Architecture for Integrated and Dynamic Data Analysis (AIDA) Server Guide

Summary

The Architecture for Integrated and Dynamic Data Analysis (AIDA) application has two components, namely a user *client* and a *server*. The server, described in this documentation, prepares data for analysis by the client. The intent of this document is to provide a general overview of how the server accomplishes its main tasks. Although all object classes will not be described in this document, the document will discuss important classes that allow AIDA's Server to properly function in order to produce a searchable output. Specific details on objects within the server can be found in /Server/docs.

Table of Contents

1. Introduction

The Architecture for Integrated and Dynamic Data Analysis (AIDA) application has two components, namely a user client and a network server. The client allows analysts to work with the system. The server prepares data for analysis by the client. This document describes the server component of the AIDA application. The server is responsible for retrieving documents; parsing and analyzing them; and producing the cache used by the AIDA Client. More particularly, the server reads data from sources such as newspapers (e.g., the *New York Times*) and Really Simple Syndication (RSS) service feeds; extracts their content; identifies keywords in that content; indexes the content; and then writes the indexed content and keywords to the cache for later analysis.

2. Architecture

The server takes a component-based approach to this process where each component handles a different aspect (i.e., reading, keyword extraction, etc.). These components are tied together and run using the Unstructured Information Management Architecture or UIMA (http://incubator.apache.org/uima/index.html). The UIMA documentation should be read in conjunction with this document.

UIMA enables applications to be decomposed into components, for example "language identification" → "language specific segmentation" → "sentence boundary detection" → "entity detection" (person/place names, etc.). Each component implements interfaces defined by the framework and provides self-describing metadata via XML descriptor files. The framework manages these components and the data flow between them. Components are written in Java or C++; the data that flows between components is designed for efficient mapping between these languages.

UIMA itself doesn't provide concrete components but rather enables a wide variety of components to be run together in a structured fashion. UIMA also provides a key-value pair parameter system for setting and retrieving component parameters. In a typical UIMA application, each document passes through reading, analysis and consumption phases. A final step then generates the cache itself. The phases are as follows:

1. In the reading phase, the document is parsed and its content is extracted. The document is also tagged with additional metadata such as its location (URL), author (if available), title and publication date.
2. In the analysis phase, the document content is analyzed using a UIMA analysis engine that wraps GATE (General Architecture for Text Engineering) (http://gate.ac.uk/), tagging relevant keywords: diseases, sanctioned entities, and so forth.
3. In the consumption phase, the document is "consumed" into a Lucene (http://lucene.apache.org) index for term (i.e., word and phrase) indexing.
4. Once all the documents have been processed, the cache is generated from the Lucene index.

The first two phases annotate the documents with particular tags. When a document is read, metadata tags are added that apply to the entire document. In the second phase, particular words or phrases are tagged, identifying the word or phrase as a keyword. For example, any mention of *sunflower* or *helianthus* will be tagged as *Sunflower*. These tags

are represented as UIMA types and in the code itself as Java objects. In the third phase the document is indexed. Once all the documents are processed, the index itself is processed to become the cache. Figure 1 shows the overall document workflow.

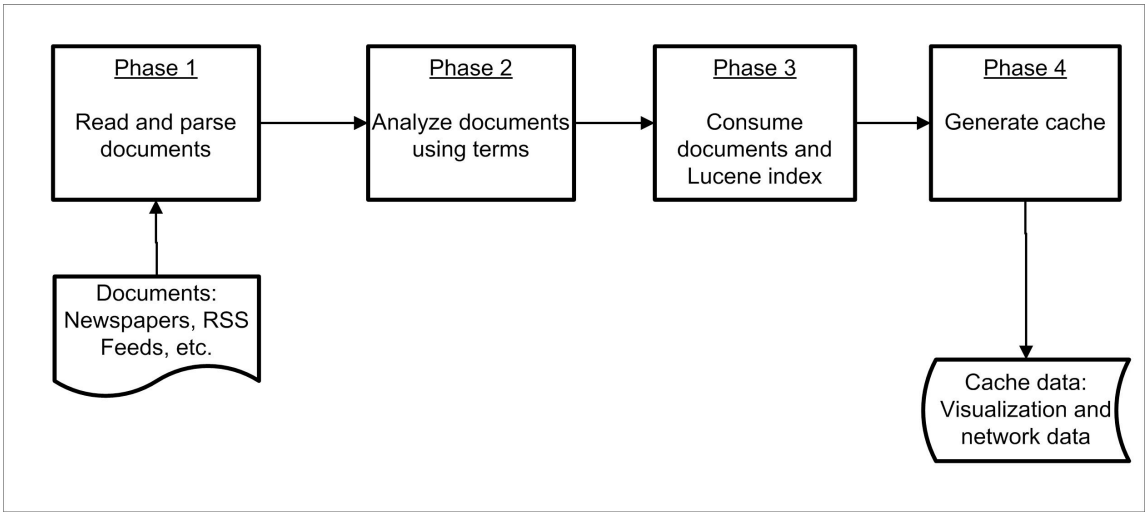

Figure 1. Workflow.

3. Collection Reading

Collection reading is performed by the anl.aida.reader.AIDACompositeReader class, which applies (i.e., extends in Java) an UIMA collection reader that delegates the actual reading of documents to classes that implement an anl.aida.reader.AIDAComponentReader interface. The AIDACompositeReader reads the component classes to use and instantiates those classes during initialization (Figure 2). The AIDACompositeReader passes itself to each component's initialize method and any UIMA-style parameters set on it can be retrieved by the component.

The readers iterate over an index file that describes what they should be reading (see /Server/scenarios/data/dailygate_index.txt for an example and other text files in the same directory). The index file has the following format:

"timestamp:::url:::title:::author".

The ":::" is the delimiter between the timestamp and other fields. The timestamp is a long value that can be used in a java.util.Date constructor to create a Date. The author field is optional. The classes StandardIndexLineParser and IndexIterator (in anl.aida.util) are used to parse individual lines and iterate through entire indices. The index files are generated by Java classes that are given set dates to find relevant files (e.g., see anl.aida.reader.local.DailyGateIndexMaker.java and other similar IndexMaker classes in anl.aida.reader and anl.aida.reader.local). These classes parse websites or archival data to create a cache or parse RSS feeds to create a live cache. The AbstractAIDAComponentReader (which the individual AIDAComponentReaders extend) works with index files and does much of the iteration on the index file references. Subclasses need only do the actual reading of the link (Figure 2). See anl.aida.reader.local.DailyGateReader.java for an example.

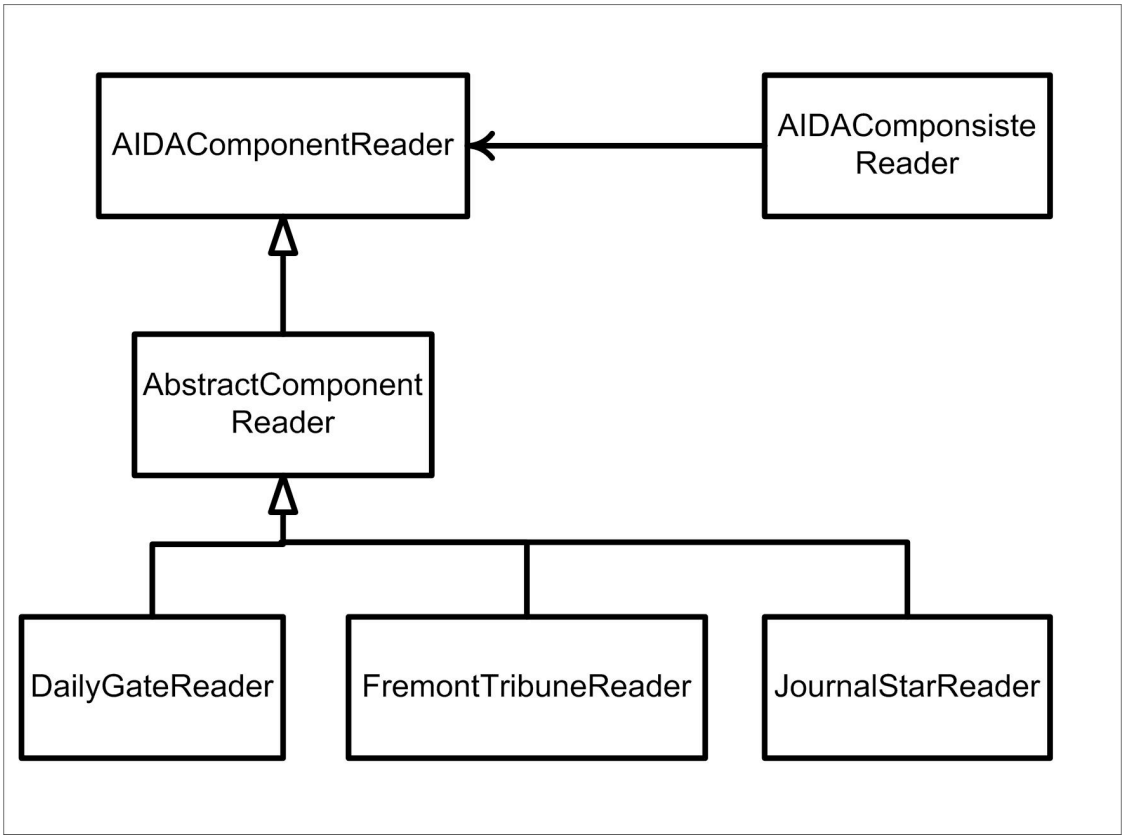

Figure 2. Reader architecture UML class diagram.

4. Document Analysis

Once the document has been retrieved and its content identified, the document is passed to the analysis component. Document analysis is performed by UIMA analysis engines. AIDA uses an analysis engine, the Gate Annotator, that will run a GATE application and translate back and forth between UIMA and GATE annotation tags.

4.1. GATE

AIDA utilizes GATE (General Architecture for Text Engineering) (http://gate.ac.uk/) for its Natural Language Processing (NLP). GATE is free and open source (LGPL License) and has been developed by language engineering researchers at the University of Sheffield (UK) since 1995. GATE uses Language Resources (LR), Processing Resources (PR) and Visual Resources (VR) to analyze and annotate textual documents.
GATE comes with an integrated set of such resources known as CREOLE (Collection of REusable Objects for Language Engineering) which can be used directly or freely modified as necessary.

Typically GATE PRs are combined into analysis pipelines, known as GATE Applications. The GATE Applications can take in a document or collections of documents for analysis. As documents are passed from PR to PR in the GATE Application, they are annotated with GATE annotations. Some PRs simply add annotations to previously un-annotated text while other PRs modify existing annotations.

When documents emerge from a GATE Application they will contain the annotations resulting from the sequence of PRs which operated on them. In AIDA, the un-annotated document content is passed to GATE which then parses the content and performs keyword identification and annotation tagging. Below, the GATE Application and the PRs used in AIDA are described.

Standard LRs in GATE include text, ontologies, and other text corpora. GATE has the ability to import text with the following formats: plain text, HTML, SGML, XML, RTF, Email, PDF, and Microsoft Word documents. AIDA loads plain text LRs exclusively, passing a document's content from the collection reader components to GATE.

GATE includes a default set of PRs, the ANNIE (A Nearly-New Information Extraction) system. AIDA applies the AnnotationDeletePR, DefaultTokeniser, SentenceSplitter, DefaultGazetteer, and ANNIETransducer PRs for document annotation.

- AnnotationDeletePR: This PR removes all previous annotations from LRs.

- DefaultTokeniser: This PR is the ANNIE English Tokeniser. A tokeniser splits text into simple tokens (e.g., numbers, punctuation, words). According to the GATE documentation:

  > "[T]he English Tokeniser is a processing resource that comprises a normal tokeniser and a JAPE transducer ... The transducer has the role of adapting the generic output of the tokeniser to the requirements of the English part-of-speech tagger. One such adaptation is the joining together in one token of constructs like " 'm", " 're", " 'til", " 've", etc. Another task of the JAPE transducer is to convert negative constructs like "don't" from three tokens ("don", " ' " and "t") into two tokens ("do" and "n't")..."

- SentenceSplitter: This PR splits the text into sentences, utilizing a gazetteer list of abbreviations to distinguish end-of-sentence punctuation from other types of punctuation.

- DefaultGazetteer: This PR annotates text (with "Lookup" annotations) based on plain text lists of words. An index file (/Server/scenarios/data/gazetteer/lists.def) is issued to locate the set of lists to be used. The index file lists the file name of the list (e.g., /Server/scenarios/data/gazatteer/agriculture.lst; see other lists in the same directory), followed by the "major type" to associate with members of the list and optionally also a "minor type" to associate with members of the list. Within each word list file, in addition to simply listing words separated by newline characters, there is also the ability to include annotation feature specifications with the use of a GazetteerFeatureSeparator. AIDA uses the GazetteerFeatureSeparator to specify the canonical names for terms. For example, the lines:

  sunflower%cName=Sunflower
  helianthus%cName=Sunflower

identify the terms *sunflower* and *helianthus* with the canonical name *Sunflower*, where the feature separator is the "%" character and the GATE feature name is "cName." AIDA reads the gazetteer lists but ignores capitalization so, for example, terms like *Helianthus* are annotated properly.

- ANNIETransducer: This PR uses JAPE (a Java Annotation Patterns Engine) rules to annotate text with entity annotations. With JAPE, patterns over annotations can be defined and matched in annotated text, allowing for general entity extraction. For example, the JAPE rule:

```
Rule: Agriculture
(
{Lookup.majorType == agriculture}
)
:agriculture -->
:agriculture.Agriculture = {kind = "Agriculture",
canonicalName = :agriculture.Lookup.cName, rule =
"Agriculture"}
```

creates an Agriculture annotation with the "kind" feature equal to "Agriculture," the "sub_kind" feature equal to the minorType of the matched Lookup annotation, and the "canonicalName" feature equal to the cName feature of the matched Lookup annotation. See the main jape file (/Server/scenarios/data/jape/main.jape), which references all the jape files (located in the /Server/scenarios/data/jape directory) used in this step.

In AIDA, the ANNIE PRs are used in the sequence listed here and are saved as a GATE Application (see /Server/gate_app/application.xgapp) in Figure 3.

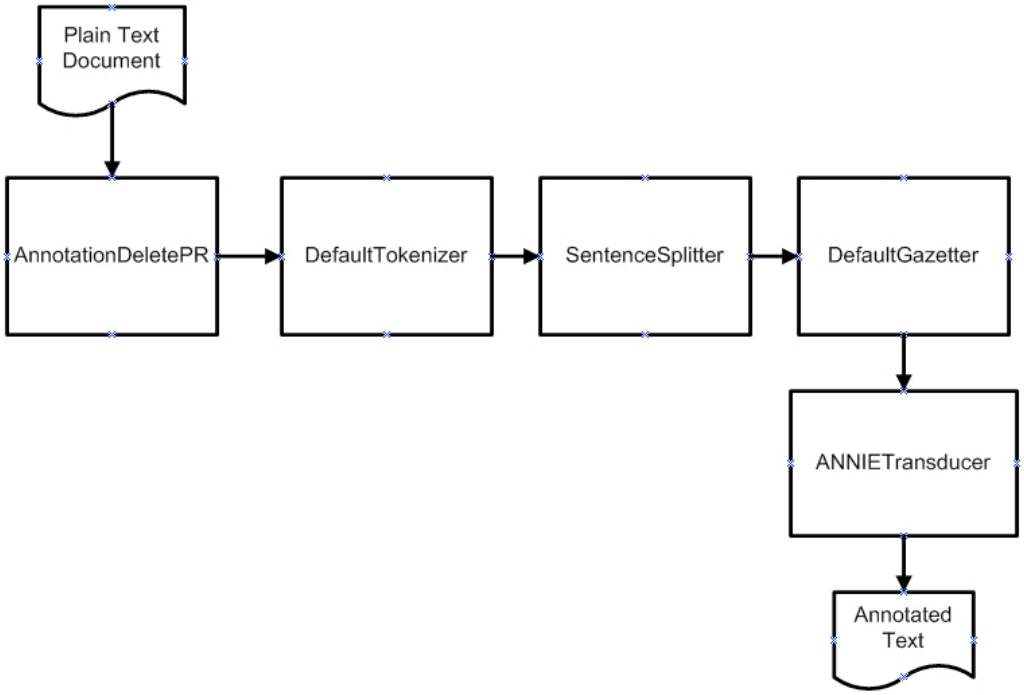

Figure 3. AIDA Gate application processing.

Testing of the PRs was done using the GATE VRs. The VRs used to develop the AIDA capabilities include GAZE (Gazetteer Visual Resource) and the Document Editor with Annotation Views. For more information on the details of GATE, see the GATE User Guide (http://gate.ac.uk/sale/tao/split.html).

4.2. Data for GATE Processing Resources

The gazetteer lists and entity extraction rules used by the GATE PRs can be derived from a number of sources. In the example case study, there are four categories of entities that AIDA detects via gazetteer lists and JAPE:

1. Urban and Transportation
2. Agriculture
3. Forestry and Grassland
4. Other

These categories were constructed by conducting an initial search on archived newspaper articles (i.e., articles covering from 2006-2009). These articles produced common terms on land use topics that added to our initial list of terms. In addition, we apply WordNet's (http://wordnetweb.princeton.edu/perl/webwn) online search engine, using anl.wordnet.reader.WordNetReader, that finds synonyms (i.e., synsets) to terms found in the initial searches. Alternative approaches one can apply are the inclusion of existing databases and terminology lists for specific fields that can make searches more focused on topics.

5. Mapping Between UIMA and GATE

AIDA uses an analysis engine, the Gate Annotator, that will run a GATE application and translate back and forth between UIMA and GATE annotation tags. The descriptor file is located at /Server/desc/GateDescriptor.xml. The parameters for the GATE annotator can be found in the parameter section of the appendix.

5.1. Input and Output Mapping

The input and output section describes how annotations should be translated into GATE and UIMA style annotations. These annotations are added to the document prior to being processed by the GATE and UIMA applications. Each UIMA element describes how particular annotations and their features should be mapped to GATE annotation type and feature maps. The details of the mapping are shown in Table 1.

Table 1. UIMA and GATE mapping.

Attribute	Description	Required
type_name	The name of the UIMA annotation to translate to a GATE annotation.	YES
annotation_set	The name of the GATE annotation set to add the GATE annotations to. If not specified then the default set is used.	NO
gate_type_name	The name of the GATE type to create from the UIMA type_name.	YES

Table 2 describes how to apply feature mapping between UIMA and GATE feature maps.

Table 2. UIMA to GATE input feature mapping.

Attribute	Description	Required
name	The name of the UIMA annotation feature.	YES
gate_feature_name	The name of the GATE feature.	YES
type	The type of the feature. Valid values are: string, float, double, long, int, boolean.	YES

In the above description, a GATE annotation of type POS will be created for each annotation in the current document of type GeniaPOSTag. The value of each POSTag's "value" feature will be added to the corresponding POS annotation's feature map with a feature key of "value." In addition, the annotation is tagged as "updated," insuring that it will be properly updated on output.

In the above example, a new DTType annotation will be created for each GATE annotation of type DT. The feature map value for each DT annotation's "confidence" feature will be set on the corresponding DTType's "conf" feature. In addition, the existing "GeniaPOSTag" annotations will be updated from their corresponding GATE "POS" annotations such that the value of the POS annotation's "value" feature will be set on the corresponding GeniaPOSTag "value" feature. The actual mapping file used in the case study (/Server/scenarios/data/gate_mapping.xml) is currently relatively simple; all mapping is done in the "new" tagged section of the xml file. The example file maps the searched

categories tagged in documents by GATE to their UIMA equivalents.

5.2. Custom Mapping

If a one-to-one mapping is not possible between UIMA and GATE annotations then it is possible to specify a Java class to do the mapping instead, as shown below. The class is specified in the feature element. Note this type of mapping is not used in the current land use example shown in the gate_mapping.xml file.

<feature class="anl.gtou.OrthMatcher"/>

The mapping class must implement anl.aida.ae.gate.FeatureBuilder and contain a constructor that takes an org.apache.uima.cas.TypeSystem as an argument (see anl.gtou.OrthMatcher as an example). The implementation of the public void build(FeatureStructure fs, FeatureMap map) method will then perform the more complicated mapping. For example, this is done using a GATE feature map value as a condition upon which to set some particular UIMA feature value.

6. Document Consumption

The consumption phase consists of adding a document's contents and its annotation tags to a Lucene index (Lucene's documentation should be read in conjunction with this document). An index descriptor file describes how these annotations will become fields in the Lucene index. By default, the following annotations from the document metadata are stored in the index.

- Location: The documents location (URL) is stored in a "path" field but not indexed.
- Author(s): Author's names, when available, are stored in an "authors" field and indexed.
- Date: The publishing date of the document is stored in a "timestamp" field and indexed.
- Title: The title of the document is stored in a "title" field and indexed.

The contents of a document are also indexed but not stored. Consequently, every document can be searched but its content is only available via its location property. The IndexDescriptor file in UIMA describes how any additional UIMA annotations, such as the keywords tagged by the GATE annotator, are to be indexed. The current Lucene indexing file used in the case study discussed is included in the server component (/Server/scenarios/data/lucene_mapping.xml). The following format from this file is shown below.

```
<fields>

    <field name="agriculture" boost="1.0" keywordSet="Agriculture">
        <feature>anl.aida.types.Agriculture:canonicalName</feature>
    </field>
```

....
</fields>

The field and feature elements are explained in Table 3.

Table 3. UIMA to GATE document consumption mapping.

Attribute	Description	Required
name	The name of the Lucene field to create	YES
boost	A boost value for the created field	NO
keywordSet	Marks this field as containing keywords. The contents of this field will become keywords in the named set	NO

Multiple field elements can be specified. The boost value (if used) relates to term scoring, which is explained more below; in short boosting a field makes the occurrence of the terms in the boosted field more important relative to terms in other fields.

The feature element describes what annotation features should be stored in the field. The UIMA feature specification format is used here: everything to left of the ":" is an annotation type and everything to the right is the feature name. In the above example, the annotation type (i.e., Java class) is "anl.aida.types.Agriculture" and the feature is "canonicalName." Using the above example, for each indexed document, the anl.aida.lucene.LuceneConsumer class will create an indexed but not stored Lucene field called "agriculture" and populate it with the value of the canonicalName feature for each "Agriculture" annotation in the current document. In addition, the contents of this field over all the documents will become the keywords in the "Agriculture" keyword set. The resulting Lucene index will thus map keywords and terms to documents and allow the relative scoring of terms in the documents.

7. Scoring

The anl.aida.ae.lucene.TermScorer class provides term scoring. It implements tf-idf scoring together with Lucene's normalization. This score value is used to rate the relative importance of a term in a document. The full formula is:

$$S_{td} = TF_{td} * IDF_t * N_d$$

where S is the tf-idf score of a term (t) in a document (d), TF is the term's frequency, IDF is the inverse document frequency, and N enables a normalization factor or boost value of d to affect S regardless of d's length. TF is evaluated by:

$$TF_{td} = \frac{n_t}{\sum_i w_i}$$

with the number of instances (n) of t divided by the sum total of all term (w) occurrences in a document. The final coefficient used for the tf-idf score, IDF, is defined as:

$$IDF_t = 1 + \log(\frac{|D|}{|\{d : t \in d\} + 1|})$$

where *D* is the total number of documents and *d* represents each document containing the term (*t*). The *IDF* value, in essence, provides a higher score for more rare terms. The tf-idf score is calculated for each term in each document. These scores can then be used in the client in the creation of semantic maps. The LuceneConsumer configuration parameters can be found in the appendix.

8. Creating the Cache

The final phase is the actual creation of the cache used by the AIDA Client. Cache creation occurs after the LuceneConsumer has consumed all the documents. The cache is then created from the resulting Lucene index. The cache consists of two parts:

1. The Lucene index itself. This allows the client to do a google-like search of the cache for relevant documents.
2. A collection of ScoredTermCollection (STC) files: STC files encompass the duration of the entire cache and additional files for the individual time slices within the cache. The STC files are further explained below.

The cache is created from the Lucene index. In the cache, an STC file that covers the cache's date range is created as well as STC files for specified time slices within the overall range. Currently, the time slices are produced by an anl.aida.util.DateRangeProducer class. A DateRangeProducer takes a start date and the total number of days to produce ranges for a range length (in days). It produces ranges that represent sliding windows within the range, specified by start + total days. For example, if the start date is 03/01/09, the total number of days is 31, and the range (or interval) is 7 days then the producer will produce the following ranges:

- 3/01/09 - 3/08/09,
- 3/02/09 - 3/09/09,
- 3/03/09 - 3/10/09,
- …,
- 3/24/09 - 3/31/09.

These ranges can then be used to create the individual STC files that make up a cache. Unfortunately, it's not possible to filter documents by date and insure that the term frequencies and document counts pertain only to those documents in the filtered set. Consequently, the process of creating a cache requires filtering documents out of the main full index and creating smaller temporary sub-indices.

The anl.aida.ae.lucene.CacheCreator is used to create the cache itself. The cache for the example scenario will be written to the scenarios directory (i.e., /Server/scenarios/data/cache) when the server is executed. The cache takes a Lucene IndexReader, DateRangeProducer, and a term filter. The IndexReader should point to the full Lucene index. The DateRangeProducer is used to produce the STC files for the time slices and the filter can be used to filter out terms when creating the STC files. The other

relevant piece used by the CacheCreator is an anl.aida.ae.lucene.ReaderProducer. This produces a Reader that is used to read a document's contents when creating the sub-indices. The CacheCreator creates a directory that includes all the STC files for the cache, a copy of the Lucene index in an “index” subdirectory, and a cache.xml file that describes the cache. Example contents from a cache.xml file are shown below.

```
<cache lucene_index="index">
    <range>
            <start>1235887200205</start>
            <length>245</length>
            <interval>7</interval>
    </range>

    <cache_items>
        <cache_item>stc_090301_091101.bin</cache_item>
        ....
    </carche items>
    <keywords>
                    <keyword_set name="Agriculture">
                        <keyword>
                                <term>vegetables</term>
                                <label>Vegetables</label>
                            </keyword>
                            ....
                     </keyword_set>
                ....
    </keywords>
</cache>
```

The index attribute is the path to the cache's Lucene index and the cache_item elements are paths to STC files. The range element specifies the cache's range and time slice interval. Note that an STC file contains its date range timestamp within its title as well. The keywords element contains the keyword sets that have been generated by the cache creator and the LuceneConsumer. These keywords are generated from the Lucene fields that have been marked as keywords in the index descriptor file. The term corresponds to the “commonName” in the feature and the “label” to the “canonicalName”.

9. Index Post-processing

The STC files are the result of doing some post processing of the Lucene index: calculating tf/idf scores and generating additional statistics. These files are time-stamped with a date range and provide the client with the data for that date range (see anl.aida.core.ScoredTermsCollection, and anl.aida.ae.lucene.STCCreator for code details). ScoredTermsCollection can be created with a filter, limiting the terms that become part of the cache. The current binary file format for an STC file is shown below.

2 long integers: the collection’s date range (start - end)

1 standard integer: the number of documents in the cache

1 standard integer : the number of terms in the cache

for each document:
1 string: the doc title
1 string: the doc URL

for each term:
1 string: the term text
1 standard integer: the number of documents the term is in
1 standard integer: the max frequency of the term
1 standard integer: the min frequency of the term
1 floating point number stored as a standard integer[1]: maximum score
1 floating point number stored as a standard integer[1]: minimum score

From here until the end of the doc, there are a series of quadruple standard integers, one for each term doc frequency and score:

1 standard integer: the term index
1 standard integer: the doc index
1 standard integer: the term frequency in that doc
1 floating point number stored as a standard integer[1]: the score

STC files can be read with anl.aida.core.STCReader and written with anl.aida.core.STCWriter.

10. Running the Cache Creator

The actual cache is created using the create_cache.sh or create_cache.bat scripts in the server component (/Server/cache_cache.sh and /Server/create_cache.bat). These scripts execute the anl.aida.CPERunner class, which in turn runs a UIMA Collection Processing Engine (CPE). This CPE is defined in /Client/config/sample_cpe.xml. When executing the scripts, all the phases and code described previously are run. The /Client/scenarios/data/server.properties file contains the configuration parameters for the CPE. This file, essentially, serves as the required inputs and file references that configure the phases described. The current file is set to execute a search on the Iowa sub-scenario, covering the dates of 03/01/09-11/01/09 with six local newspapers read and parsed. Additional comments in that file describe each configuration property. The end result of the CPE is a new cache created in the directory mentioned earlier (see /Server/cache_Iowa and /Server/cache_Nebraska for existing scenario output). Standard output and runtime errors are logged to the console and the uima.log file.

[1] The storage encoding uses the "floatToIntBits" function.

Appendix

Applied Reader Files and Indices Folder

Reader classes=anl.aida.reader.local
Newspaper index files produced=/Server/scenarios/data/

Required User Input for AIDA Server

Required input=/Server/scenarios/data/server.properties

Gate Annotator Parameters

Parameter Key	Description	Constant	Default Value
anl.aida.ae.gate.GateAnnotator.GateAppFile	the location of the .gapp file to run	GateAnnotator.GATE_APP_FILE	/Server/gate_app/application.xgapp
anl.aida.ae.gate.GateAnnotator.MappingFile	the file that describes the mapping GATE and UIMA annotations	GateAnnotator.MAPPING_FILE	/Server/scenarios/data/gate_mapping.xml
anl.aida.ae.gate.GateAnnotator.GateHome	GATE's home directory	GateAnnotator.GATE_HOME	/Server//gate_app
anl.aida.ae.gate.GateAnnotator.GateLogProps	gate's log4j properties file	GateAnnotator.GATE_LOG_PROPS	/Server/gate_app/log4j.properties

Lucene Consumer Parameters (see anl.aida.ae.lucene.LuceneConsumer)

Parameter Key	Description	Constant*	Default Value
anl.aida.ae.lucene.IndexDescriptor	xml file which specifies which annotation types / features to add to the index as Lucene document fields	MAPPING_FILE	<N/A>
anl.aida.ae.lucene.IndexDirectory	the path the lucene index directory	INDEX_DIR	./indices/uima
anl.aida.ae.lucene.ClearIndex	a boolean indicating whether or not clear the index during component intialization. Useful for debugging when we want to regenerate index each time during testing	CLEAR_INDEX	true
anl.aida.ae.lucene.CachedContentDir	directory where temporary content can be put when creating the cache	CACHED_CONTENT_DIR	/Server/cached_content
anl.aida.ae.lucene.OutputDir	directory where the cache will be created	CACHE_OUTPUT_DIR	./cache
anl.aida.ae.lucene.CacheStartDateDate	starting date of the cache (yyyymmdd)	CACHE_START_DATE	20090301
anl.aida.ae.lucene.CreateCache	a boolean indicating whether or not the cache should be created	CREATE_CACHE	true

* The Constant value refers to constants defined in the class LuceneConsumer that can be used to refer to the parameter keys programmatically.