HIVE Core Technical Report

Jose R. Perez-Aguera, Lina Huang, Hollie White,
Jane Greenberg and Ryan Scherle

July 25, 2010

Chapter 1

HIVE Core

1.1 What is HIVE

HIVE try to solve traditional problems of vocabularies in digital environments. We can enumerate these problems as follow:

- Concept Retrieval
- Automatic annotation
- RIA technology for vocabulary server Web User Interface
- Linked Data for Concepts based on SPARQL end points

In the next sections we will explain how HIVE solves these problems, to offer a methodology and tool which allows vocabularies use on the Web.

HIVE Core is a SKOS Core based library written entirely in Java. It is a technology suitable for nearly any application that requires vocabularies management.

1.2 HIVE core architecture

HIVE Core allow users to access to the vocabularies in different ways, As you can see in the image below, the system can be queried in two different ways:

Natural Language Interface SPARQL interface

HIVE integrate different libraries to allow these access. Natural Language interface allow us to use keyword based queries to retrieve concepts from the vocabularies. This interface is based on Lucene, so each query is fired against a Lucene index where the concepts are stored.

1.3 Loading vocabularies from SKOS Core

Loading new vocabularies in HIVE is quite simple. HIVE is able to load any vocabulary in SKOS Core format from RDF files. So If we have a vocabulary in a file with SKOS Core format, you could import it into HIVE very easily.

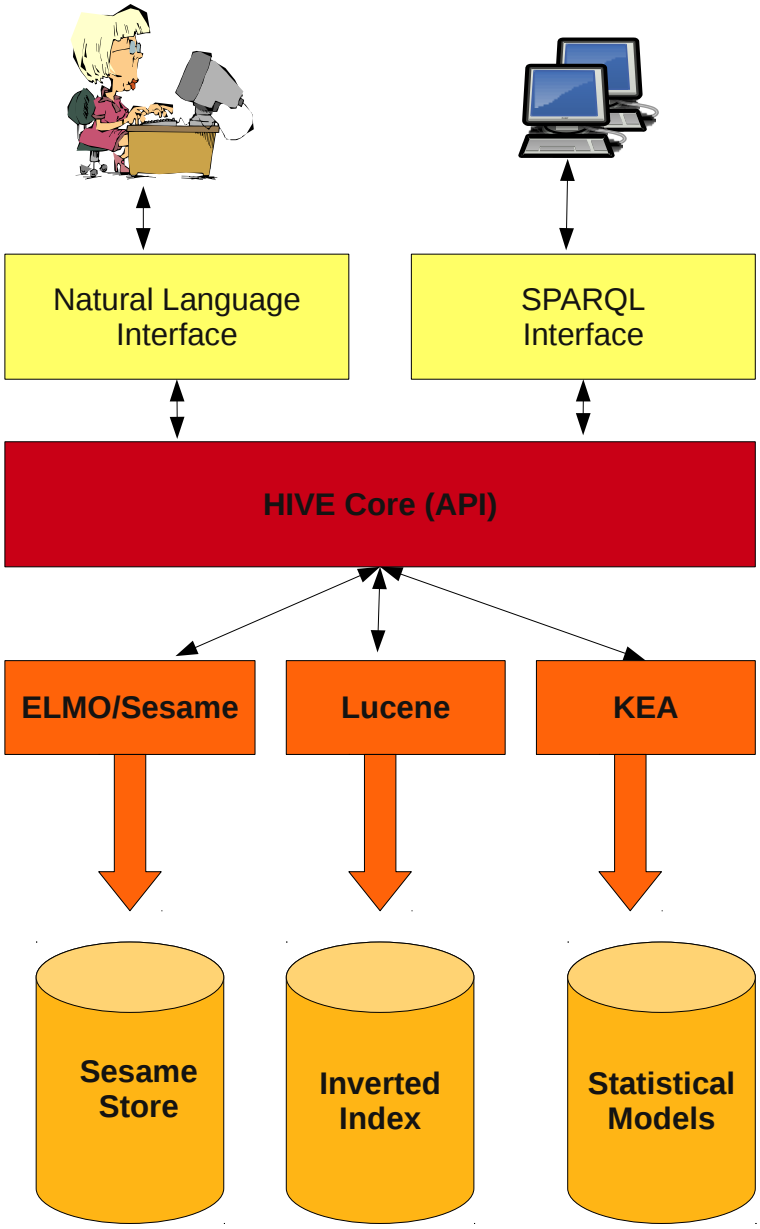

Figure 1.1: HIVE Core architecture

To import a vocabulary you only need execute AdminVocabularies class, which have a main method that implement the basic importation. The parameters to execute this class are two:

1- the path to the ../conf directory for HIVE, where the configuration file for the HIVE server and for every vocabulary are. In each vocabulary configuration file you have to define where do you want store the different databases that HIVE will use to store the vocabulary and you can also define the location of the RDF file from where you want import the vocabulary.

The format of the file for the example of lcsh.properties file would be the following:

```
 #Vocabularies data
name = LCSH
longName = Library of Congress Subject Headings
uri = http://id.loc.gov/authorities

#Sesame Store
store = /home/hive/hive-data/lcsh/lcshStore

#Lucene Inverted Index
index = /home/hive/hive-data/lcsh/lcshIndex

#Alphabetical Index
alpha_file = /home/hive/hive-data/lcsh/lcshAlphaIndex

#Top Concept Index
top_concept_file = /home/hive/hive-data/lcsh/lcshTopConceptIndex

#Dummy tagger data files
lingpipe_model = /home/hive/hive-data/lingpipe/postagger/models/medtagModel

#KEA data files
stopwords = /home/hive/hive-data/lcsh/lcshKEA/data/stopwords/stopwords_en.txt
kea_training_set = /home/hive/hive-data/lcsh/lcshKEA/train
kea_test_set = /home/hive/hive-data/lcsh/lcshKEA/test
rdf_file = /home/hive/hive-data/lcsh/lcsh.rdf
kea_model = /home/hive/hive-data/lcsh/lcshKEA/lcsh
```

You should have one file like this for every vocabulary that you want load in HIVE.

```
  String configpath = args[0];
String vocabularyName = args[1].toLowerCase();

SKOSScheme schema = new SKOSSchemeImpl(configpath, vocabularyName, true);

ImporterFactory.selectImporter(ImporterFactory.SKOSIMPORTER);
Importer importer = ImporterFactory.getImporter(schema);
importer.importThesaurustoDB();
importer.importThesaurustoInvertedIndex();
```

```
importer.close();
System.out.println("Import finished");
// if (args[2].equals("train")) {
// TaggerTrainer trainer = new TaggerTrainer(schema);
// trainer.trainAutomaticIndexingModule();
//  }
}
```

1.3.1 Creating databases and indices

The AdminVocabularies class use an SKOSScheme that represent a SKOS vocabulary into HIVE and a Importer which implement all the importation process. Importer are implemented as a Factory, although at this moment there is only implemented one importer for SKOS format. Once we have decide what importer we want to use we can call different methods. Every method is related with a different kind of storage:

importThesaurustoDB(): this method implement the interface to add data to the Sesame database. HIVE use NativeStore, so every vocabulary imported using this method will be stored on your file system with this format. importThesaurustoInvertedIndex(): this method create a Lucene index to store the information about concepts. At this point we are following a document oriented approach to represent concepts in the inverted index, so each concept is actually represented as a document with different fields. Each field represent the different elements which we can find in a vocabulary, preferred term, broader terms, scope notes, etc.

HIVE importer create to additional indexes for every vocabulary, in order to optimize the access to different representations of the same vocabulary. These two additional indexes are the following:

Alphabetical index: This index is an alphabetically ordered list which make easier represent concepts in a alphabetical way Hierarchical index: This index allow us to create hierarchical representation of the data in order to implement representations based on the hierarchical structure of the vocabularies.

Both indexes are created after the inverted index.

All theses indexes and databases can be stored wherever you need in your file systems. The location of each database and index have to be defines in the properties file what we have for every vocabulary in the ../conf directory.

1.3.2 Creating training data for automatic metadata generation

HIVE automatic metadata generation system is based in KEA. When a vocabulary is imported we have the chance to create a model to use the vocabulary for automatic metadata generation. In this section we will explain how create this model and when locate the information about this model. The theoretical framework of KEA and how the model is used for automatic metadata generation will be explained in chapter 3.2.2.

KEA algorithm is based on machine learning techniques and more specifically in a Naïve Bayes based classification system. Machine Learning algorithms use to need two different sets of data: Training data and Test data.

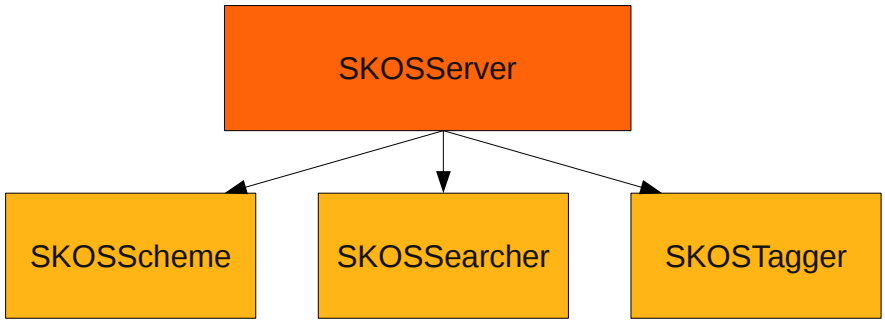

Figure 1.2: SKOS Server

Training data are used to build an statistical model which is used by the algorithm to recognize positive and negative examples. This statistical models use to be based on real world example, so for automatic metadata generation we will use a corpus of documents where keywords and metadata has been assigned by hand.

1.4 SKOS Server

SKOSServer is the high level classes that we have to instantiate to get up the Vocabulary server:

```
SKOSServer server =
 new SKOSServerImpl("/home/hive/workspace/hive-core/conf/vocabularies");
```

where the method's argument makes reference to the configuration file where the names of the thesauri to load is written.

SKOSServer manages the three basic classes which take the work to implement the HIVE basic functionalities: Concept Search (SKOSSearcher), Vocabularies management (SKOSScheme) and indexing SKOSTagger.

Any of theses classes are actually interfaces which are implemented using different libraries,

1.5 SKOS Scheme

Every vocabulary in modeled in HIVE using SKOSScheme class. This class contains information about the vocabularies and some methods to manage them.

We can get information about statistics for every vocabulary, number of terms, and relations, updates, etc. The following piece of code show you how to do that:

```
TreeMap<String, SKOSScheme> vocabularies = server.getSKOSSchemas();
Set<String> keys = vocabularies.keySet();
Iterator<String> it = keys.iterator();
  while (it.hasNext()) {
  SKOSScheme voc = vocabularies.get(it.next());
  System.out.println("NAME: " + voc.getName());
  System.out.println("\t LONG NAME: " + voc.getLongName());
  System.out.println("\t NUMBER OF CONCEPTS: "
      + voc.getNumberOfConcepts());
  System.out.println("\t NUMBER OF RELATIONS: "
      + voc.getNumberOfRelations());
  System.out.println("\t DATE: " + voc.getLastDate());
  System.out.println();
  System.out.println("\t SIZE: " + voc.getSubAlphaIndex("a").size());
  System.out.println();
  System.out.println("\t TOP CONCEPTS: "
      + voc.getTopConceptIndex().size());
  }
```

1.6 Search functionalities with SKOSSearcher

SKOSSearcher, is the interface which defines the search functionalities in HIVE. Theses functionalities can be divided into two different kind of search:

- Formal Search
- Keyword based search

1.6.1 Formal Search based on SPARQL

Formal search is based on SPARQL, a formal language to get information from RDF databases. SPARQL is the standard query language for Semantic Web application and it is useful to implemented Web Services based on SPARQL end points, and so allowing third part application retrieve information from a RDF database.

```
 searcher.SPARQLSelect(
  "SELECT ?s ?p ?p WHERE {?s ?p ?o} LIMIT 10",
  "nbii");

 searcher.SPARQLSelect(
  "PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
  SELECT ?s ?p ?o WHERE {  ?s ?p ?o . ?s skos:prefLabel \"Damage\" .}",
  "nbii");

 searcher.SPARQLSelect(
  "PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
```

```
  SELECT ?uri ?label WHERE { <http://thesaurus.nbii.gov/nbii#Mud>
  skos:broader ?uri . ?uri skos:prefLabel ?label}",
  "nbii");
```

1.6.2 Keyword based search

Keyword based search is based in natural language search, so the user of the system can write queries without any constraint. HIVE keyword based search is based our research on Concept and Entity retrieval which has been described in José R. Pérez-Agüera, Javier Arroyo, Jane Greenberg, Joaquin Perez-Iglesias and Victor Fresno. Using BM25F for Semantic Search. Semantic Search Workshop at the 19th Int. World Wide Web Conference WWW2010 April 26, 2010 (Workshop Day), Raleigh, NC, USA

```
System.out.println("Search by keyword:");
List<SKOSConcept> ranking = searcher.searchConceptByKeyword("accidents");
System.out.println("Results in SKOSServer: " + ranking.size());
String uri = "";
String lp = "";
  for (SKOSConcept c : ranking) {
    uri = c.getQName().getNamespaceURI();
    lp = c.getQName().getLocalPart();
    QName qname = new QName(uri, lp);
    String origin = server.getOrigin(qname);
    if (origin.toLowerCase().equals("nbii")) {
      System.out.println("PrefLabel: " + c.getPrefLabel());
      System.out.println("\t URI: " + uri + " Local part: " + lp);
      System.out.println("\t Origin: " + server.getOrigin(qname));
    }
  }
```

1.6.3 URI based search

```
System.out.println("Search by URI:");
SKOSConcept c2 = searcher.searchConceptByURI(
  "http://thesaurus.nbii.gov/nbii#", "Enzymatic-activity");
Concept c2 = searcher.searchConceptByURI(uri, lp);
List<String> alt = c2.getAltLabels();
System.out.println("PrefLabel: " + c2.getPrefLabel());
for (String a : alt) {
  System.out.println("\t altLabel: " + a);
}
System.out.println("\t Origin: " + server.getOrigin(c2));
System.out.println("\t SKOS Format: \n" + c2.getSKOSFormat());

/**
/ * Get Children by URI test
 */
SKOSConcept con = searcher.searchConceptByURI(
    "http://id.loc.gov/authorities/sh2001009743#", "concept");
```

```
TreeMap<String,QName> children = searcher.searchChildrenByURI(
  "http://id.loc.gov/authorities/sh2001009743#", "concept");
for (String c : children.keySet()) {
  System.out.println("prefLabel: " + c);
}
```

1.7 Tagging documents with SKOS Tagger

One of the most important features of HIVE is the module for automatic metadata extraction. Use this functionality is quite easy using the SKOSTagger class. This class is accesed via SKOSServer using the method getSKOSTagger().

Once we have an SKOSTagger we can use the method getTags(), to get the keywords for a given document. The arguments required for this method are the source of the text, usually a document, the list of vocabularies to normalize the keywords and a SKOSSearcher object. Following there is an example about how to use HIVE tagger.

```
SKOSTagger tagger = server.getSKOSTagger();

String source = "/home/hive/Desktop/ag086e00.pdf";
source = "http://en.wikipedia.org/wiki/Biology";

List<String> vocabs = new ArrayList<String>();
vocabs.add("nbii");
vocabs.add("lcsh");
vocabs.add("agrovoc");

List<SKOSConcept> l = tagger.getTags(source, vocabs,
  server.getSKOSSearcher());
System.out.println();
System.out.println("Tagging Results for ALL");
for (SKOSConcept s : l) {
  System.out.println(s.getPrefLabel());
  System.out.println(s.getQName().getNamespaceURI());
}
```

Chapter 2

HIVE Web

HIVE web is a demo that implement a web interface for HIVE Core.

2.0.1 Web UI for Browsing and Searching vocabularies

We design and implement HIVE Web User Interface based on Rich Internet Application(RIA) model. In traditional web application, the user interface is only assigned for two tasks: display information and send the request to the designated web server, while all the data storage and application logic are maintained by the program in web server. The hindrance of traditional web application model created for user experience has two folds: (1)Each request sent by user interface requires a page load hence reduce the responsiveness of the user interface. (2) The web page is a static document to present information rather than interactive application for user to control and manipulate. The computing power of personal computer is sufficient to process and render complex data logic in client browser. The RIA model distributes a part of computation to client side user interface, so that user can perform complex interaction with the interface without requesting the server. Furthermore, if user's interaction must request server to refresh some data, the client can selectively retrieve from the server only the information that needed to be changed, update its internal status, and redisplay the modified content. Typical examples of RIA in the market nowadays are GMail, Google Document, etc.

We design and implement HIVE Web User Interface based on Rich Internet Application(RIA) model. In traditional web application, the user interface is only assigned for two tasks: display information and send the request to the designated web server, while all the data storage and application logic are maintained by the program in web server. The hindrance of traditional web application model for user experience has two folds: (1)Each request sent by user interface requires a page load hence reduce the responsiveness of the user interface. (2) The web page is a static document to present information rather than interactive application for user to control and manipulate. The computing power of personal computer is sufficient to process and render complex data logic in client browser. The RIA model distributes a part of computation to client side of user interface, so that user can perform complex interaction with the interface without requesting data from the server. Furthermore, if user's interaction must request server to refresh some data, the client can selectively

retrieve from the server only the information that needed to be changed, update its internal status, and redisplay the modified content. Typical examples of RIA application in the market nowadays are GMail, Google Document, etc.

The technologies for implementing RIA rapidly emerged during recent years. AJAX, Silverlight, Flex are all the popular technologies to implement RIA user interface. One thread of the RIA technologies are based on Flash, which usually requires users to install Adobe Flash in their browser. The other thread is powered by AJAX, standing for Asynchronous JavaScript and XML. Programming in AJAX has two major downsides: (1) Programming large trunk of JavaScript is tedious and error-prone, and is difficult to maintain and reuse. (2) JavaScript suffers from browser incompatibilities. A lot of AJAX framework is dedicated designed to address the programming issues of AJAX. We abandoned Flash thread technologies to implement RIA because we want HIVE can be accessible via standard browser without additional installation of new technology. We chose Google Web Tool Kit as final RIA solution for HIVE because it eases the burden by allowing developers to build and maintain large and high performance JavaScript front-end applications in the Java programming language, while addressing the browser incompatibilities at the same time.

The technologies for implementing RIA rapidly emerged during recent years. AJAX, Silverlight, Flex are all the popular technologies to implement RIA user interface. One thread of the RIA technologies are based on Flash, which usually requires users to install Adobe Flash in their browser. The other thread is powered by AJAX, standing for Asynchronous JavaScript and XML. Programming in AJAX has two major downsides: (1) Programming large trunk of JavaScript is tedious and error-prone, and is difficult to maintan and reuse. (2) JavaScript suffers from browser incompatibilities. A lot of AJAX frameworks are dedicatedly developed to address the programming issues of AJAX. We abandoned Flash thread technologies to implement RIA because we want HIVE can be accessible via standard browser without additional installation of new technology. We chose Google Web Tool Kit as final RIA solution for HIVE because it eases the burden by allowing developers to build and maintain large and high performance JavaScript front-end applications in the Java programming language, while addressing the browser incompatibilities at the same time.

2.1 The HIVE Web UI

The HIVE Web UI are partitioned into three components: HIVE Home, HIVE Concept Browser and HIVE Indexing. HIVE Home is the home page for HIVE Web UI, which introduces the concept of HIVE and shows the vocabulary statistics to help user has a overall understanding of what HIVE is and what HIVE can help them accomplish.

The HIVE indexing includes the key functionality, called HIVE indexer, which give a UI to HIVE automatic annotation tool described in past sections. HIVE indexer automatically extract concepts from document based on user's selection of authoritative vocabulary sources. The concepts extracted are shown in term cloud, with color code to show the source of vocabulary and the font size to show the relevance of extracted concepts. Figure 2.1 shows HIVE Web UI for automatic annotation tool after a document is processed and concepts are extracted by HIVE indexer.

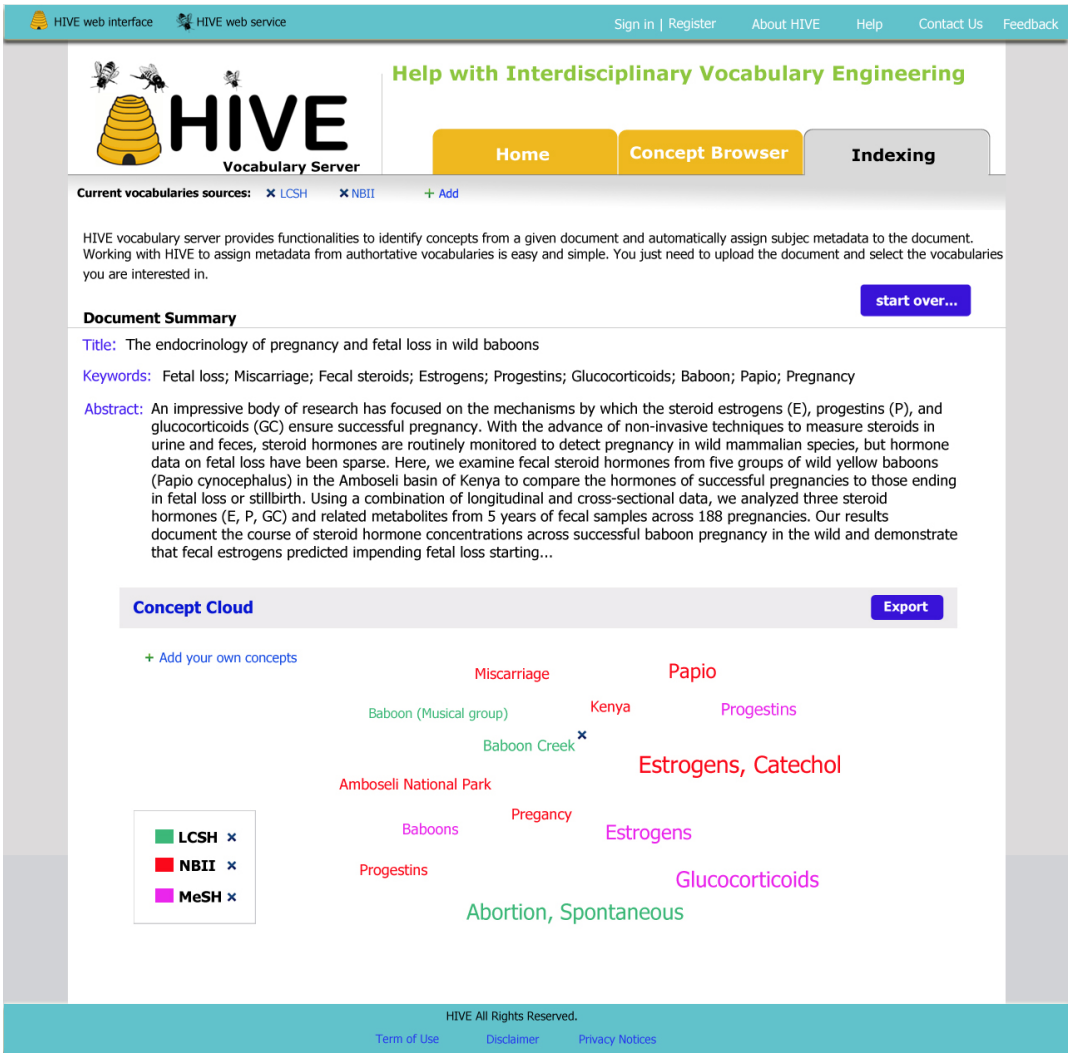

Figure 2.1: Automatic Annotation tool

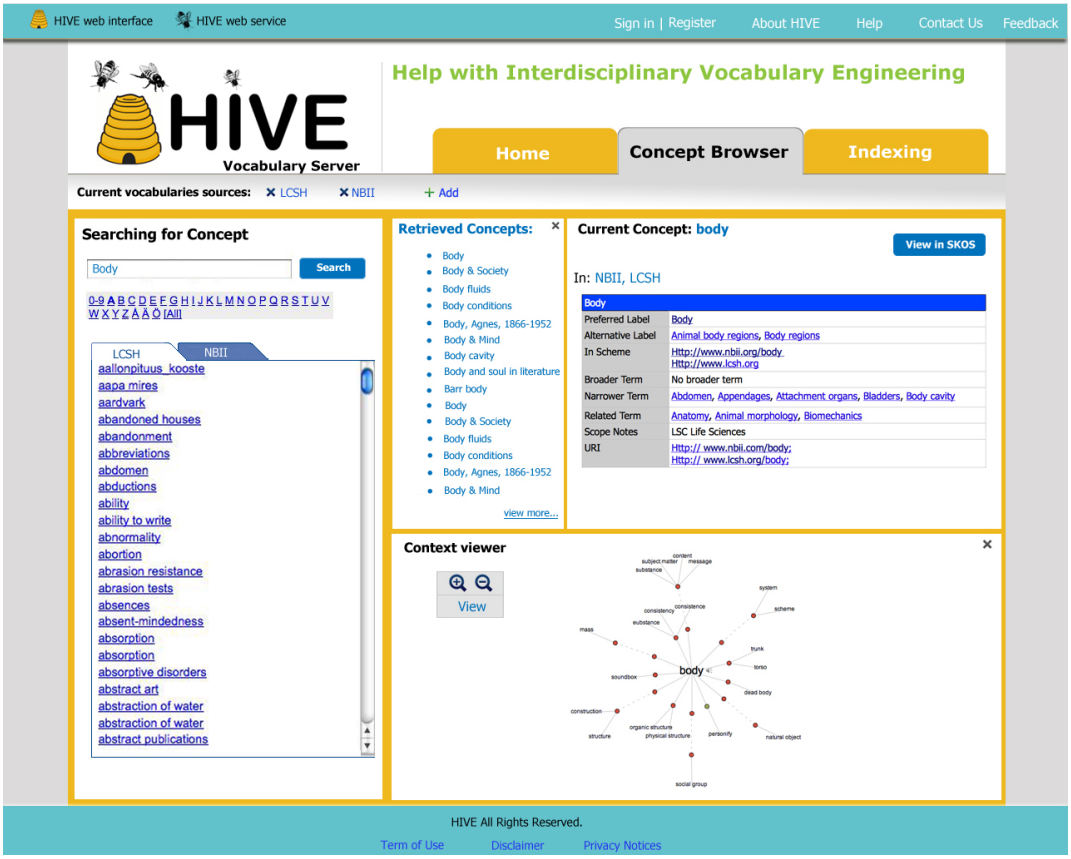

Figure 2.2: Concept Browser

HIVE Concept Browser, showed in figure 2.2 is the most important component of HIVE Web UI. The HIVE Concept Browser also articulates the RIA paradigm applied in the design and implementation process.

The goal of HIVE Concept Browser is to allow users to browse and search concepts from heterogeneous vocabulary authorities such as Library of Congress Subject Headings and National Biological Information Infrastructure. The other important aspect of Concept Browser is to visualize the context of concept via showing the relations to help user better understand difficult concept as well as its usage. The functionalities of Concept Browser include:

- Allow user to traverse from different vocabularies.
- Allow user to open and close a vocabulary.
- Allow user to search concepts in multiple vocabularies, and filtering the concepts based on user selection.
- Visualize the hierarchical structure of user selected vocabulary for browsing.
- Visualize the context of concepts to help user better understand difficult concept.

- Display the concept in SKOS metadata schema.

The design principles of the functionalities described above are based on RIA, and the implementation are powered by RIA technology, Google Web Tool Kit. Future work on HIVE Web UI will focus on the usability testing for the RIA feature to gain a constructive understanding of impact of RIA technologies on the usability of interactive web application.

2.1.1 Linked data for concepts

HIVE is a multi-vocabulary environment that allows access to concepts and vocabularies using a simple URL. This idea was presented for LCSH, and we have generalized this approach for any vocabulary stored on a server. HIVE implements the technological infrastructure to store millions of concepts from different vocabularies and make them available on the Web by a simple HTTP call based on RESTful paradigm. Vocabularies can be imported in HIVE using SKOS/RDF format. As a result of this work, sharing concepts and vocabularies on the Web becomes easy and straightforward.

2.2 Installing HIVE web

The installation of the demo HIVE web is quite simple. as we have seen previously, HIVE web is based on GWT, so the installation is similar to other Java based applications.

HIVE web can be installed in application servers like tomcat or Resin. We have using Tomcat 6.0 for this document.

The main constraint is that HIVE web must be installed in the ROOt directory of your application server, because there is a bug in the GWT multipage module that has not been solved yet by the developers of this module (for the discussion about this you can read the thread in the GWT multipage web page http://claudiushauptmann.com/gwt-multipage/)

To run HIVE web in your application server you should follow the following steps:

- Copy the content of the directory hiveweb in your HIVE distribution in the ROOT directory of your Tomcat installation.
- Go to the ../WEB-INF/conf directory to configurate the configuration files.

There are to kind of configuration files in HIVE:

- vocabularies: This is the general file for configuration. This file contains the information about the vocabularies that will be loaded when the application starts. So if we want load three vocabularies we will include its names in this file. You only can have one “vocabularies” file for each HIVE instance.
- $< vocabularyName >$.properties: These properties files include information about each vocabulary that can be loaded by the system. For example the file lcsh.properties will include information to load LCSH vocabulary in

HIVE. This file not only have information about the vocabulary, but also have the paths to the different databases where the vocabulary is stored. As we said previously, HIVE have a double storage system, one based on Sesame RDF native databases and one based on Lucene. Lcsh.prperties will include the path in the file system to this databases. The format for these files in described in detail in 1.3.

2.2.1 Memory problems

If we need load a big amount of data in HIVE we will have to allocate more memory before start our Tomcat server. To do that we will need to define a environment variable like this:

```
export CATALINA_OPTS="-Xmx1512m -XX:-UseGCOverheadLimit"
```

with this variable we are allocating more memory for tomcat and we will avoid problems to load huge vocabularies on HIVE web.

Chapter 3

Automatic Metadata Generation with KEA

3.1 Introduction

The algorithm that we are using in HIVE is KEA++, KEA++ has been developed for Ian Witten's group from University of Waikakato. Last version of KEA has been part of the Olena Medelyan phd. The other interesting result of her thesis has been Maui[1], that is based on KEA.

3.2 How it works?

KEA use a Machine Learning approach for automatic keyphrase extraction:
Machine Learning approaches use to have two different phases:

- Training phase, where examples of solutions are used to explain to the system how the problem can be resolved. In our case these solutions are documents indexed by human indexers using controlled vocabularies. From now this set of documents will be called training set.
- Test phase, where the system try to solve new problems which are similar to the solved problems used to train the system. In our case, these problems to solve will be the documents that we want to index using the vocabularies that we have in HIVE.

KEA use a Machine Learning scheme that can be divided in the following parts:

- Candidate identification using a set of features
  - Candidate extraction
  - Features identification
- Filtering

[1] http://code.google.com/p/maui-indexer/

– Training the model using the training set
– Extracting keyphrases from new documents that are not present in the training set

3.2.1 Candidate identification

Candidate identification is the process where the most representative keyphrases are identified for each document in the training set.

Candidate extraction

Candidates are extracted from documents using some simple methods like n-gram identification, stopwords filtering and stemming. KEA calls this candidates pseudo-phrases. This pseudo phrases are normalized using the vocabularies. For example, phrases such as "algorithm efficiency", "the algorithms' efficiency", "an efficient algorithm" and even "these algorithms are very efficient" are matched to the same pseudo phrase "algorithm effici", where "algorithm" and "effici" are the stemmed versions for the corresponding full forms.

The result is a set of candidate index terms for a document, and their occurrence counts. As an optional extension, the set is enriched with all terms that are related to the candidate terms, even though they may not correspond to pseudo-phrases that appear in the document. For each candidate its one-path related terms, i.e. its hierarchical neighbors (BT and NT in Agrovoc), and associatively related terms (RT), are included. If a term is related to an existing candidate, its occurrence count is increased by that candidate's count. For example, suppose a term appears in the document 10 times and is one-path related to 6 terms that appear once each in the document and to 5 that do not appear at all. Then its final frequency is 16, the frequency of the other terms that occur is 11 (since the relations are bidirectional), and the frequency of each non-occurring term is 10. This technique helps to cover the entire semantic scope of the document, and boosts the frequency of the original candidate phrases based on their relations with other candidates.

In both cases—with and without related terms—the resulting candidate descriptors are all grammatical terms that relate to the document's content, and each has an occurrence count. The next step is to identify a subset containing the most important of these candidates.

Features identification

The features which KEA use are the followings:

- TF*IDF
- First occurrence position of the keyphrase in the document normalized by document length
- Length of the keyphrases (in words, which boots multiterms)
- Node degree reflects how richly the term is connected in the thesaurus graph structure. The "degree" of a thesaurus term is the number of semantic links that connect it to other terms—for example, a term with one

broader term and four related terms has degree 5. KEA considers three different variants:

  - the number of links that connect the term to other thesaurus terms
  - the number of links that connect the term to other candidate phrases
  - the ratio of the two.

- Appearance is a binary attribute that reflects whether the pseudo-phrase corresponding to a term actually appears in the document.

3.2.2 Filtering

Training the model using the training set

KEA use a classifier to create a model for good and bad keyphrases. In Machine Learning clustering and classification are very common techniques. In this case we have a classifier. Classifiers can be understood using the same sense that we use when we classify books. In these example we hay to classify keyphrases in only two classes: GOOD Keyphrases (these keyphrases that were used by human indexers in our training set) and BAD keyphrases (these keyphrases that, although appears in the documents that we are using to train the system, has not been used by our human indexer to index the documents). These kind of binaries classifiers are very common in automatic classification and there exits a lot of methods to classify elements. KEA use a method named Naive Bayes, which is very simple and effective. When we finish to classify the keyphrases that appears in our training set, we have a model that we can use to estimate the probability that a keyphrase have to be a good or a bad keyphrase for a new document, based on the model that we are generated using our training set.

Extracting keyphrases from new documents that are not present in the training set

Once the model has been generated, we can offer our system to the users to index new documents. To index a new document, we only need to know the probability that have the keyphrase that occur in this new document to be a GOOD or a BAD keyphrase. The computation of this probability is quite simple. We need to compute both probabilities based on the information which we have in our model and in the new document. So, suppose just the two features $TF \times IDF$ and position of first occurrence are being used. When the Naïve Bayes model is used on a candidate pseudo phrase with feature values t and f respectively, two quantities are computed:

$P[yes] = (Y/Y + N)P_t fidf[t|yes]P_d istance[f|yes]$

and

$P[no] = (N/N + Y)P_t fidf[t|no]P_d istance[f|no]$

where Y is the number of positive instances in the training files—that is, author-identified keyphrases—and N is the number of negative instances—that is, candidate phrases that are not keyphrases.

The overall probability that the candidate phrase is a keyphrase can then be calculated:

$P_k = P[yes]/(P[yes] + P[no])$

Candidate phrases are ranked according to this value.