SUPPORTING KEYWORD SEARCH ON SEMANTIC WEB DOCUMENTS

by

RAVI PAVAGADA

(Under the Direction of Dr. Amit P. Sheth)

ABSTRACT

Most contemporary search engines [8, 17, 41] allow searches on keywords and support direct matching of the keywords with document contents. These search engine return Web pages that contain the search terms by performing the direct or pattern matching of search terms with the page contents. Additionally, the matched search terms might appear in any paragraph of the returned page. Hence, most of these searches return large set of matched Web pages that may or may not be relevant to the context of search. Thus, more often than not the users have to sift through the retrieved pages to find the information they are looking for. In this thesis, we address this problem of search by returning meaningful results that are relevant to the search. We present a prototype search and retrieval system for retrieving information from RDF which represents the knowledge contained in the Web documents. We have addressed the problem of search by returning meaningful results that are relevant to the query. Our proposed system uses the concept of keyword search by extending the concept of keyword search, to ontological classes, literals and relationship. The system processes the entered search terms by matching them to the ontological concepts and relationships. The results returned by our system are either a set of triples or a sub-graph relevant to the query. Our system currently doesn't allow searches on documents, but can be extended to support searches on annotated documents. The key feature of

our system is that it exploits relationships in RDF and returns a sub-graph relevant to the query and allows users to enter keywords that are related to the ontological concepts and relationships. We adopt an integrated approach that uses the existing knowledge in the ontology and WordNet [38] along with lexical processing to find related words, unlike other systems that either use WordNet [37] or a domain specific ontology [3, 9, 31] to find related words. Additionally, our system accepts multiple search terms per search, unlike other systems [9, 12, 14, 24] that allows a single search term or literal per search.

We compared the precision values of a keyword based retrieval system [8] with that of our system. The comparison indicated that the results returned by our system were very accurate and relevant to the query. On the other hand, the other retrieval system returned many Web pages which weren't relevant to the search.

INDEX WORDS: Keyword, Semantic Web, RDF (Resource Description Framework), XML, ontology, Semantic Information Retrieval, Relationship based retrieval, Semantic Document Retrieval

SUPPORTING KEYWORD SEARCH ON SEMANTIC WEB DOCUMENTS

by

RAVI PAVAGADA

Bachelor of Engineering (BE), Kuvempu University, India, 2001

A Dissertation Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2006

SUPPORTING KEYWORD SEARCH ON SEMANTIC WEB DOCUMENTS

by

RAVI PAVAGADA

Major Professor:	Amit P. Sheth
Committee:	John A. Miller Prashant Doshi

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August, 2006

DEDICATION

I would like to dedicate this thesis to my parents Raghavendra Rao and Sandhya, sister Roopa and my brother Ajay.

ACKNOWLEDGEMENTS

I would like to thank my major advisor Dr. Amit P. Sheth for all the support and guidance he extended to me throughout this work. I also would like to thank my co-worker Kemafor Anayanwu for all her suggestions and help. I would like to thank Dr. John A. Miller and Dr. Prashant Doshi for being part of my committee and for their support. Special thanks, to Angela Maduko for all her help and support. I would like to thank Dr. Willam S. York for his continuous guidance and support in the Glycomic's project. I thoroughly appreciate my interactions with the other members of the LSDIS Lab including Tian Hao, Christopher Thomas, Mathew Perry, Kunal Verma, Matthew Evanson, Meena Nagarajan, Cartic Ramakrishnan, Farshad Hakimpour, Cory Henson and Aleman Meza. I would like to thank my maternal uncles for their continuous encouragement and love. Finally, I sincerely thank my parents for their support and the sacrifices they made to get me into graduate school. Words are inadequate to express my gratitude to them for giving me the strength and courage needed to fulfill my goals.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

Chapter 1 – Introduction

The techniques used by present-day keyword search systems for the Web such as Google [8], Yahoo [41], MSN [17] rely primarily on document content for determining relevance. For such systems, only documents which contain the search terms are considered relevant and heuristics are employed to determine the relevance for each document. Typically, a very large collection of documents are returned for a given query and users are left with the time-consuming task of extracting specific information items of interest from such a collection. Further, given that the information items that a user may be interested in may be located in different documents, it is often necessary for users to iteratively refine their search terms in order to obtain different sets of documents which may hold additional or related pieces of information that they may need.

The limitation of Web search systems is rooted in the fact that the Web is simply a network of documents with little discernible meaning associated with the contents of the documents and the links connecting them. The vision of the Semantic Web [27] proposed by Tim Berners-Lee [6] is to improve upon this by evolving the Web to a Web of "named" objects linked by "named" relationships with meaning ascribed to its objects and relationships. Some of the advancement towards realizing this goal is development of standards for expressing and exchanging metadata such as which model relationship as first class objects. Another important building block is the bootstrapping of the Semantic Web from the current Web using automatic metadata extraction. For example, the Semantic Enhancement Engine [5] discussed the ability to support ontology-driven metadata extraction of a very large number of textual documents, and SemTag [33] demonstrated Web scale (albeit shallower) semantic tagging of approximately 264 million Web pages and could generate 434 million automatically disambiguated semantic tags for annotations. The Semantic Web environment provides a good foundation for building advanced semantic

search systems that can overcome the limitations of present-day approaches. Thus, metadata can be extracted from a set of documents and be used to improve on the problems that exist in the contemporary search engines.

In this thesis, we present a system that uses semantic search to improve the accuracy of the search. The system searches on the meta-data which has been extracted from a set of documents and returns a set of triples or a sub-graph relevant to the query. Our system currently doesn't allow searches on documents, but can be extended to support searches on annotated documents. We present a prototype search and retrieval system that allows keyword searches on related words of ontological classes, relationships. To our knowledge, this is the first system that allows multiple search terms per search, unlike other systems [9, 12, 14, 24] that allows single search term or literal. This is also the first system that uses a unique keyword expansion technique that uses the knowledge in WordNet along with a conversion algorithm and ontology to expand search terms. Ours is also the first system that allows searches on related words of ontological classes or relationships. Additionally, our system displays a sub-graph that is relevant to the query, unlike other systems [9, 12, 14, 24] that display either the matched triples or the matched literal, property, or classes.

The vision of developing semantics-driven search systems for the Semantic Web is based on a shared understanding or common agreement on the information captured in the ontology. Hence, any storage systems must provide support for retrieving the captured information. While many of these systems [16, 30] are based on formal querying languages [7, 22, 32], a few [9, 14, 24] allow the querying of Semantic Web repositories using keyword queries. [24] uses a spread activation algorithm to find related instances for a given set of concepts using a initial set of relationship weights. Swoogle [14] is a search and retrieval system for finding ontologies on the

Web and allows pattern based searches of classes, and properties. We believe that the latter approach or keyword search is very powerful because of its ease of use and wide spread familiarity to everyday users.

However, these approaches still have room for improvement. First, it will be helpful for such systems to depart from the document-centric view where logical information units are documents and adopt finer grained and contextually meaningful information units. This may help alleviate the problem of users having to rummage through several documents in order to piece together the pieces of information relevant to them. For example, a user supplying the keywords "*Tim*" "*Berners-Lee*" "*Semantic*" "*Web*" is more likely to be interested in a seeing a summary of the relationships (paths) between both entities including talks, papers, etc, than to see a set of Semantic Web documents containing assertions about such relationships. This is particularly important because, if a relationship connecting the two entities forms a path of more than one hop, then users may have to chase down these links by looking at several documents. A more natural representation of such a result is a labeled graph showing all the relationships found. This is similar to the approaches proposed for supporting keyword searches over relational [2, 11] and XML [10] databases which return sub-graphs of the data graph as query results. However, these approaches often limit their search to the set of literal values i.e. leaves or terminal nodes, e.g. the title of a book or an author's name. However, we observe that sometimes a keyword has the role of qualifying other keywords. For example, consider the query *Amit-Sheth writings*. Please refer to Figure 1 which shows the RDF [20] graph of a portal ontology for the above example. The graph shown in the figure has some of the most important classes, instances and relationships that exist in the schema and data of the ontology. It contains most up to date information of real world entities and has been populated with the most recent up to date

information of entities such as professors, publications, research labs, etc. It has 17 unique properties, around 1558 literals, 112 schema properties (properties connecting the classes) and 1024 instance statements (instance triples). Given this query, the user will be interested in seeing a sub-graph that connects the instance *Amit Sheth* to all of his papers, articles, book chapters and presentations. Therefore, all nodes and edges of a data graph should be included in the search space.

A second issue is that of semantic query expansion, a hallmark of semantic querying techniques. This is an important feature that allows a query context to be expanded beyond the set of keywords given by a user. Most semantic search approaches support this by expanding keywords using hierarchical relationships such as hyper/hyponymy defined in either a domain-specific ontology or a general purpose nomenclature and terminological system such as WordNet.

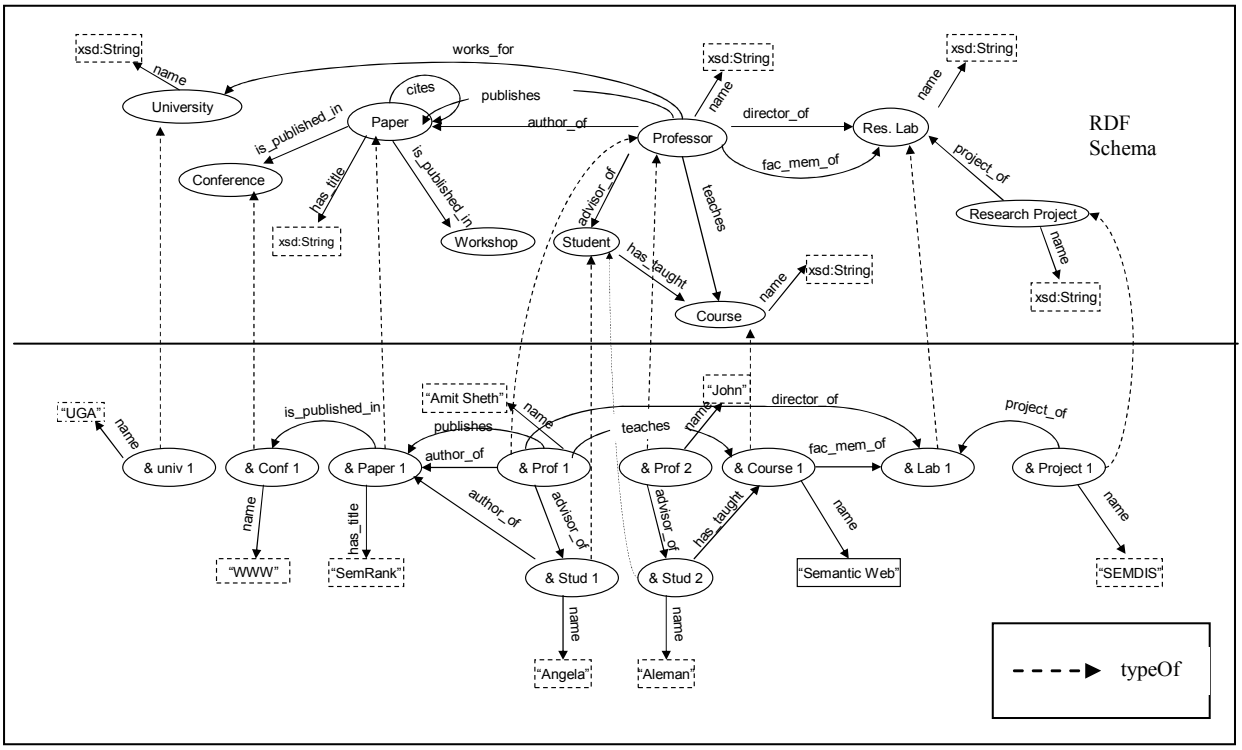


Figure 1: RDF graph of a LSDIS lab portal ontology

The approach described in this thesis, offers significant advantages over the current approaches for several reasons. First, we adopt the philosophy of a graph as a query result. In other words, given a set of keywords as a query, the result of the query is the relevant sub-graph of the data graph which connects the keywords given in the query. Named relationships in the graph paradigm give a significantly richer description of information in the query result. Further, it uses an advanced semantic query expansion technique that combines information about relationships between terms from both domain-specific ontology (RDFS) and the WordNet. More importantly, is uses other kinds of relationships besides hierarchical relationships. In particular for Semantic Web data graphs that have labels for nodes and edges, it is entirely possibly that a keyword that represents a noun may exist in a data graph in its verb form, perhaps labeling a relationship edge. Therefore, the semantic query expansion is performed by exploiting a variety of term relationships for expanding query context. For example, consider the above example *Amit Sheth Writings*. The ontology might contain a relationship *author_of*. The word *writings* cannot by any means be mapped on the *author_of* by performing WordNet expansion on *writings*. The related words such as *writing, authorship, composition* and *penning* can be obtained by performing direct expansion using WordNet. This set of expanded words is not related to the relationship *author_of* or the word *author.*

Another important feature is that of dealing with syntactic and lexical issues that arise when dealing with ontologies. Often, an ontology may label an edge or node with a compound words sometimes separated using separator character. For example, the relationships such as *has_title, author_of, has_taught* from Figure 1 contain multiple words separated by a separator or delimiter. Such occurrences will be missed without the use of special handling techniques.

In this thesis, we have presented an approach that goes beyond the direct text or pattern based search. And Hence, we are able to get results such as the one mentioned above. The following are the research contributions:

1 We propose a prototype system that allows keyword searches on direct or related words of a class, relationship or a literal.

2 We propose several pre-computed indices including the related word index and the Lucene [15] triple index that are used during query expansion. The pre-computation of indices helps in reducing the query processing time. The related word index stores information of each class or relationship to its set of related words. This index is unique since it uses the knowledge in WordNet and domain specific ontology to find related words. It uses a unique conversion algorithm to convert noun to verbs and verb to nouns. We have explained the importance of this algorithm and the intuition behind the creation of this algorithm in the pre-processing phase. As discussed earlier, most systems expand search terms using either WordNet or a domain-specific ontology.

3 We propose a semantic expansion technique which performs query expansion using the pre-computed indices.

4 We propose a filtering algorithm that filters triples

Chapter 2 – Background

2.1 Semantic Web

The current Web is a network of interlinked information that lacks semantics. The user navigates through a network of hyperlinked pages without having the prior knowledge of pages selected page. Semantic Web envisioned by Berners-Lee aims at solving this problem by adding metadata to the existing Web. This is a two step process which involves metadata extraction and annotation. This metadata mainly captures the relationships that exist between the resources of the Web. Adding metadata to the Web helps in making the Web machine readable and thus allowing software agents to process the contents of the annotated page. Many existing applications can take advantage of annotated Web resources. One of the applications that can benefit heavily are search engines. The current search engines retrieve Web pages by performing direct matches of search terms to the Web page contents. Thus, the trouble of extracting the relevant information is left to the users of the system. The users generally have to sift through the large set of retrieved pages to find the relevant information. This problem can considerably be reduced and even eliminated if the documents or pages are semantically marked-up or annotated with metadata. There are different techniques which have been used to extract metadata. The extraction of metadata can be done using semi-automatic techniques as in CREAM [25] or by using automatic techniques as in SemTag [33], SCORE [5] or Semagix Freedom. SemTag describes a Seeker platform for large scale text analytics, which can automatically generate semantic tags. SemTag was able to perform semantic tagging of approximately 264 million Web pages and could generate 434 million automatically disambiguated semantic tags for annotations. The extraction of metadata is done by populating the ontology with the extracted disambiguated entities. After populating the entities ontology is further enriched with additional relationships

that connect these entities. Thus, the extracted metadata needs to be enriched with relationships and validated before extracting more information. This process is repeated until a comprehensive ontology is obtained. The two most commonly addressed issues are entity identification and entity disambiguation. There are different techniques used in entity identification. The Semagix Freedom uses regular expressions to identify entities. The entity disambiguation is a tough problem [4]. The extracted information can now be represented in RDF, or OWL [19]. Generally, the information extraction is done on Web pages of a particular domain. Recently, the researchers in the semantic Web community have been focusing heavily on extracting information and storing them in ontologies represented in RDF or OWL, leading to large publicly available general and domain specific ontologies.

2.2 Semantic Web layer cake

The W3C organization has been a leader in developing technologies for the Web. Tim Berners-Lee, the founder of current Web is the director of World Wide Web Consortium. At W3C, Berners-Lee has actively been involved in the development of Semantic Web technologies for his envisioned Semantic Web. The Semantic Web layered Cake [28] is a layered set of Semantic technologies of Semantic Web, according to Berners-Lee and W3C. The Figure 2 shows the different layers of Semantic Web.

The W3C has been actively involved in the development of standards for each of the layers in the Semantic Web layered cake. In the Semantic Web layer cake, each layer can be seen as building on the below layers. Each layer in the layer cake uses the technologies or support of the layers below it as it gets more specialized and complex as we move higher in the layer cake. Currently, there are seven layers in the Semantic Web layer cake.

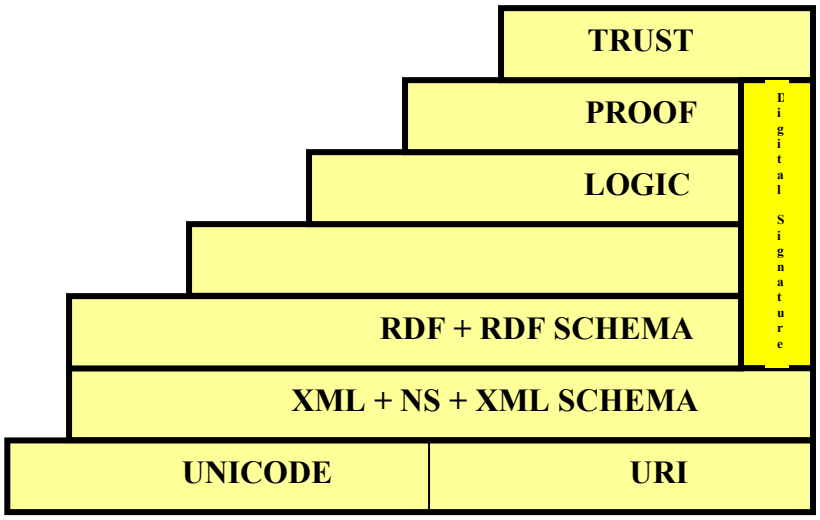


Figure 2: Semantic Web layer cake (proposed by Tim Berners-Lee)

We have explained each of the following layers in detail:

2.2.1 UNICODE and URI

URI (http://en.wikipedia.org/wiki/Uniform_Resource_Identifier) is a short string of characters used to identify a name or a resource. A URI can be used for locating a resource on the Web. Unicode [36] is an industry standard designed to allow symbols and text to be consistently represented and manipulated by computers.

2.2.2 XML, NS and XML Schema

XML [39] or Extensible Markup Language is a meta-language or a language for defining other languages. XML does not have a fixed syntax as HTML and allows creation of markup languages.

An XML namespace (NS) is a collection of names, identified by a URI reference, which are used in XML documents as element types and attribute names.

XML Schema [40] defines the semantics structure and contents of a XML document. XML Schema can be used to express shared vocabularies and allows machines to process XML documents based on the rules specified in the schema.

.

2.2.3 RDF and RDF Schema

RDF is metadata model recommended by W3C to model metadata about the resources on the Web. The basic element of a RDF model is a triple, which is comprised of a subject, predicate and an object. RDF model can be viewed as a directed labeled graph whose nodes represent entities and edges represent relationships which connect a pair of entities. Figure 1 shows the RDF graph model of the LSDIS Lab portal ontology. Additionally, RDF supports container such as Sequence, Bag or Alternative. A Sequence (RDF: Seq) represents a group of unique or duplicate resources or literals in a particular order. This, it is used when the order of the resources or literals is important. A Bag (RDF: Bag) represents a group of unique or duplicate resources or literals. The Bag does not take care of ordering of the resources or literals. An Alternative (RDF: Alt) represents a group of resources or literals that are *alternatives.* An example would a list of Websites where a particular book might be found. The Figure 3 shows the RDF graph model of a Sequence. A Sequence might be used to store information of all the

publications sorted in an alphabetical order. It shows the two publications of a resource *Amit*. The resource *Amit* is connected to a blank node which in turn connects to the publication resources via URI *http://w3.org/1999/02/22-RDF-syntax-ns#ns_1 or http://w3.org/1999/02/22-RDF-syntax-ns#ns_2.*

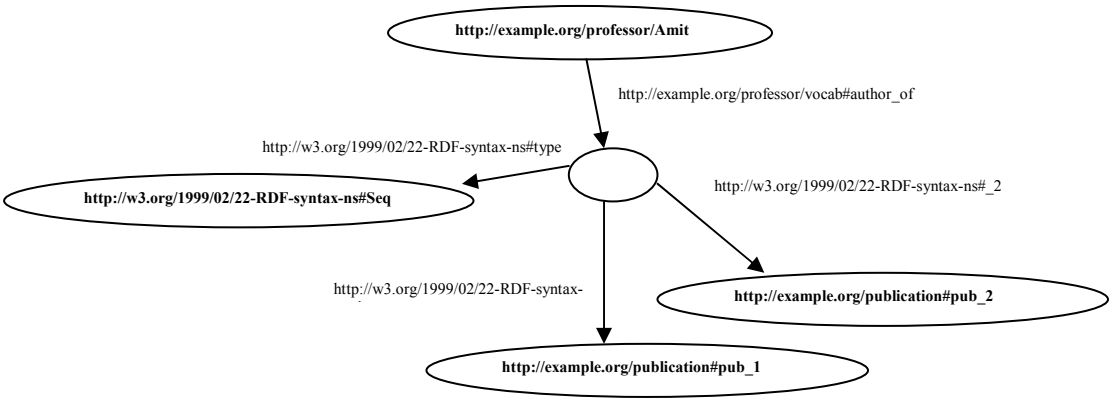


Figure 3: An example of a container RDF: Seq

RDF Schema [21] is a semantic extension of RDF for describing groups of related resources and their relationships with each other. RDF Schema defines a set of concepts and their relationship with each other.

2.2.4 Ontology

An Ontology forms the core of the Semantic Web technologies. Ontology [35] is described as a specification of conceptualization of a knowledge domain. An ontology can also be described as a controlled vocabulary which describes concepts and relationships of a domain in a formal and meaningful manner. This vocabulary also helps in representing an unambiguous view of a domain. Ontologies are represented using RDF and RDFS, OWL, or in DAML (DARPA Agent Markup Language).

2.2.5 Logic and Proof

The Logic and Proof are two different layers in Semantic Web stack. The Logic layer provides logical reasoning and the proof layer is responsible for deriving conclusions from proofs. Thus, it can derive conclusions which were not explicitly stated.

2.2.6 Trust

Trust layer is responsible for providing support for verification of trustworthiness of data, Web services and agents.

2.3 OWL

OWL is a formal language of modeling ontologies in the Semantic Web and facilitates greater machine interpreting capabilities by providing additional vocabulary support and formal semantics compared to RDF/RDFS, or XML. It has three sub languages OWL Lite, OWL DL and OWL Full which support increasing expressivity. OWL Lite supports simple constraints, classification hierarchies and reasoning. OWL DL supports expressivity through its language

constructs based on Description Logic. All constructs in OWL DL which are computationally complete and decidable. OWL Full offers maximum expressiveness, but offers no computational guarantees.

2.4 RDF Storage and Retrieval Systems

RDF and OWL have become W3C standard for modeling metadata for Semantic Web. With the increasing focus on data extraction, there has been a considerable interest among the Semantic Web community to develop efficient storage and retrieval system. Currently, there exist many open source storage and retrieval systems which provide query language support for information retrieval. The most well know systems are Jena [16], Sesame [30], and Redland [23]. Our lab i.e. LSDIS, has two different RDF main memory model implementations namely SemDis or Semantic Discovery API and BRAHMS.

2.5 BRAHMS

BRAHMS [13] was implemented in C++ as part of the SemDis or Semantic Discovery research project funded by NSF. BRAHMS was developed for faster access of RDF/RDFS information and was designed for loading large ontologies. It was designed as a main memory storage system. Currently, BRAHMS is being extended to provide SPARQL [32] support.

2.6 Semantic Discovery (SemDis) API

Semantic Discovery API [29] was influenced by BRAHMS RDF Store design and has the object design similar to that of BRAHMS. SemDis API uses an ARP RDF parser (http://www.hpl.hp.com/personal/jjc/arp/) to parse RDF. It provides a common high level

Interface for accessing information from RDF data. This high level Interface provides various abstract method implementations for retrieving paths, or instances, or literals, or classes, or relationships from the RDF graph.

2.7 Jena

Jena is storage and retrieval system developed by RDF Core Working Group of HP Labs. Jena is a Java API frame work for developing Semantic Web applications. It provides SPARQL query language support for accessing parts of RDF/RDF or OWL and inference capabilities through SPARQL's inference engine. SPARQL can be used to extract RDF sub-graphs, construct new RDF graphs based on the information in queried graphs and extract information in the form of URIs, blank nodes, and literals.

Jena uses ARP RDF parser to parse RDF or OWL. It provides persistent storage of RDF using relational database. Some of the relational databases supported by Jena include ORACLE, MySQL, Microsoft SQL Server, and PostgreSQL.

2.8 Sesame

Sesame is an open source RDF storage and retrieval system which provides relational database support for storing RDF and provides inferencing and querying support. Sesame provides support for various query languages including SPARQL which is similar to SQL. Sesame provides flexibility of the underlying storage system and supports various storage systems including relational database, main memory storage, file systems etc.

2.9 Lucene

Lucene search engine is a Jakarta open source project used to build and search indexes. It can index text documents and retrieve them based on various search criteria. It provides a basic framework which can be used to build a full-featured search engine. Lucene indexes using document objects. Thus, the text documents which are to be indexed have to be converted to document objects. Each document object consists of a set of field objects containing name and value pairs. The *name* is of type String and *value* can either be a String or a Reader object. Field class in Lucene provides various methods depending on whether the text in the *value* part of the field is tokenized, indexed or stored. Depending on the requirements some of the text information is tokenized, indexed or stored. A Lucene allows users to search on the values of these fields and this is done using an IndexSearcher object. All query terms are parsed using an analyzer, which is wrapped within the query object. Lucene provides four different analyzers to parse the search terms in the query: the StopAnalyzer, WhiteSpaceAnalyzer, SimpleAnalyzer, and StandardAnalyzer. An analyzer takes in a stream of text and returns a set of tokens. Lucene tokenizes the queries depending on the kind of analyzer. The StopAnalyzer is used to split the terms and eliminate any stop words that exists in the query. The WhiteSpaceAnalyzer splits the query terms based on white space. The SimpleAnalyzer splits the text at non-character boundaries, such as special characters (‘@’,’&’ etc.). The StandardAnalyzer is the most sophisticated parser with rules for email addresses, acronyms, hostnames, floating point numbers, as well as the lowercasing and stop word removal. Lucene provides two important classes to build and search on a index. IndexWriter class is used to build the index and IndexSearcher class to search on the built index. Lucene provides tools to generate query objects called Query Parser. The QueryParser class takes the search terms or queries and wraps them in a

query object. This query object is later used by the *search* method in the IndexSearcher class. Later, the IndexSearcher returns the *Hits* object for the query. This *Hits* object is similar to a vector and contains the ranked list of document objects for a given query. For our use, we have implemented a PorterStemAnalyzer by extending Lucene's analyzer class and have used it to stem the words to its base forms to eliminate any stop words.

2.10 WordNet

The WordNet is an online lexical reference system developed at the Cognitive Science Lab of Princeton University. Currently WordNet contains about 150000 words organized into 115,000 synsets of nouns, verbs, adjectives and adverbs. Each synset or set of words are related to other synsets by common relationships such as hypernym or hyponym, and meronym or holonym, verb groups i.e. groups of related verb forms, synonyms or similar meaning words, derivational forms or morphological forms etc. There exist different groups of synonymous words that are grouped based on the sense of a particular word. For example, the word *faculty* has two synsets since it has different senses based on the usage context.

WordNet can retrieve the different sets of related word information depending on the POS (Part of Speech) of the word. For example, the word *teaches* has related word forms such as verb groups, synonyms, derivational forms, and hyponyms. The hyponym or hyponym relationship indicates that one (hyponym) is a kind of other (hypernym). The meronym or holonym relationship indicates that one (meronym) is a part of other (holonym). The derivational form of a word is given by adding the morphological suffixes. For example, derivational form of a word *write* is *writing*.

WordNet indexes all words in its singular form or in its base form depending on whether the word is a noun or verb, respectively. Hence, all nouns have to be converted to its singular form and all verbs have to be reduced to its base form before searching in WordNet. As part of this research, we have implemented a conversion algorithm that converts plural words to singular form. We have also implemented a stemmer to reduce a verb to its base form. Additionally, we have implemented an algorithm which returns all the derivational forms of a word.

2.11 Touch Graph

TouchGraph (www.touchgraph.com) uses a spring-embedding algorithm to display graphs. There are many applications that are developed using TouchGraph. TouchGraph basically reads an XML file and outputs a graph that is representative of the XML file.

Our system uses a TouchGraph applet to display the sub-graphs. This is done by serializing the resultant triples in a XML file, which is read by TouchGraph.

Chapter 3 – Approach

We have developed a prototype search and retrieval system which provides answers to the queries entered by the users. The system accepts queries in the form of keywords and returns a sub-graph relevant to the query. The system supports different types of searches. Our system uses an integrated approach that completely utilizes the knowledge in the WordNet and the ontology to expand search terms. A unique feature of our system is that it allows keyword searches on direct or related words of ontological classes or relationships or literals. There exist to main phases in our system: pre-processing phase and the query processing phase. During the pre-processing phase the system builds various indices which are used in the query processing phase. The system builds different indices including the related word index and the Lucene triple index. We have explained each of them in detail in the following chapters. The system pre-computes all the indices at the pre-processing phase in order to reduce the time needed for query processing. Each of the indices built during the pre-processing phase has its own significance during query processing. During the query processing phase, the system processes all the search terms, expands them and finds matches in the ontology. Since our system accepts keywords, there exist times wherein the search term matches a class and a relationship. In which case, the system needs consider one of them. We handle this using an elimination algorithm which is explained in the following chapter. The expanded sets of words are later matched to the ontological instances, classes and relationships. After the semantic query expansion, the system matches the expanded terms on to the instances, classes and relationships in the schema and instance file. Later, it retrieves the triples using the Lucene triple index. Then, the retrieved triples are filtered and later joined to form a meaningful sub-graph, which are later displayed to the users.

Chapter 4 – Architecture of the system

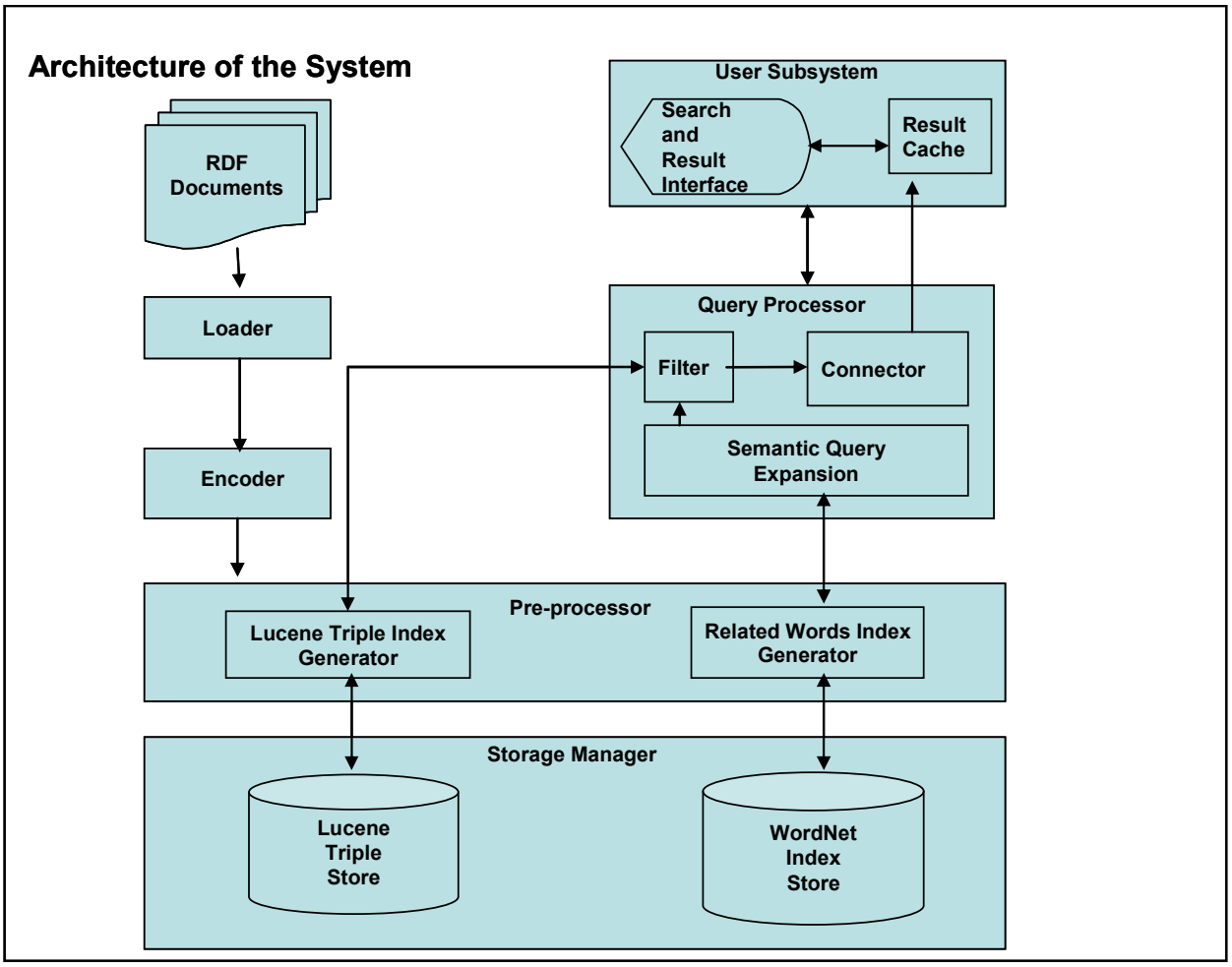


Figure 4: Architecture of the System

Currently, the system is designed as a client/server or Web based application. The system administrator of the system will be responsible for loading the ontology and building of indices. Initially, the system administrator uploads the RDF documents into the system using a Loader, which loads the RDF documents into memory. Later, the users of the system are allowed to query on the uploaded RDF documents. Then, the RDF encoder encodes the classes, instances and relationships in the ontology and data file by assigning unique numbers. The encoding is done for efficiency reason. Thus, each URI in the RDF graph is assigned a unique number.

After encoding the graph, the system builds the triple indices and related word indices, which are later serialized to disk by the storage manger. The related word indices are generated using WordNet and NLP (Natural Language Processing) techniques and are built for every class and relationship in the ontology. Additionally, the generator creates an inverted index that stores the information of the related words to its respective class or relationship in the ontology. The system also generates Lucene triple index for every triple in the RDF documents. All the indices which are pre-computed during the preprocessing phase are used during the query processing phase while performing query expansion and filtering of triples. During the query processing phase, each of the search terms are expanded and matched to the ontological classes, relationships and instances. Matching to ontological concepts and relationships has posed challenges. Most ontologies contain relationship (URI's without namespace information) which are comprised of words separated by a delimiter. The relationship names such as *has_Name, has_first_name, has_Course_Name, teaches_course* are some of the examples of such URIs. We have tried to address these issues. We have used tokenizers to tokenize them to individual terms, stop word removers to remove any stop words from the compound word, stemmers to reduce words to their base form and converters to make conversions from plural to singular form. Once the pre-processing is complete, the system is ready for querying. The query processor expands the entered search terms and matches the expanded set of terms to the ontological classes, instances and relationships to retrieve relevant triples that match the expanded set of terms. The filtering module helps the query processor to filter the irrelevant triples and displays only the triples that are relevant to the query. Later, the triples are connected to form a meaningful sub graph by the Connector.

Chapter 5 – Preprocessing Phase

The system builds various indices including two main indices: Lucene triple index and the related word index during the preprocessing phase. As explained earlier, all these indices are pre-computed to reduce the query processing time. During pre-processing phase, the system encodes all the nodes and edges of the RDF graph. The encoder gives unique number to the nodes depending on whether the node is a literal, instance or a class and is used during the query processing to retrieve triples. The encoder also builds an index that stores information of each of the class ID, instance ID or relationship ID or a literal ID to its respective URI and an inverted index for the same. The system also builds other indices such as class to derived instance index, and instance URI to its respective literal index. Later, all the created indices are serialized to disk. We have explained each of them in the following sub chapters.

5.1 Lucene triple index generator

It builds Lucene triple index for all the triples in the schema and instance file. As mentioned in the previous chapter, Lucene can be used to build and search index. Lucene allows loading of large RDF documents without having to constrain to the memory of the machine and allows faster retrieval matched of RDF triples. The significance of this index is that it allows faster retrieval of triples. This index is used in the query processing phase to retrieve triples that matched the classes, relationships or literals. The index generator builds this index in the following steps:

Initially, the Lucene triple generator gets all the triples in the dataset and schema from the loader and builds the index for every triple using its subject, predicate and object or literal and their respective encoded numbers. It creates Lucene field objects for each of the subject,

predicate, object or literals in the triple and their respective encoded numbers. These triple indices are later used by the query processor to retrieve triples based on a subject, or literal, or a property from the index store. In order to allow searches on property names, we have indexed all the property URI's without any separators or delimiters. For example, the property *has_name* was converted to *"has name"* before indexing the triple associated with this property.

Indexing RDF documents using Lucene is little different from indexing text files. Lucene has primarily been used for indexing unstructured text. Recently, Lucene has been used to index RDF triples [34]. The ways in which these triples are indexed is very much dependent on its usage. We have indexed RDF triples by the creating Lucene field objects for each of the subject, predicate and object or literal. We have also created field objects that stores subject ID, predicate ID and literal or object ID. Thus, each triple has six field objects which are wrapped in a document object. This allows searches either on subject, predicate, object or literal and can retrieve the triples and their respective subject, predicate, and object ID's. We have created triple index on literal statements or triples that contain literals, triples that are comprised of domain or range classes and on triples that contain only instances (instance URI).

5.2 Related word index generator

It builds the related word index for each of the classes and relationships in the ontology. This index significantly reduces the query processing time of a query. The index generator builds this index using the information in WordNet and ontology which is the most unique part of our system. The Query Processor uses this index to find related words of a given search term. The index also helps in matching of the search terms to ontological classes, and relationships. This was possible because of its unique processing of URI or label while building the index.

The related word index generator uses a unique conversion algorithm that helps in matching the related words of a class to its respective relationship. We have explained this at the end of this sub-chapter. The same applies to properties in the opposite manner. Hence, conversion algorithm is vital for our system. Most ontologies contain classes such as *Author* and relationships such as *has_publication*, or *author_of, has_composed etc* associated with that class *Author. Consider another example,* class *Instructor* might have relationship such as *instructs,or teaches_course, teaches etc.* The only way we can match to these relationships or classes is by using the conversion algorithm that converts *noun to verbs or verb to nouns* depending on the POS of the word. Thus, the queries such as "*Eric Miller" instructor or "Eric Miller" teacher* return similar results. The results of this query are the courses or classes thought by *Eric Miller*.

This index also helps in entity disambiguation. The entity disambiguation is done based on the type of the literal. Hence, a query like *Teacher Eric Course* should return similar results as *Instructor Eric Course,* even though ontology may or may not contain a class related to *Course.* This is because the words *Teacher* and *Instructor* are related to each other.

It is a two way index that stores the information of each class or relationship to its respective set of related words, and from the set of related words back to its respective class or relationship in the ontology. The system creates these indices for each of the class or property label or URI in the ontology. Hence, the names of URI or the label need to be meaningful enough to create these indices. The system also provides an option for the system administrator to choose either the label or URI based on the ontology. The indexer also generates information that links each class to its respective direct or derived instances. The derived instances can be defined as instances of all the sub classes of a class. It also builds an inverted index that stores information of each instance bag to its respective class in the ontology. This index is used during the query

processing to filter any unwanted triples. Initially, it builds the related word indices for all the classes in the ontology. Later, it builds the indices for all the properties in the ontology in a similar manner.

The related word index generator performs the following steps for building indices for each of the classes and properties in the ontology:

Step 1: Processing of URI or label: The index generator does processing depending on whether the index generator creates indices using URI or label. The creation of indices on class or property label is a straightforward process compared to the creation of indices on URI. It processes each of the URI by tokenizing using different sets of commonly occurring delimiters. Then, it removes any stop words occurring in the tokenized words. Later, it performs stemming or conversion from plural to singular based on whether the term is verb or noun. This is necessary since WordNet indexes on singular forms of the nouns or base forms of the verbs. Hence, all words have to be processed before searching in WordNet index. The index generator uses a stemmer to remove stop words and to reduce the word to its base form.

The conversion algorithm converts a word to its singular form. This is done based on certain linguistic rules and string manipulation. The conversion algorithm uses a set of plural suffixes and its respective singular suffixes to convert a word from its plural to singular form. It works by pattern matching of all the suffixes against the word and if a match is found, it is replaced with a sub-string based on the linguistic rule associated with the suffix. The word is checked against all the suffixes and later, validated by searching in the WordNet. This ensures that the converted word is valid and is in its singular form.

We have extended Lucene Analyzer and created our own analyzer class named PorterStemAnalyzer. This analyzer processes input text by converting input text to lower case,

tokenizing it using the commonly occurring delimiter, removing any stop words and stemming words to its root forms or base form. It removes any stop words defined in the stop word array of this class. We have used this analyzer to reduce the words (verbs) to its base form.

Step 2: Later, it finds the all related word for each of the words. The related word for a given word can be retrieved by searching the WordNet for related words such as hypernym, hyponym, related verb forms, synonyms, meronyms, and inherited member holonyms. Currently, the index generator considers only the most frequently occurring sense of the related word. There exist a many senses for a given word. As explained earlier, the word *faculty*, has two senses and each of the sense has many synsets. In order to reduce the amount of expanded words, we have considered only the most frequently occurring sense. As explained earlier, our expansion technique exploits the hierarchical information in the ontology. The generator adds all the super-classes or super-properties of all the tokenized words to the bag of related words. This process is important since the index generator uses knowledge present the ontology and WordNet to build related word index.

Step 3: Then, for each of words in the bag, the index generator does conversions from noun to verb or verb to noun and gets the related words for each the converted words by searching in WordNet. This conversion is very important since relationships such as *teaches*, or *instructs* will have related words such as *teacher* or *instructor* respectively. This algorithm works by using a set of linguistic rules to convert a word in one form to other. Based on the POS of the word, the word is either converted to verb form or noun form respectively. It works by pattern matching either on the noun to verb suffixes or verb to noun suffixes. Later, it adds the suffix based on the linguistic rule associated with the matched suffix. Then, the algorithm adds all the related words

of the converted words by searching for all the related words in WordNet. It searches for all the related words explained in step 2.

Step 4: It then adds all the derivational forms of all the words in the bag, which is obtained from the derivational form algorithm. The derivational form of word *write* is *writing.* The derivational algorithm works by initially checking the POS of the word. Then, depending on the POS of the word, the algorithm performs pattern matches either on the noun suffixes or verb suffixes respectively. Once a match is found, the algorithm replaces the suffix with its respective suffix based on a linguistic rule. Later, the retrieved words are validated by searching in WordNet before updating them in the bag of related words.

At the end, the index generator associates the processed URI or label to the computed set of related words. The indices are later serialized on to disk by the storage manager.

The following example illustrates the creation of a related word index for *author_of* relationship. The index generator initially tokenizes the relationship into two words *author* and *of.* Then, it removes any stop words that exists in the set of tokenized words. Thus, the word *of* is removed. The word *author* is in its singular form and hence it does not have to do any conversion. Later, it gets all related words of *author* including *writer* and adds them to the bag along with its super-classes or super-properties. Then, it gets all the verb forms of the word *writer* such as *write,* from the conversion algorithm. Later, it adds all the related words of *write* such as *compose, pen, and indite* into the bag. At last, it adds all the derivational forms of all the words in the bag. The derivational form of word *write* is *writing*. Thus, the related words of author*_of* relationship are *writing, write, indite, compose, pen, writer, author* etc.

Consider another relationship *teaches_course*. The index generator initially splits the relationship into two words *teaches* and *course* and removes stop words. Then, it gets all the

range classes of the relationship and eliminates any range class that exists in the string URI of the relationship. In our example, it eliminates the word *course*. Later, it reduces the word *teaches* to its base form *teach* before finding the related words using WordNet and the conversion algorithm, which allows searches only on the singular form or base form of a word. It now stores the information from the relationship *teaches_course* to the related words of *teach.*

Consider the class URI or label *Professor*. The index generator processes this label by finding the related words using the WordNet and conversion algorithm. Here, the conversion algorithm will not be of much help. The related words of *Professor* will come in handy while processing the query. The user can enter related words of class *Professor* and also the sub-classes of class *Professor*. This index allows the system to find all the direct and derived instances of the class for any word related to the word *Professor*. Thus, a query like *Faculty Sheth writings* and *Professor Sheth writings* will return similar results even though ontology does not contain any sub-classes of *Professor*. Now consider another class URI or label *Author*. The index generator processes this label by finding all the related words of *Author* such as *Writer.* The index generator also obtains other related words which we have not mentioned in this example. The index generator then converts the word *Writer* to verb *write* and also gets the derivational forms such as writing, writes etc. Later, it gets the related words of all the words. Hence, it has related words such as compose, pen, publish etc. which are obtained through the conversion and searching of related words in WordNet for all the words in the bag. Consider a scenario wherein the ontology has relationship such as *publishes* or *has_published or composes* which connects the class *Author* to his writings. The user enters the query *Sheth writer or Sheth Composer* will be able to retrieve all the writings *of Amit Sheth*

5.3 Class to derived instance index

This index store information of each of class to its respective direct and derived or indirect instances. This index is necessary during query processing and is used by filtering algorithm to remove irrelevant triples. We will discuss this index in the query processing chapter. This index is used during query processing to find all the instances of a matched class.

5.4 Instance URI to literal index

The instance URI to literal index stores information of each instance URI to its lateral. Most RDF documents have instances defined using relationships such as RDF labels or RDF. This index stores information of each URI to its respective literal that are connected by the relationships such as RDF labels, RDF name, RDF given name, RDF last name, etc. This index is used during the query processing to find the URI associated with the literal.

Chapter 6 – Query processing phase

The system processes the search terms during the query processing phase. The query processor uses several pre-computed indices in this phase to expand search terms. It performs the semantic query expansion of search terms and matches the expanded terms onto the ontological classes or relationships. Later, the filtering algorithm does the filtering of triples depending on the query.

6.1 Semantic Query Expansion

The query processor performs the expansion of search terms using a unique semantic expansion technique. The expansion of search terms is done using various pre-computed indices which were created during the pre-processing phase. The query expansion works by expanding the search terms depending on whether it is a literal or a word that is related to a class, or a relationship. Later, the query processor expands the search terms using the direct or related words of ontological classes or properties. Later, the query processor finds the related words for a given search term using a pre-computed related word index. As discussed earlier, the related word index stores information of each class or relationship to its respective bag or set of related words. Additionally, the query processor also keeps track of any matched literals, classes or relationships. The matching of triples is done by searching for each of the expanded terms in the pre-computed Lucene triple index. Later, the filtering algorithm filters any irrelevant triples.

The semantic query expansion involves the following steps:

Step 1: The search terms are initially compared to the instance labels or names in the ontology. If the match is found, the respective instance label or name is added to list of found instances. It uses the pre-computed indices and performs pattern matching of each of the search terms to the

instance labels or full names (including given name or first name and last name) in the ontology. Later, it adds all the found literals to the bag of expanded terms.

Step 2: The search terms are matched against the classes in the ontology. This is done by initially converting each of the search terms to its singular form or base form. Later, it performs direct or pattern matches of each of the converted words to the words (keys) in the pre-computed related word index. This index stores information of each class (reduced to it base form or singular form) to its respective set of related words. If the match occurs, the query processor updates the bag of expanded query terms along with its related words. Otherwise, the transformed search term is matched against each of the word in the bag or set of related words. The bag of related words is obtained for each class using the pre-computed related word index. If there was a match, it adds the all the found classes and related words to the bag of expanded query terms.

Step 3: Similar steps are carried out while matching the search terms against the ontological properties. It uses the pre-computed property indices that stores information of each of the property to its respective related words.

Step 4: Later, the Lucene triple index is used to retrieve triples that match the expanded terms. This step is clearly explained in the next sub-chapter. Filtering algorithm retrieves triples using the Lucene triple index and performs the filtering of triples depending on the type of query.

Consider an example query *Sheth writings*. Here, the user is looking to find all the *writings* of *Amit Sheth*. The query processor performs the semantic query expansion using the pre-computed indices. It performs the semantic query expansion for the above query by performing the following steps:

When the user enters the query *Sheth's writings*, "*writings"* is initially converted to its singular form *writing*. Later, it checks the previously computed bag of related words or related

word index of class and relationships to see if *writing* belongs to any of these bags. We had explained the creation of related word index in the previous chapter. Later, the search term *Sheth* is matched to an instance literal *Amit Sheth* and *writing* to a relationship *author_of*. As discussed earlier, related word index is an index that stores information of a class or a property to its respective bag of related words. The word *writing* is checked in the pre-computed related word indices for classes and properties. It initially checks *writing* with the keys in the related word index of classes or properties and if there wasn't a match, it is checked in the bag (value) which contains a set of related words for a given word or key. The query processor was able to find the word *writings* in a bag that contains *author*. Later, the triples associated with the matched instances and properties are retrieved using a pre-computed Lucene triple index. Thus, the query processor retrieves all the triples associated with *Amit* and *author* using the Lucene triple index. The filtering algorithm performs the filtering of triples depending on the query. At the end, a sub-graph is displayed to the user. In the above example, a sub-graph that connects Professor *Sheth* to his *writings* is displayed to the user.

6.2 Filter

The filtering module in the query processor is responsible for filtering triples based on the type of query. As discussed earlier, the Filter allows four different kinds of queries. The filtering module in the query processor uses various pre-computed indices including the Lucene triple index to find relevant triples. The filtering algorithm retrieves all the triples that matched the literals and relationships in the dataset. The unique feature of this algorithm is the way in which it handles the RDF containers such as Sequence. This algorithm needs to detect RDF containers in the ontology and add the triples based on whether the matched instance or property connects

to a blank node or not. The number of triples that needs to be added depends on whether the query matched instances (i.e. instance URI associated with the matched literal) or relationships whose triple contain a blank node.

Please refer Figure 3 for the following example. Figure 3 shows a resource or entity of type *Professor* connected via relationship *author_of* to a blank node of type RDF Sequence. Later, the blank node is connected to other instances via RDF sequence number relationships. Thus, a query such as *Amit Writings* should retrieve the following triples:

1. A triple that connects the literal *Amit* to instance URI *Amit*.
2. A triple that connects the instance URI *Amit* to the blank node
3. A set of triples that connect the blank node to a set of papers, articles, presentations etc.
4. A set of literals associated with the instance URIs of papers, articles, presentations etc.

The algorithm retrieves the following triples in case the matched triples do not contain blank nodes.

1. A triple that connects the literal *Amit* to instance URI *Amit*.
2. A set of triples that connect the instance URI *Amit* to a set of papers, articles, presentations etc via relationship *author_of*.
3. A set of literals associated with the instance URIs of papers, articles, presentations etc.

The filtering algorithm within the filtering module works in the following steps:

Step 1: Initially, the filtering algorithm builds an index that stores information of each of the instances (instance URI) of the matched class to its respective literal. This is done in the following manner:

After performing semantic query expansion, the filtering algorithm uses a pre-computed index which stores information of each class to its respective derived instance to find all the derived instances of the matched classes. Later, it gets the instance URI for each of the derived instances and builds an index that stores information associated with each of the instance URI of the matched class to its respective literal.

Step 2: eliminates any irrelevant relationships using an elimination algorithm. As discussed earlier, sometimes there might be instances where a search term matches a class and a relationship in the ontology. This causes problems during filtering since the filtering algorithm needs to perform the filtering of triples depending on whether the search terms matched the classes, the relationships or the literals. Hence, the removal of irrelevant properties is necessary before retrieving and filtering triples. This is done by the elimination algorithm.

The elimination algorithm uses two indices: one of them stores information of each of the matched instance URI and its respective literal and the other stores information of each of the matched class instance URI and its respective literal. These indices are built at the query processing time using a pre-processed index which contains information of each of the literal to its respective instance URI. The elimination algorithm works by retrieving all the triples that contain the matched properties and then checking to see if the triples contain either the matched instance URI or the URI of the derived instances of the matched class. If the triple does not contain any of the URI, then the property is removed from the set of matched properties.

The filtering algorithm filters the triples depending on whether the search terms matched the ontological classes, relationships or literals. The first kind of query matches the search terms to the ontological classes and instances. The second query type matches the search terms to the instances and properties in the ontology. The third kind of query matches the search terms to the

classes, instances and relationships in the ontology. The fourth type of query matches the search terms to a class or an instance or a relationship in the ontology. As discussed earlier, the system allows users to search using the direct and related words of the classes, relationships or instances in the ontology. The results of the search are triples that contain the matched classes, or instances or relationships.

Consider an example *Professor Amit.* This type can also be used for analysis and validation of information. If a user wants to know if *Amit* is a *Professor*, he can always check by searching using the above mentioned query. Sometimes the entered search term may not directly match the classes or relationships. The system uses the related word index generated at the pre-processing time to handle this problem. Also, the ontology may not have any direct instances associated with the given class. Initially, the algorithm gets the derived instances of the class. Later, the filtering algorithm filters based on whether the query term matches the classes, instances or properties. The above query matches search terms to the classes and instances in the ontology. Thus, the filtering algorithm must be able eliminate any irrelevant triples that do not have *Amit*. Furthermore, the filter algorithm removes any triples that are not the instance of class *Professor* using a pre-computed index that stores the information of each class to its respective derived instances.

The filtering algorithm performs the following steps upon matching the instances and relationships in the ontology. Initially, it gets all the triple labels that contain the matched instance URIs. Then, it gets the triples that contain the matched relationships and checks to see if triples contain the matched subject or object URIs. If there was a match, it adds both of them to the triple set. It then checks to see if the triple contains a blank node which connects to other instances. If the blank node exists, it adds all the triples associated with the blank node. Later, it

adds all the subject and object labels associated with the triples. An example will be *Sheth writings* as explained earlier or *Sheth Semantic Web,* wherein the user expects to see all information that has links *Sheth* to *Semantic Web.*

The last type of query matches the search terms against the classes, instances and relationships in the ontology. This query can be used to disambiguate entity. The algorithm initially gets all the triple labels which are associated with the matched class instances. Then, it checks to see if the retrieved triples contain the matched instance URIs. Later, it gets all the triples that contain the matched relationships and checks to see if any triple contains the subject or object URIs. If the matched triple contains a blank node which connects to other instances, the filtering algorithm adds the triples associated with the blank node. Later, the subject, and object labels associated with the triples are added. An example for the given query will be *Professor Sheth writings* as explained earlier or *Courses Sheth instructs,* wherein the user expects to see all courses that Professor *Sheth instructs.*

6.3 Connector

This is a component of the query processor. Here, the retrieved triples are joined on the common subject or object. It also adds the literal statements associated which are associated with each of the instance URI. The resultant sub-graph is later displayed to the user. The connecter joins the triples using the encoded ID which were retrieved along with the triple.

Chapter 7 –Implementation Details and Comparison

This application is a Web based application, developed using Apache Tomcat, Servlets, TouchGraph applet, SemDis API, Lucene, and WordNet. The application has two main phases: pre-processing phase and query processing phase. During the pre-processing phase, the RDF files including the schema and instance file are loaded in to the system. We have used SemDis API to load RDF files. Later, the indices are created and serialized on to disk. The entire pre-processing phase is handled by the System Administrator. During pre-processing the system creates a static object of the ontology model of RDF documents. This ensures that there exists only one model in the memory.

The serialized indices are loaded into memory during the query processing phase. Again, all the objects were made static to ensure that the indices are loaded only once and any subsequent request will use the loaded indices. After query expansion and filtering, the system displays the resultant output in the form of a graph. We have used TouchGraph to display graphs. The system is designed to store information of all the previous searches such as matched literals, relationships, or classes. This information is stored in static maps and the map is updated every time a new search is performed. The current system returns results by displaying two sets of triples namely: literal statements and instance statements respectively. It can also to display a sub-graph of the retrieved triples.

Comparison was done by comparing the precision values of our system to a well known keyword based retrieval system like Google. We have calculated the values for different queries supported by our system and have used the LSDIS lab portal ontology for the comparison. The ontology contains most up to date information of real world entities and has been populated with the most recent information of entities such as professors, publications, research labs etc. It has

17 unique properties, around 1558 literals, 112 schema properties (properties connecting the classes) and 1024 instance statements (properties connecting instances).

Currently, our system can only extract the metadata information relevant to the query, but can easily be extended to retrieve semantically annotated documents. The comparison was not straight forward since Google returned documents, while our system returned a sub-graph.

The definitions of precision and recall are as follows. The precision can be defined as the ratio of the number of relevant results retrieved to the total number of results returned. While performing the evaluation, we made sure that Google retrieved Web pages that contained all of the queried terms and restricted the searches in Google to the LSDIS lab Web site.

Table 1: Precision values of a keyword based search and retrieval system

No.	Query	Hits	Precision
1.	*Sheth teaches courses*	9	6/9 = 67%
2.	*Sheth instructs courses*	3	0/3=0%
3.	*Sheth teaches classes*	6	1/6 =16.7 %
4.	*Sheth teaches*	14	6/14= 42.8 %
5.	*Amit writings*	2	2/2 = 100%
6.	*Sheth advises students*	3	0/3 = 0%

Table 1 shows the Google's precision values for different queries on the LSDIS site. The precision values shown in Table 1 are based on the documents retrieved on June 28th 2006. For

the first query, Google retrieved 6 documents which were partially related to the query. The Google was able return Web pages that contained the two courses thought by Prof. *Sheth* and was also able to find pages that contained the names of the Teaching Assistants. The Google was not able to retrieve any relevant result for the query *Sheth instructs courses*. The query *Sheth teaches classes,* retrieved 6 pages which contained words such as *classes, teachers or teaches and Sheth* in it. Only one of them was relevant to the search. The results of the fourth query were similar to the first query. The query *Amit writings*, resulted in two hits and all of them were relevant to the search. In query 5, the user was looking to find all the *writings* such as publications, book chapters etc. that has been authored by Professor *Sheth*. The query *Sheth advises students*, retrieved 3 results and again none of them were relevant to query. In query 6, the user was looking to find all the students advised by Professor *Sheth*.

Table 2: Precision values for our system

No.	Query	Hits	Precision
1.	*Sheth teaches courses*	3	3/3= 100%
2.	*Sheth instructs courses*	3	3/3= 100%
3.	*Sheth teaches classes*	3	3/3=100%
4.	*Sheth teaches*	3	3/3=100%
5.	*Amit writings Web services*	14	14/14=100%
6.	*Amit writings*	204	204/204=100%
7.	*Sheth advises students*	13	13/13 =100%

The Table 2 shows the precision and recall values of our system for the above queries. The ontology had most up to date information of all the entities. Thus, the information in the ontology was accurate and complete. The values were calculated based on our interpretation and meaning associated with the query. Our system returned accurate results for all the above queries. The output of the system was a relevant sub-graph relevant to the query. Please refer to the appendix chapter to see some of the results retrieved by our system.

Please refer to the LSDIS portal ontology shown in Figure 1 for the above queries. Though the figure does not show many instances, we have populated the ontology with similar real world instances. For the first query, the system matched *teaches* in the query to the ontological

relationship *teaches*. The system matched the search terms *Sheth* and *Courses* to entity instance *Sheth* and class *Course* respectively. The results of the above query were all the courses thought by Prof. *Sheth*. Consider the second query *Sheth instructs Courses*. Even though the ontology did not contain relationship *instructs,* the system returned results similar to the first query. The system retrieved similar results for the third and fourth queries. The fourth query did not match to any of the classes in the ontology, but was still able to return results which were similar to the queries 1, or 2, or 3. The query 5 returned all the *writings* of *Amit Sheth*, which had *Web Services*. The above query matched the search term *Amit* to the instance *Amit* and *writings* to the relationship *author_of* and *Web Services* to all the entity instances that contained *Web Services* in its RDF labels. The results of query 6 were all writings authored by Prof. *Amit*. In this case, it matched *Amit* and *writings* to instance *Amit* and relationship *author_of* respectively. The seventh query returned all the 13 students advised by Prof. *Amit*, wherein *Sheth* matched to an instance, *advises* matched to a relationship *advisor_of* and students matched the class *Student* in the ontology. Over all, we think that our searches were very accurate compared to keyword based retrieval system such as Google. Our system displayed sub-graph that was relevant to the search. Additionally, our system allows keyword searches on related words. The information retrieved by our system was based on knowledge available in the ontology. Currently, the expansions of search terms are limited to the information in the ontology and WordNet. The results displayed by our system are based on the matches in the ontology. And hence in order to retrieve results, it is mandatory that the query matches to any of the lateral, relationship or classes in the ontology.

Consider the query *Sheth Publications*. Our system was not able to find any matches for *Publications* in the ontology and hence could not retrieve any meaningful results. The system processes the above query in the following steps:

Initially, the query processor matches the search terms to the ontological classes, relationships or literals. The search term *Sheth* is matched to the literal *Amit Sheth*. The search term *publication was* converted to its singular form *publication* before searching for related words in the related word index. The system was not able to find any word related to *publication* in the related word index and hence could not retrieve any triples. As explained earlier, the related word index was created for each class or relationship in the ontology. This was done in order to reduce the expansion time while processing the query. Our system was developed to retrieve information from the ontology and hence would not return triples or sub-graph if the search term does not match to the ontological classes or properties or literals.

Chapter 8 – Related Work

In this thesis, we have proposed a prototype system that allows keyword searches on direct or related words of an ontological class, or relationship or a literal. We adopt an integrated approach which uses the knowledge in WordNet and ontology to expand search terms. The key feature of our system is that it allows keyword searches especially on relationships and can display sub-graphs that are relevant to the query. Additionally, our system allows users to enter a set of keywords that may or may not be related to each other.

While many of these systems [1, 16, 30] are based on formal querying languages [7, 22, 32], a few allow the querying of Semantic Web repositories using keyword queries [9, 12, 14, 24] or rdf path fragments [34]. Our system is different from the above systems, as it supports keyword search not only on literals, but also on related words of classes and relationships. The systems [9, 12, 14, 24] allows users to enter only a single keyword or literal per search, unlike our system which allows multiple keywords in a single search. The key feature of our system is that it supports searches on related words of classes and relations, unlike the direct or pattern based keyword searches supported in [9, 12, 14, 24]. Additionally, our system displays search results in the form of a sub-graph. Kowari [1] is a native RDF store that stores information using a RDF database. It allows users to query using iTQL RDF query language, which is similar to SQL. Sesame is a RDF database with support for RDF Schema inference and querying. It supports several query languages including SeRQL. Jena [16] provides persistent storage of RDF using relational database. It provides SPARQL query language support for accessing parts of RDF/RDF or OWL and inference capabilities through SPARQL's inference engine. Swoogle [14] is a search and retrieval system for searching ontologies on the web. Swoogle uses a ranking scheme that utilizes relationship weights between Semantic Web Documents (SWD) to model

the probability of being explored. Swoogle allows keyword searches on classes, literals or properties. The system [24] uses a spread activation algorithm to find related instances or literals for a given set of concepts using a initial set of relationship weights. QuizRDF [12] is another search engine that allows keyword searches on annotated documents. The searches in QuizRDF are limited to literals. Beagle++ [34] is a desktop search application that supports RDF path fragment queries and retrieves annotated desktop resources. It uses Lucene to index RDF triples and paths. The system expects the user to have knowledge of the ontology. It takes path sequence queries such as *creator/affiliatedTo MIT* to find all documents whose authors are affiliated to MIT. The systems [9, 12, 14, 24, 34] do not support searches on related words of ontological classes or relationships, unlike our system that supports both. The semantic search proposed in [9] supports keyword based queries on the existing literals in the ontology. The system performs query expansion by navigating through the instances graph using a breadth first search. Thus, the system retrieves the other related information of the keyword by performing instance graph based expansion.

Similar approaches have been proposed for supporting keyword searches over relational [2, 11] and XML [10] databases. However, these approaches often limit their search to the set of literal values i.e. leaves or terminal nodes, e.g. the title of a book or an author's name. The applications [2, 11] retrieve data by repeated joining of the data or tuples associated with the matched fields. Additionally, Banks [2] provides data and schema browsing through interactive displays. XRank [10] allows searches on XML elements or tags. Our system provides keyword searches on RDF documents and hence the challenges are different compared to keyword searches on database or XML documents. Additionally, our system allows searches on related word of classes, relationships.

There exist several applications that either use WordNet [37] or a domain specific ontology [3, 9, 31] for query expansion. We have adopted an integrated approach that uses the knowledge in WordNet and domain specific ontology to expand search terms. Varelas et al [37] approach the problem of retrieving the related documents by computing semantic similarity between the discovered related words in the document and the expanded query terms. Aitken and Reid [3] have developed a search and retrieval system for retrieving documents. It expands the search terms using a domain specific ontology and matches the expanded query terms to the terms in the document. The searches in [3] were confined to the ontological classes and expanded search terms using the *RDF: subClassOf* relationship or hierarchy information in the ontology. OWLIR [31] is a prototype Information Retrieval system that uses an event ontology encoded in DAML+OIL to retrieve documents. The system allows searches on semantically marked-up documents which are annotated using DAML+OIL ontology and incorporates inference services which can answer questions about the explicit and implicit knowledge specified in the ontology. It answers some the known queries by initially providing the name and description of the retrieved result. Later, the user can get more information through a software agent that gathers information from the actual website. This system uses a fixed set of queries (questionnaires) which are parsed and later matched to the ontology classes.

Chapter 9 – Conclusion and Future Work

We have proposed a prototype system that allows keyword searches on direct or related words of classes, relationships and instances in the schema or instance data. We adopted an integrated approach that uses the knowledge in ontology and WordNet along with the lexical processing to expand search terms. We proposed a system that allows users to enter a set of keywords that may or may not be related to each other.

We discussed the two phases that our system uses to retrieve the triples. We discussed how the indices are built during the pre-processing phase and used during the query processing phase. Later, we described all the steps performed during query expansion and addressed different issues involving in matching of search terms to classes and relationships in the ontology. We also discussed the different queries considered by our filtering algorithm.

We compared the results of our system with a keyword search and retrieval system such as Google and calculated the precision for different types of queries. The initially comparison indicated that the results returned by our system were very accurate and relevant to the query. On the other hand, the other system returned many Web documents which weren't relevant to the search. Our system retrieved triples by performing matches onto the ontological classes, relationships or literals. Hence, it is important that entered search terms match literals or are related to the ontological classes or relationships to retrieve relevant meaningful sub-graphs.

Our current system returns a set of triples and a relevant sub-graph, and can be extended to return a set of documents like any other contemporary search system. This can help in reducing the search problems that exist in the contemporary keyword based search and retrieval systems.

REFERENCES

[1] Adams, T., Gearon, P., Wood, D., Kowari. A Platform for Semantic Web Storage and Analysis. http://www.itee.uq.edu.au/~dwood/docs/www2005-kowari.pdf , 2005.

[2] Aditya, B., Bhalotia , G., Chakrabarti , S., Hulgeri , A., Nakhe , C., Parag, and Sudarshan, S. BANKS: Browsing and Keyword Searching in Relational Databases. In proceedings of VLDB, 2002.

[3] Aitken, S, and Reid, S. Evaluation of an Ontology-Based Information retrieval Tool. ECAI'00, Applications of Ontologies and Problem-Solving Methods, 2000.

[4] Aleman-Meza, B., Nagarajan, M., Ramakrishnan, C., Ding, L., Kolari, P., Sheth, A.P., Arpinar, I.B., Joshi, A. and Finin, T., Semantic Analytics on Social Networks: Experiences in Addressing the Problem of Conflict of Interest Detection. In 15th International World Wide Web Conference, (Edinburgh, Scotland, 2006).

[5] B. Hammond, A. Sheth, and K. Kochut. Semantic Enhancement Engine: A Modular Document Enhancement Platform for Semantic Applications over Heterogeneous Content. In V. Kashyap & L. Shklar (Eds.), Real World Semantic Web Applications (pp. 29-49): Ios Pr Inc. 2002.

[6] Berners-Lee, T., Hendler, J., Lassila, O. The Semantic Web.Scientific American. May 17, 2001.

[7] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In Proceedings of the 11th WWW Conference, Honolulu, Hawaii, USA, 2002

[8] Google search engine: *http://www.google.com*

[9] Guha, R., McCool, R., and Miller, E. Semantic Search. Proceedings of WWW 2003.

[10] Guo, L., Shao, F., Botev, C., and Shanmugasundaram, J. XRANK. Ranked keyword search over XML documents. In ACM SIGMOD, 2003.

[11] Hristidis, Vagelis and Papakonstantinou, Yannis: DISCOVER: Keyword search in relational databases. In VLDB, 2002.

[12] J. Davies, R. Weeks, and U. Krohn. QuizRDF. Search technology for the semantic Web. In Workshop on Real World RDF and Semantic Web Applications, WWW, 2002.

[13] Janik, M. and Kochut, K., BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery. In Fourth International Semantic Web Conference, (Galway, Ireland, 2005)

[14] L. Ding, T. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. C. Doshi, and J. Sachs. Swoogle: A search and metadata engine for the semantic Web. In Proceedings of the Thirteenth ACM Conference on Information and Knowledge Management, Washington, DC, Nov. 2004.

[15] Lucene: *http://lucene.apache.org*

[16] McBride, B. Jena: Implementing the RDF Model and Syntax Specification. Proc. of 2nd International Workshop on the Semantic Web, May 2001.

[17] Msn: *http://www.msn.com*

[18] Ontology: *http://www-ksl.stanford.edu/kst/what-is-an-ontology.html*

[19] OWL: *http://www.w3.org/TR/owl-features/*

[20] RDF: *http://www.w3.org/TR/rdf-concepts*

[21] RDF Schema: *http://www.w3.org/TR/rdf-schema*

[22] RDQL: *http://www.w3.org/Submission/2004/SUBM-RDQL-20040109*

[23] Redland: *http://librdf.org*

[24] Rocha, C., Schwabe, D., Poggi de Aragao, M. A Hybrid Approach for Searching in the Semantic Web. WWW2004.

[25] S. Handschuh and S. Staab. Authoring and Annotation of Web Pages in CREAM. In Proceedings of the 11th International World Wide Web Conference, WWW 2002, Honolulu, Hawaii, May 7-11, 2002, pages 462–473. ACM Press, 2002.

[26] S. Handschuh, S. Staab, and F. Ciravegna. S-CREAM –Semi-automatic CREAtion of Metadata. In Proceedings of EKAW 2002, LNCS, pages 358–372, 2002.

[27] Semantic Web: *http://www.w3.org/*

[28] Semantic Web layer cake: *http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html*

[29] SEMDIS: *http://lsdis.cs.uga.edu/projects/semdis/sweto/index.php?page=5*

[30] Sesame: *http://www.openrdf.org*

[31] Shah, U., Finin, T., and Joshi, A. Information Retrieval on the Semantic Web. Proc. of the Eleventh International Conference on Information and Knowledge Management, McLean, VA, USA, 2002.

[32] SPARQL: *http://www.w3.org/TR/rdf-sparql-query*

[33] Stephen Dill, Nadav Eiron, David Gibson, Daniel Gruhl, R. Guha, Anant Jhingran, Tapas Kanungo, Sridhar Rajagopalan, Andrew Tomkins, John A. Tomlin, and Jason Y. Zien. Semtag and seeker: bootstrapping the semantic Web via automated semantic annotation. In Proceedings of the Twelfth International Conference on World Wide Web, pages 178–186. ACM Press, 2003.

[34] T. Iofciu, C. Kohlsch¨utter, W. Nejdl, and R. Paiu. Keywords and RDF fragments. Integrating metadata and full-text search in beagle++. In Proc. of the Semantic Desktop Workshop held at the 4th International Semantic Web Conference, 2005.

[35] Thomas Gruber. It Is What It Does: The Pragmatics of Ontology. Invited presentation to the meeting of the CIDOC Conceptual Reference Model committee, Smithsonian Museum, Washington, D.C., March 26, 2003.

[36] Unicode: *http://en.wikipedia.org/wiki/Unicode*

[37] Varelas, Giannis, Voutsakis, Epimenidis, Raftopoulou, Paraskevi, G.M. Petrakis, Euripides, Evangelos, E. Milios. Semantic Similarity Methods in WordNet and their Application to Information Retrieval on the Web. 7th ACM International Workshop on Web Information and Data Management (WIDM 2005), pp. 10-16, Nov. 5, 2005, Bremen, Germany.

[38] WordNet: *http://wordnet.princeton.edu*

[39] XML: *http://www.w3.org/XML*

[40] XML Schema: *http://www.w3.org/XML/Schema*

[41] Yahoo: *http://www.yahoo.com*

APPENDIX A – Screenshots of some of the searches in our system

Figure 5: Screenshot of our search and retrieval system

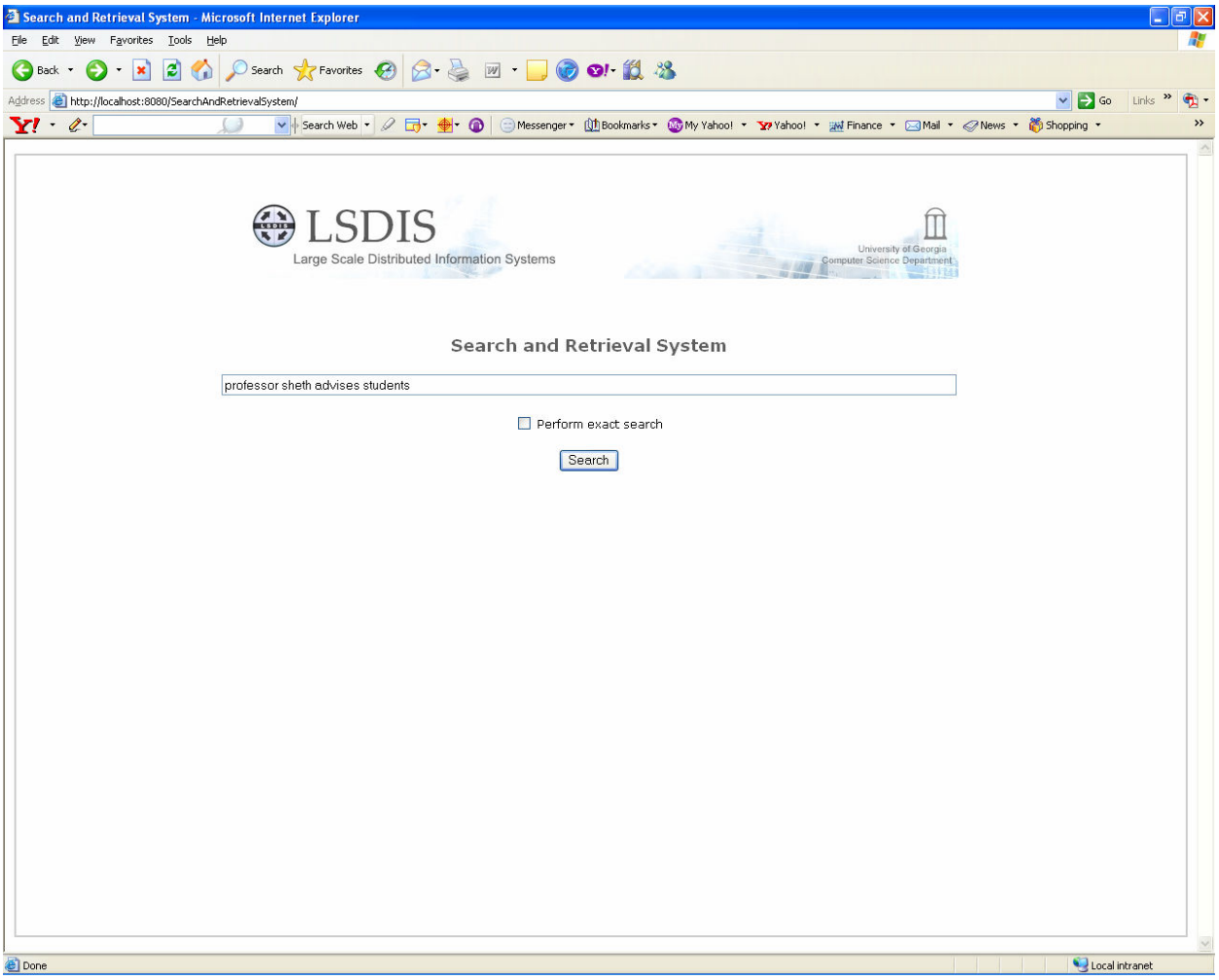


Figure 6: Retrieved triples for the query: *professor sheth advises student*

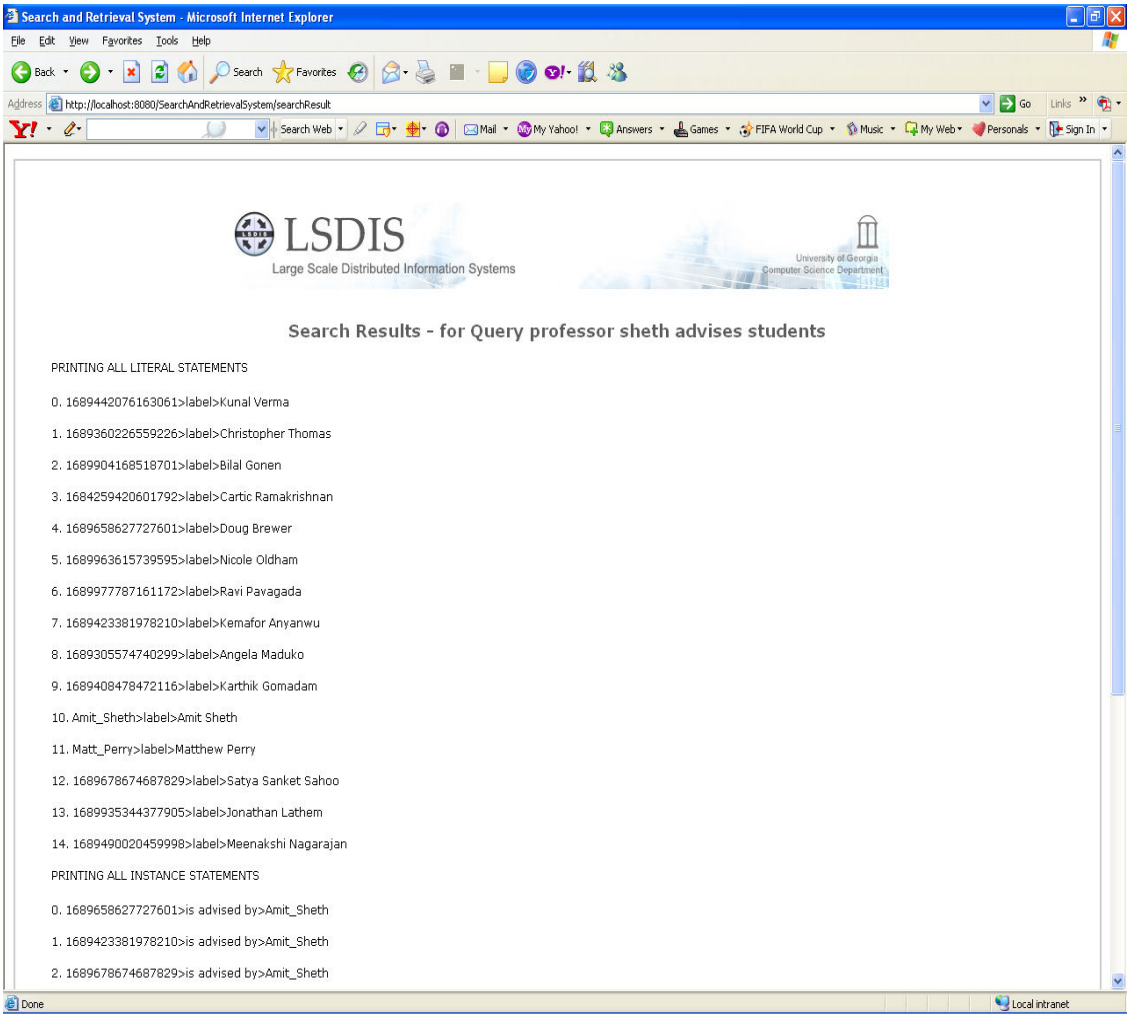


Figure 7: TouchGraph display for the query: *professor sheth advises student*

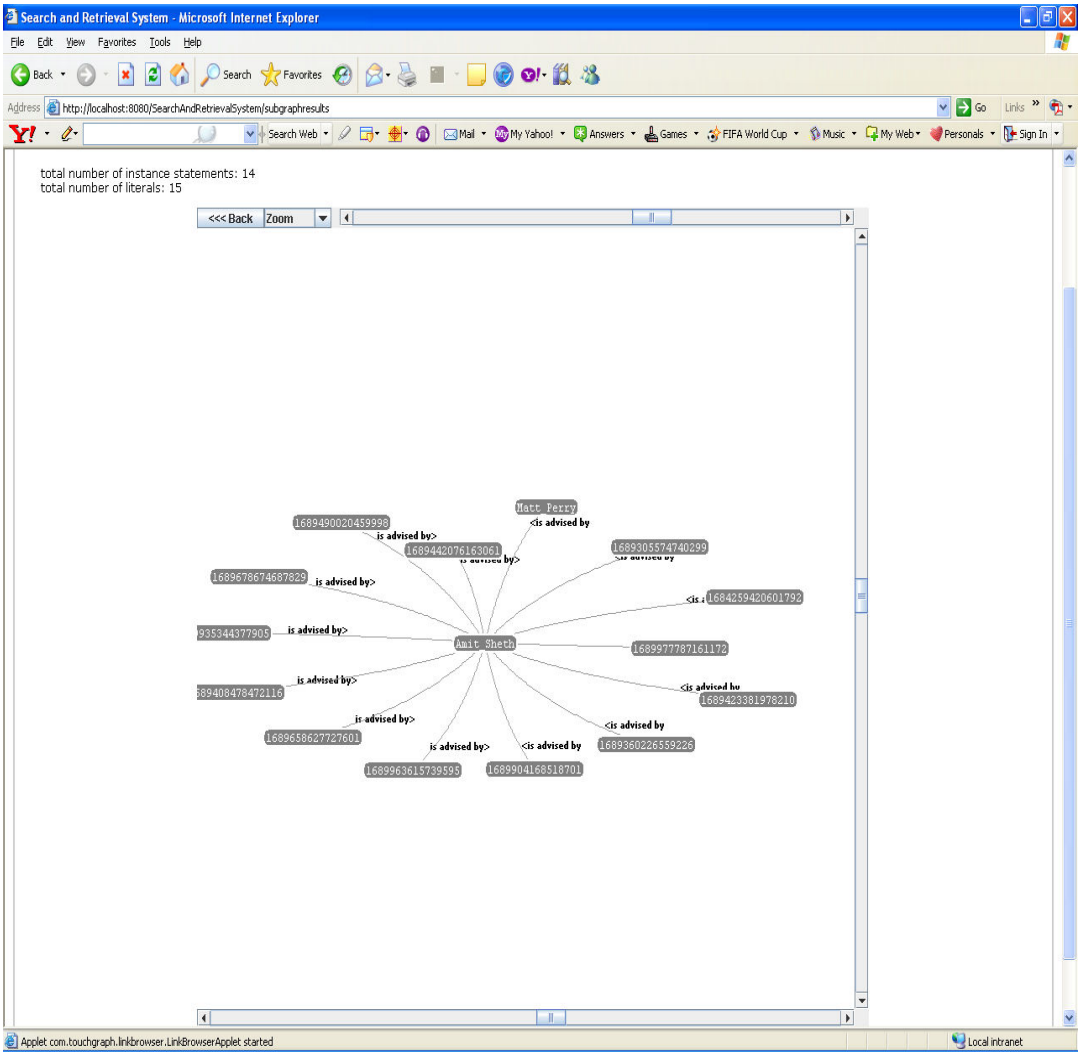


Figure 8: Retrieved triples for the query: *Sheth instructs courses*.

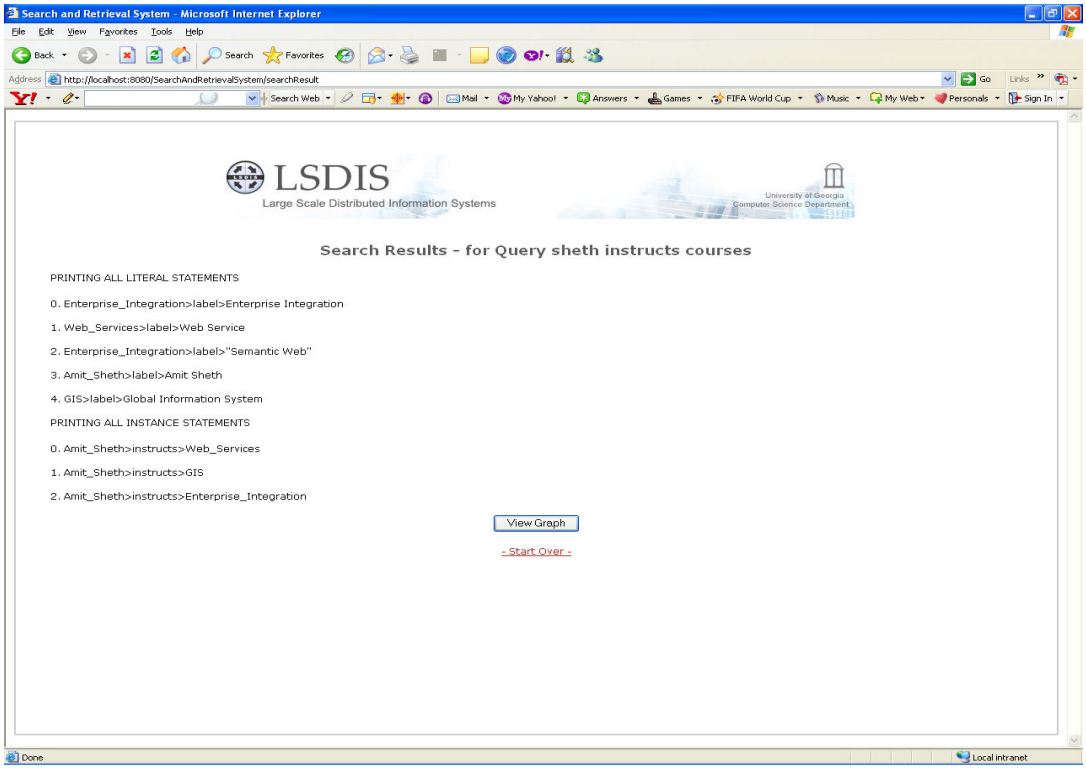


Figure 9: Retrieved triples for the query: *Sheth teaches classes*.

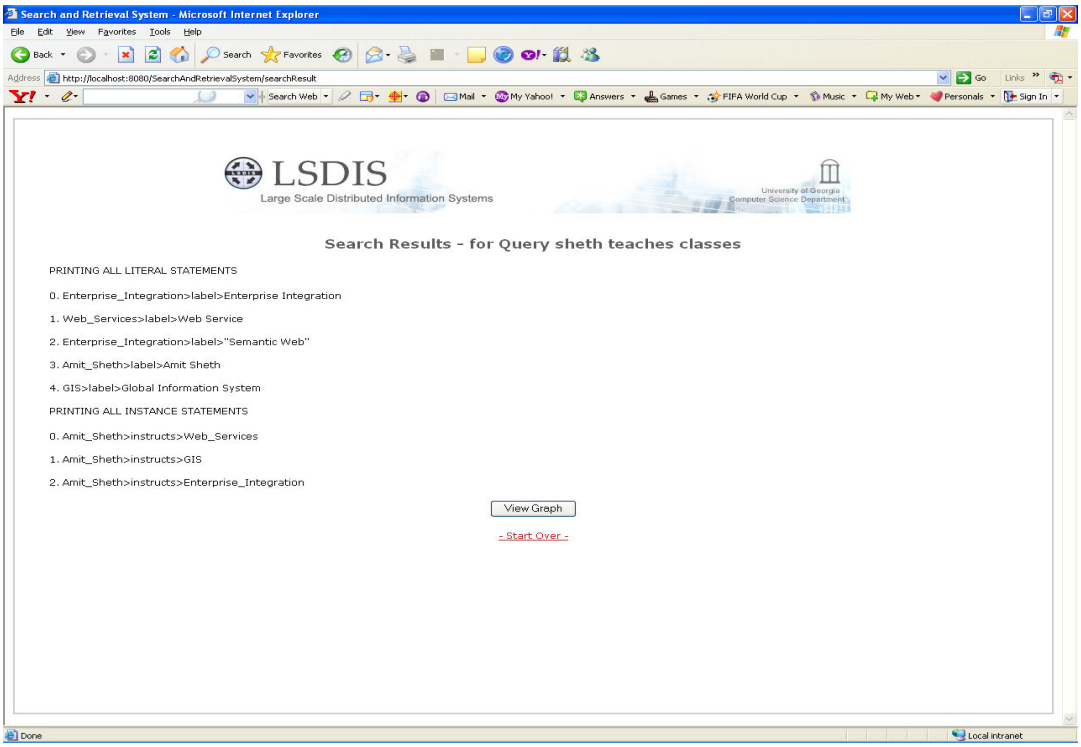
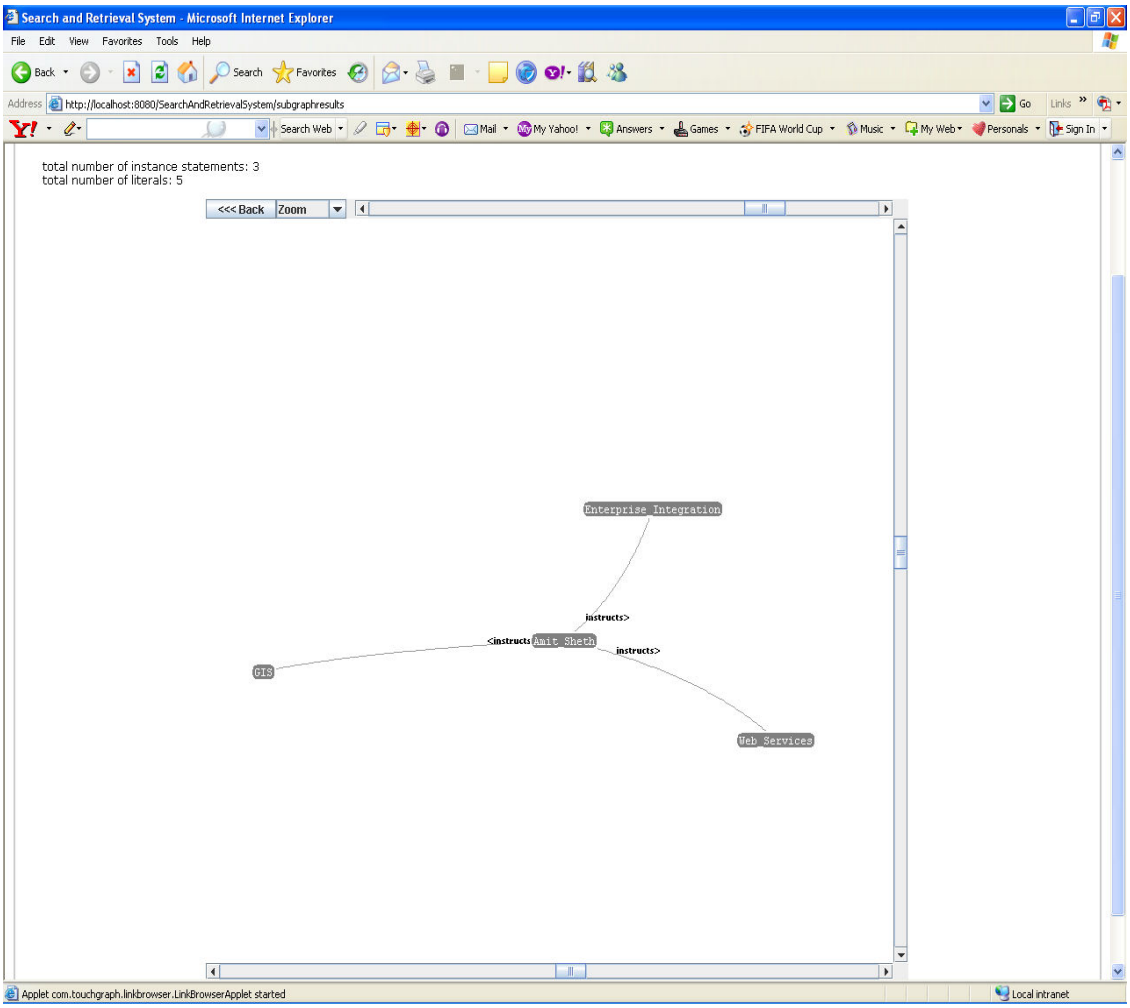


Figure 10: TouchGraph display for the query: *Sheth teaches classes*



APPENDIX B -Glossary of Acronyms

RDF:	Resource Description Framework
RDFS:	Resource Description Framework Schema
OWL:	Web Ontology Language
SPARQL:	Simple Protocol and RDF Query Language
RDQL:	RDF Data Query Language
RQL:	RDF Query Language
SQL:	Structured Query Language
LSDIS:	Large Scale Distributed and Information Systems Lab
SEMDIS:	Semantic Discovery