched because iKnow provides a Japanese language model.

- Apostrophes do not separate words. iKnow recognizes contractions (such as "can't") and abbreviated verb forms (such as "there's") and separates them into two words, while ignoring apostrophes used for other purposes, such as possession ("John's"). Without iKnow support, iFind cannot separate contractions and abbreviations into separate words. You can compensate for this by pre-processing your text, inserting a blank space before or after apostrophes as needed.

- A UserDictionary cannot be applied to the texts before iFind indexing.

For further details, refer to Queries Invoking Free-text Search in the "Querying the Database" chapter of *Using InterSystems SQL*.

24.3 Highlighting

You can highlight words in a returned text using the *search_items* syntax. Highlighting syntax is:

```
(text,search_items,search_option)
```

search_items: Highlighting uses the same *search_items* syntax as searching. This allows you to use the same *search_items* value for both returning records and highlighting the strings within those records that caused them to be returned. This also allows you to use the **TestSearchString()** method to validate highlighting *search_items* syntax. However, because highlighting is applied to every instance of every match, highlighting ignores the *search_items* syntax AND, OR, and NOT logical operators in a *search_items* string.

search_option: The optional *search_option* is 0 (the default).

You can apply highlighting using either of the following:

*   SELECT item highlighting:

    SQL

    ```
    SELECT %iFind.Highlight(Narrative,'"visibility [1-4] mile*" AND "temp* ? degrees"')
    FROM Aviation.TestSQLSrch
    WHERE %ID %FIND search_index(NarrBasicIdx,'"visibility [1-4] mile*" AND "temp* ? degrees"',0,'en')
    ```

*   Utility method highlighting:

    You can use the **%iFind.Utils.Highlight()** method to perform an iFind Search and apply highlighting to the results.

By default, highlighting inserts the and (bold) XML tags at the appropriate places in the string. By default, highlighting is not case sensitive.

When used with Word Search, this method highlights separately each occurrence of each specified word.

When used with Positional Phrase Search, this method highlights each occurrence of the positional phrase.

24.4 iFind Examples

In the following examples, Basic index iFind syntax can be used with any type of iFind index. Semantic index iFind syntax requires a Semantic or Analytic iFind index.

These examples require that you have created and populated the Aviation.TestIFind table, as described in Indexing Sources for iFind Search earlier in this chapter.

For simplicity of display, these examples return record counts rather than the record text itself. These counts are the number of records that match the search criteria, *not* the number of matches found in the records. A record may contain multiple matches, but is only counted once.

24.4.1 Basic Search Examples

The following examples uses Basic index search to search the Aviation.TestSQLSrch table.

Search for records that contain at least one instance of the words "electrode", "plug", and "spark" (in any order):

```
SELECT COUNT(Narrative) FROM Aviation.TestSQLSrch
WHERE %ID %FIND search_index(NarrBasicIdx,'electrode plug spark',0)
```

Note that this is word search, not string search. Therefore, the following example may return different results, and may actually return more results than the previous example:

```
SELECT COUNT(Narrative) FROM Aviation.TestSQLSrch
WHERE %ID %FIND search_index(NarrBasicIdx,'electrodes plug spark',0)
```

Search for records that contain at least one instance of a word beginning with "electrode" (electrode, electrodes), and the word phrase "spark plug" (in any order):

```
SELECT COUNT(Narrative) FROM Aviation.TestSQLSrch
WHERE %ID %FIND search_index(NarrBasicIdx,'electrode* "spark plug"',0)
```

Search for records that contain a word beginning with "electrode" (electrode, electrodes), and the word phrase "spark plug" (in any order) within a co-occurrence proximity window of 6 words. Note the punctuation used to specify words and word phrases in a co-occurrence search:

```
SELECT COUNT(Narrative) FROM Aviation.TestSQLSrch
WHERE %ID %FIND search_index(NarrBasicIdx,'[electrode*,spark plug,1-6]',0)
```

Search for records that contain the two different word phrases `normal wear` and `"normal" wear`:

```
SELECT COUNT(Narrative) FROM Aviation.TestSQLSrch
WHERE %ID %FIND search_index(NarrBasicIdx,'"normal wear"',0)
```

```
SELECT COUNT(Narrative) FROM Aviation.TestSQLSrch
WHERE %ID %FIND search_index(NarrBasicIdx,'"\"normal\" wear"',0)
```

Search for records that contain at least one word containing the string seal (seal, seals, unseal, sealant, sealed, previously-sealed), and the word phrase "spark plug":

```
SELECT COUNT(Narrative) FROM Aviation.TestSQLSrch
WHERE %ID %FIND search_index(NarrBasicIdx,'*seal* "spark plug"',0)
```

Search for records that contain the wildcard phrase "wind from ? ? at ? knots." Possible values might include "wind from the south at 25 knots" and "wind from 300 degrees at ten knots." Note that if there is a space between two sequential question marks (? ?) the wildcard represents exactly two words; if there is no space between the two question marks (??) the wildcard represents from 0 to 6 words:

```
SELECT COUNT(Narrative) FROM Aviation.TestSQLSrch
WHERE %ID %FIND search_index(NarrBasicIdx,'"wind from ? ? at ? knots"',0)
```

The following example uses Basic index with Regular Expression search (with n=4). It searches records that contain occurrences of strings specifying dates between "January 10" and "January 29" inclusive:

```
SELECT COUNT(Narrative) FROM Aviation.TestSQLSrch
WHERE %ID %FIND search_index(NarrBasicIdx,'"January [1-2][0-9]"',4)
```

For further details, refer to Regular Expressions in *Using ObjectScript*.

24.4.2 Semantic Search Examples

The following examples use Semantic index iFind search to search the Aviation.TestIFind table.

Search for records that contain the iKnow entity "spark plug electrodes":

```
SELECT COUNT(Narrative) FROM Aviation.TestIFind
WHERE %ID %FIND search_index(NarrSemanticIdx,'{spark plug electrodes}',0)
```

Search for records that contain an iKnow entity ending with "spark plug" or "spark plugs":

```
SELECT COUNT(Narrative) FROM Aviation.TestIFind
WHERE %ID %FIND search_index(NarrSemanticIdx,'<{spark plug*}',0)
```

Search for records that contain both an iKnow entity ending with "spark plugs" and the iKnow entity "spark plugs":

```
SELECT COUNT(Narrative) FROM Aviation.TestIFind
WHERE %ID %FIND search_index(NarrSemanticIdx,'<{spark plugs} {spark plugs}',0)
```

A
Domain Parameters

This appendix lists the available domain parameters. Domain parameter names are case-sensitive. Each domain parameter has a %IKPublic macro equivalent (for example, $$$IKPFULLMATCHONLY). The recommended programming practice is to specify a domain parameter by its macro equivalent, not its parameter name. For information on setting domain parameters, refer to "Defining an iKnow Domain".

A domain parameter can be defined for a specific domain, or can be defined "system-wide," which means defined for all current and future domains in the current namespace. System-wide domain parameters defined prior to Version 4 (Caché 2014.1) are described in the Data Upgrade Utility section of the "iKnow Tools" chapter.

Domain parameters are divided into two groups, Basic and Advanced. Basic parameters are useful for customizing iKnow default behavior. Advanced parameters significantly change iKnow behavior and performance, and should be used with caution.

Table I–1: Basic Domain Parameters

Table I–1: Basic Domain Parameters

DefaultConfig	$$$IKPDEFAULTCONFIG: A string specifying the name of the iKnow Configuration used for the domain. By default, no Configuration is assigned to a domain; this is indicated by returning the string DEFAULT. You can specify the name of any Configuration defined in the current namespace. The Configuration must exist when you assign it to this parameter. If the Configuration does not exist, this parameter remains unchanged. The DefaultConfig value can be overridden by **SetConfig()** or **ProcessBatch()**.
EnableNgrams	$$$IKPENABLENGRAMS: A boolean parameter. If set to 1, iKnow generates n-grams for the domain. n-Grams are used for Similarity entity matching within words. If set to 0, iKnow matches only parts, whole words or the beginning letters of words. The default is 0. At the domain level, you can only change the EnableNgrams setting for an empty domain (a domain the does not yet contain iKnow data). At the system-wide level, you can only change the EnableNgrams setting if there are no domains containing text data in the current namespace. n-Gram matching greatly increases the size of the data stored by iKnow, and can have a significant performance impact. It should only be enabled when required. You should not use n-gram matching with most languages. However, matching operations on German text often requires n-gram matching.
IgnoreDuplicateExtIds	$$$IKPIGNOREDUPLICATEEXTIDS: A boolean parameter. If set to 1, iKnow does not log an error if a source is loaded that has the same external Id as an already-loaded source. 1 is the recommended setting when loading sources from a location previously loaded in order to include the new sources added since the last load. The default is 0.
IgnoreEmptyBatch	$$$IKPIGNOREEMPTYBATCH: A boolean parameter. If set to 0, the Loader issues a $$$IKNothingToProcess error when a batch load is specified that specifies no sources. If set to 1, the Loader does not generate this error. The default is 0.

MAT:DefaultProfile	$$$IKPMATDEFAULTPROFILE: This parameter takes the name of a user-defined matching profile that you wish to establish as the default for the domain. The matching profile must exist when you assign it to this parameter. If the user-defined matching profile is defined as namespace-wide (not specific to a domain), you must add a zero colon (0:) preface to the name. For example, `"0:NoDomainProfile"`. If you do not set this parameter, the iKnow default matching profile is used as the domain default. For further details, refer to Defining a Matching Profile in the "Smart Matching: Using a Dictionary" chapter. You can override the domain default matching profile by specifying a custom matching profile in the **MatchSource()** or **MatchSources()** methods.
MAT:SkipRelations	$$$IKPMATSKIPRELATIONS: A boolean parameter for dictionary matching. A value of 1 (the default) specifies that only concepts, not relations, will be matched during entity matching. (Relations *are* matched during CRC and path matching operations.) Skipping relation entity matching can significantly improve performance. A value of 0 performs relation entity matching. You should only set this parameter to 0 if your dictionary includes single-entity terms that are relations.

SortField	$$$IKPSORTFIELD: A boolean parameter. Every iKnow entity has two integer counts associated with it: frequency (number of occurrences of an entity in all sources) and spread (number of sources in which the entity occurs). If set to 0, the iKnow default is to sort by frequency ($$$SORTBYFREQUENCY). If set to 1, the iKnow default is to sort by spread ($$$SORTBYSPREAD). The default is 0. At the domain level, you can only change the SortField setting for an empty domain (a domain the does not yet contain iKnow data). At the system-wide level, you can only change the SortField setting if there are no domains containing text data in the current namespace.
	Alternatively, you can change this sort order domain-wide by specifying a second parameter (sortField) to the **Create()** or **GetOrCreateId()** method of the %iKnow.Domain class. 0=sort by frequency (the default). 1=sort by spread. The sort order default should only be changed when required. You can only change the sort order default for an empty domain (a domain that does not yet contain iKnow data).
	In certain individual queries you can also specify sort order ($$$SORTBYFREQUENCY or $$$SORTBYSPREAD) or use the current domain default ($$$SORTBYDOMAINDEFAULT).
Status	$$$IKPSTATUS: A boolean parameter. If set to 1, iKnow displays detailed status information on the progress of the source loading process. If set to 0, iKnow does not display this information. The default is 0.

Stemming	$$$IKPSTEMMING: A boolean parameter. If set to 1, NLP activates stemming on the domain. If set to 0, NLP does not perform stemming. The default is 0.

Table I–2: Advanced Domain Parameters

EntityLevelMatchOnly	$$$IKPENTITYLEVELMATCHONLY: A boolean parameter that is used to limit the types of match operations performed when matching against a dictionary. By default iKnow matches entities, CRCs, paths, and sentences. You can set the this parameter to limit matching to entities only. Note that this may result in a much larger number of match results. The default is 0. Changing this parameter affects any subsequent match operations for this domain. Therefore, sources already matched before you changed this parameter must be explicitly re-matched to reflect this change.
FullMatchOnly	$$$IKPFULLMATCHONLY: A boolean parameter that is used to limit the types of match operations performed when matching against a dictionary. You can set this parameter to restrict matching to exact matches only. When this option is set (1), partial matches and disordered matches are ignored. The default is 0. Changing this parameter affects any subsequent match operations for this domain. Therefore, sources already matched before you changed this parameter must be explicitly re-matched to reflect this change.
LanguageFieldName	$$$IKPLANGUAGEFIELDNAME: A string that specifies the name of a metadata field. When set to an existing metadata field that was populated during source loading, iKnow uses that metadata field's value (if set) as the language to be used when processing the corresponding source. This option overrides automatic language identification. The metadata field value must be a two-letter ISO language code (see $$$IKLANGUAGES) for a language that has been specified in the current configuration object.

MAT:StandardizedForm	$$$IKPMATSTANDARDIZEDFORM: To enable standardized-form matching for a domain, set this domain parameter to the desired standardization function, as follows: `SET stat=domain.SetParameter("MAT:StandardizedForm","%Text")`. Standardized-form matching supports matching different forms of the same word in source text with a single dictionary term, for example singular/plural or verb forms, using the standardized form of the dictionary term. Any dictionary terms created in such a domain result in "standardized" dictionary elements, which are at match-time compared to the standardized forms of the entities in the sources. For the right standardization algorithm to be used, it's important dictionary terms are created with the right language annotation (parameter in **CreateDictionary()** methods) and sources are assigned the proper language (either through selecting the appropriate language model in a Configuration, or by using Automatic Language Identification).
MetadataAPI	$$$IKPMETADATAAPI: A string that specifies which metadata API class to use to extend %iKnow.Queries.MetadataI. The default is %iKnow.Queries.MetadataAPI. You can only change the MetadataAPI setting for an empty domain (a domain the does not yet contain iKnow data).
QUERY:MinTopConceptLength	$$$IKPMINTOPCONCEPTLENGTH: An integer that specifies the smallest concept (fewest number of characters) that a GetTop() query can return. This parameter is used to filter meaninglessly short concepts from the GetTop() result. The default is 3, specifying that concepts that are 3 letters in length or larger are returned by GetTop(). This minimum character count is inclusive of spaces between words and punctuation symbols in a concept.
SimpleExtIds	$$$IKPSIMPLEEXTIDS: A boolean parameter that is used to specify the format for external IDs for sources. If set to 0, iKnow stores the full reference as the external ID. If set to 1, iKnow stores the local reference as the external ID. The default is 0. You can only change the SimpleExtIds setting for an empty domain (a domain the does not yet contain iKnow data).
SkipExtIdCheck	$$$IKPSKIPEXTIDCHECK: A boolean parameter that specifies whether to check for duplicate external IDs. If set to 1, iKnow skips checking whether a duplicate external ID already exists when loading sources. If set to 0, iKnow checks for duplicate external IDs. The default is 0.

B

DeepSee Cube Integration (Deprecated Form)

This appendix describes how to use older, deprecated form of integration between iKnow and DeepSee cubes. It discusses the following topics:

- Overview
- How to generate iKnow source metadata fields from an associated DeepSee cube
- How to generate DeepSee cube classes for an iKnow domain
- Additional sources of information

For information on using the newer form of integration, see "Using Unstructured Data in Cubes" in the *Advanced DeepSee Modeling Guide*.

Also see the chapter "iKnow KPIs and DeepSee Dashboards."

B.1 Overview

This section provides an overview of the older, deprecated form of integration between iKnow and DeepSee cubes.

DeepSee is an analytics engine and reporting platform that is built into Caché in the same way that iKnow is. The primary purpose of DeepSee is to enable you to create interactive dashboards that you can embed on Zen pages or that users can access via the DeepSee User Portal (which is intended for all end users, not just end users of DeepSee).

B.1.1 Background

A DeepSee *cube* is a data model that enables you to create DeepSee queries by drag and drop actions. The queries are known as *pivot tables*. You create pivot tables so that you can display them on DeepSee dashboards. When the dashboard is displayed, the query is executed.

Each cube is based on a source table. You build a cube, and this process generates a *fact table* that has the same number of records as the source table (or possibly fewer if the model skips some records deliberately). The queries use the fact table rather than the source table. DeepSee provides multiple complementary mechanisms to keep the fact table synchronized with the source table.
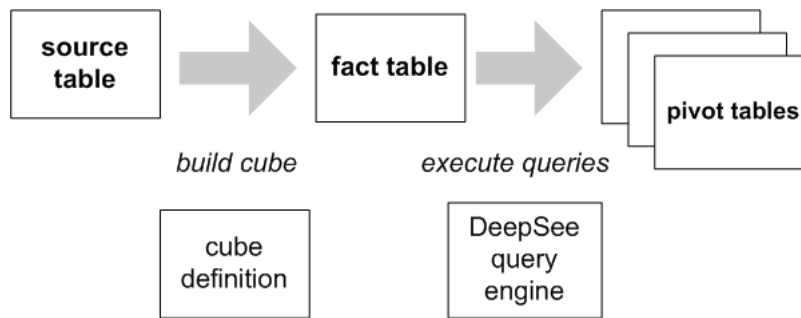
The following figure shows an example pivot table that shows the number of patients and the average allergy count per patient, grouped by age and gender.

Age Group		Female		Male	
		Patient Count	Avg Allergy Count	Patient Count	Avg Allergy Count
0 to 29	0 to 9	680	0.60	750	0.63
	10 to 19	756	0.66	769	0.69
	20 to 29	661	0.64	648	0.61
30 to 59	30 to 39	815	0.63	735	0.65
	40 to 49	728	0.68	741	0.61
	50 to 59	586	0.58	552	0.62
60+	60 to 69	397	0.64	319	0.68
	70 to 79	304	0.58	242	0.56
	80+	217	0.57	100	0.66

Using this example as a reference, let us discuss the key terms in DeepSee cubes:

- A *level* is used to group the records. In this case, we are grouping patients. Each record in the source table corresponds to one patient, as does each record in the fact table.

- A *measure* is a value displayed in the body of the pivot table. It is based on values ultimately taken from the source data. For a given context, a measure aggregates the values for all applicable records and represents them with a single value (typically by summing them).

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

A cube can also define *detail listings*, whose purpose is to display lowest-level data associated with the current context. To access a listing, you select one or more cells and click the Listing button (not shown in the previous picture). DeepSee then displays a table that lists selected fields for all the records used in the current context.

B.1.2 Relationship between a Cube and iKnow Sources

The iKnow/cube integration makes the following assumption: there is a single iKnow source associated with each record of the cube's source class. That is, the cube serves as additional information about the iKnow sources. (Phrased differently, the iKnow sources contain additional information associated with each record of the cube's source class.)

The goal is to be able to use DeepSee and iKnow filters interchangeably where possible.

You can use utility methods to do the following tasks:

- To generate iKnow source metadata fields from an existing DeepSee cube

- To generate DeepSee cube classes for an iKnow domain

B.2 Generating iKnow Source Metadata Fields for an Associated DeepSee Cube

If a DeepSee cube uses a source class that includes a reference to iKnow data source, you can use the levels of that cube as iKnow source metadata fields for those sources. This means that you can use DeepSee filters within iKnow queries (including when you access those queries via the DeepSee KPI mechanism as described in "Using Unstructured Data in Cubes" in the *Advanced DeepSee Modeling Guide*).

This section assumes that the cube is defined and has been built. The cube does not need to be modified or rebuilt.

B.2.1 Prerequisites

Before you can generate iKnow source metadata fields for a cube:

1. Load the iKnow sources.

    When you do so, make sure that the external ID of each source can be readily associated with a specific record of the source class of the cube.

    For example:

    *   If you are using a file lister, you might make sure that the file names match the patient identifiers.

    *   If you are using an SQL lister, you might make sure that the IDs of the records match the patient identifiers.

2. Make sure that the source class of the cube includes a property whose value is the external ID of the corresponding iKnow source.

    For example, this class could include a calculated property whose value follows the naming convention established in the preceding step.

    ```
    Property DocumentId As %String [ Calculated ];

    Method DocumentIdGet() As %String
    {
     quit ":FILE:c:\patient-data\patient"_..PatientNumber_".txt"
    }
    ```

B.2.2 Generating the Metadata Fields from the Cube

To generate the metadata fields, you use the **GenerateMDFieldsFromDSDims()** method of %iKnow.DeepSee.Utils. This method has the following signature:

```
ClassMethod GenerateMDFieldsFromDSDims(domainId As %Integer,
                               cubeClassName As %String,
                               ByRef levels = 1,
                               extIdProperty = "",
                               killExistingMDFields As %Boolean = 0) As %Status
```

Where:

*   *domainId* is the integer ID of an iKnow domain.

*   *cubeClassName* is the name of the cube class.

*   *levels* controls which levels are processed. If you omit this argument, the system processes all levels, skipping the ones that are not supported by the iKnow engine.

This argument, if specified, must be a multidimensional array of the following form:

Node	Value
arrayname(*level_name*) where *level_name* is the complete MDX identifier of the level, in double quotes. For example: `"[DocD].[H1].[Doctor]"`	Either 0 or 1. If you specify 1, the iKnow metadata field is generated as a bitstring, which is suitable when you expect only a small number of values.

- *extIdProperty* is the property you added in the previous subsection. For example: `DocumentId`

  More generally, *extIdProperty* is a property (in the cube's source class) whose value equals the external ID of an iKnow source in the given domain.

- *killExistingMDFields* specifies whether to remove any existing iKnow metadata fields.

B.2.3 Using DeepSee Filters with an iKnow KPI

If you also create iKnow KPIs in this domain, then you can use DeepSee filters in the KPI, provided that the KPI and the cube are both associated with the same iKnow domain.

To use this mechanism:

- Display the KPI in a widget on a dashboard.

- In the DeepSee Analyzer, create a pivot table.

- On the same dashboard, add a widget that displays the pivot table. Configure this widget to use one or more levels of the cube as filters, and specify the filter target as `*` (all widgets).

B.3 Generating DeepSee Cubes for an iKnow Domain

This release provides a preliminary version of a utility that generates DeepSee cubes from iKnow engine results. The generated cubes enable you to view entities, entity occurrences, and match results grouped in various ways. With a DeepSee cube, you can perform analysis and exploration via drag and drop actions in the DeepSee Analyzer.

You can use these DeepSee cubes in the same way that you use other DeepSee cubes; you can explore them in the Analyzer, create pivot tables, and add those pivot tables to DeepSee dashboards.

This section describes how to generate and (briefly) how to use DeepSee cubes, given an iKnow domain. It discusses the following topics:

- How to generate the cubes

- A brief look at the cubes

- How to rebuild the cubes

If you can achieve a required result through an existing iKnow query, use that query rather than trying to create an equivalent query with the generated cubes. The iKnow queries are optimized to use internal and precalculated indices and are faster as a consequence.

B.3.1 Generating the Cubes

To generate DeepSee cubes for an iKnow domain, you use the **GenerateSourceObjectAndCubes()** method of
%iKnow.DeepSee.Utils. This method has the following signature:

```
classmethod GenerateSourceObjectAndCubes(domainId,
                                        packageName="User",
                                        createCube=2,
                                        overwriteExisting=0,
                                        buildCubes=0) as %Status
```

Where:

- *domainId* is the integer ID of an iKnow domain.

- *packageName* is the package in which to place the generated classes.

- *createCube* controls which classes are generated:

  - If *createCube* is 0, the system generates and compiles the classes on which the cubes are based, but does not generate other classes. You might use this option if you wanted to create DeepSee cubes manually in the DeepSee Architect.

    These classes define mappings to access the iKnow data.

  - If *createCube* is 1, Caché also generates and compiles a cube that represents the iKnow sources.

  - If *createCube* is 2, Caché also generates and compiles a cube that represents unique entities, and another cube that represents entity occurrences.

  - If *createCube* is 3, Caché also generates and compiles a cube that represents matching results.

- *overwriteExisting* specifies whether to overwrite any existing classes.

- *buildCubes* specifies whether to build the generated cubes. This has no effect if *createCube* is 0.

  The phrase *build a cube* refers to the process of populating the structures that the cube uses. This process makes it possible to use the cube.

B.3.2 A Brief Look at the Analyzer and the Cubes

This section briefly introduces the Analyzer and the generated cubes. To access the Analyzer and display one of these cubes:

1. Click **Home,DeepSee,Analyzer**.

2. Click the Change button .

3. Click the cube name.

4. Click **OK**.

For example, the entity occurrence cube provides information about the occurrence of entities in the source documents. The name of this cube ends with `EntityOccurrences`. When you view this cube, the left area of the page displays its contents, as follows:
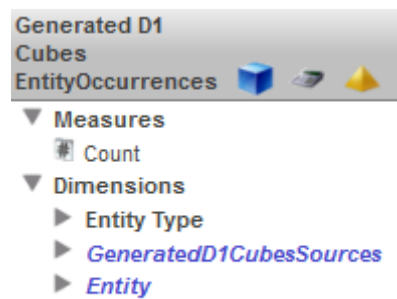
This cube defines one measure, Count, which counts entity occurrences. This measure is used by default.

The **Dimensions** section shows three dimensions. GeneratedD1CubesSource and Entity are displayed differently from Entity Type because they are defined differently. You can use this dimensions as follows:
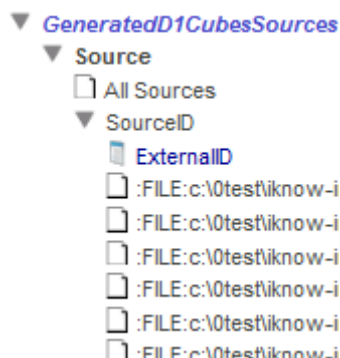
- Entity Type — This dimension enables you to categorize entity occurrences by entity type. If you expand this folder and the Entity Type folder within it, the system displays the following:
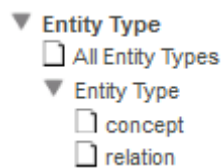

- GeneratedD1CubesSource — This dimension enables you to categorize entity occurrences by the source document to which they belong. If you expand this folder, the system displays the following:
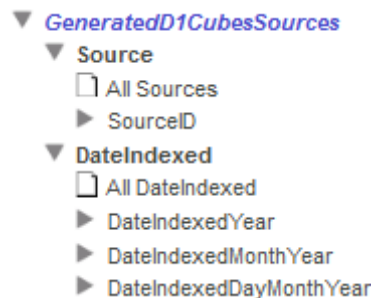

If you expand the SourceID folder, the system displays something like this, depending on your data:
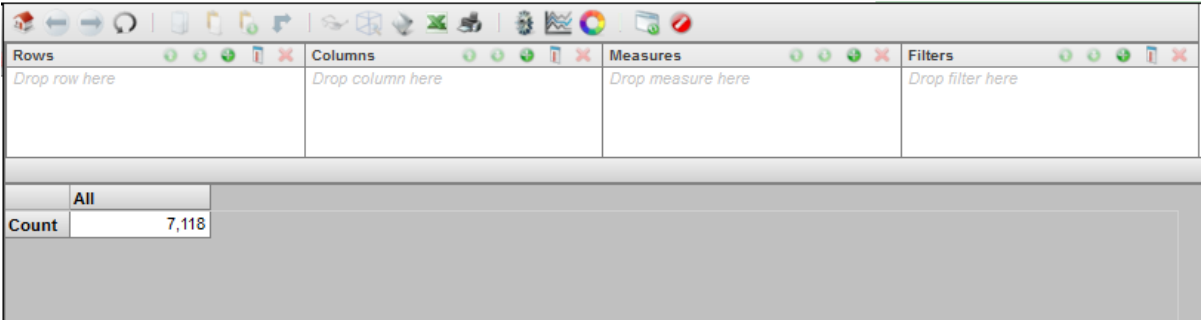

- Entity — This dimension enables you to categorize entity occurrences by entity value. If you expand this folder and the top-level folders in it, the system displays the following:
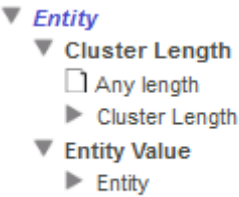
If you expand the `Entity` folder, the system displays something like this, depending on your data:
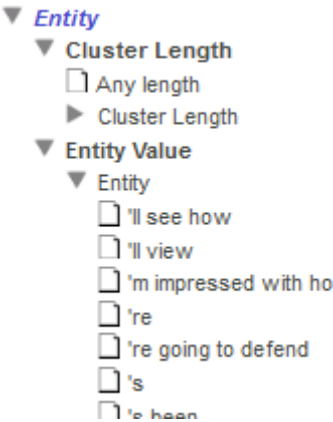
The right area of the page looks like this:

The area at the bottom is a simple pivot table. It displays the total value for `Count`. In this example, there are 7118 entity occurrences in the domain.

The area above that enables you to create pivot tables. You can drag and drop items from the left to this area, and the system automatically modifies the pivot table. For example, if you drag `Entity` (the inner `Entity` folder) to the **Rows** box, the system displays the following:

Entity	
'll see how	1
'll view	1
'm impressed with how	1
're	1
're going to defend	1
's	11
's been	1
's close to	1

This pivot table shows both relations and concepts because we have not applied any filtering. To filter the pivot table to show only concepts:

1. Expand the **Entity Type** dimension on the left:
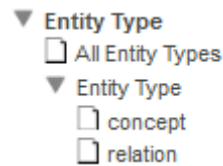
2. Drag **concept** and drop it into the **Filters** box on the right.

    This immediately updates the display.

Now you might see something like this:

The `EntityOccurrences` cube defines a detail listing. To access it, select one or more cells and click the Listing button

. (In this example, we click the cell for `ancient viruses`.) The system then displays something like the following:

#	EntOccId	EntTypeId	EntUniId	EntityValue	CorpusFrequency	CorpusSpread	SourceId	ExternalId
1	5420	0	253	ancient viruses	2	1	23	:FILE:c:\0test\iknow-input1\news3.txt
2	5582	0	253	ancient viruses	2	1	23	:FILE:c:\0test\iknow-input1\news3.txt

B.3.3 Rebuilding the Cubes

After you process more sources for this iKnow domain, it is not necessary to regenerate the classes but it is necessary to rebuild the DeepSee cubes. It is also necessary to rebuild the cubes in a specific order:

1. Rebuild the sources cube, which is the cube whose name ends in `Sources`.

2. Rebuild the other cubes, in any order.

To rebuild a cube, open it in the Architect and click **Rebuild**.

B.4 See Also

- For information on Zen pages, see *Using Zen*.

- For information on creating DeepSee dashboards, see *Creating DeepSee Dashboards*.

- For information on accessing dashboards from your application, see "Accessing Dashboards from Your Application" in the *DeepSee Implementation Guide*.

- For information on packaging dashboard definitions into classes, see "Packaging DeepSee Elements into Classes" in the *DeepSee Implementation Guide*.

- For information on the DeepSee User Portal and dashboards, see the *DeepSee End User Guide*.

- For information on the DeepSee Architect, see *Creating DeepSee Models*.